Microsoft

Microsoft Official Course

# DP-100T01

## Designing and Implementing a Data Science Solution on Azure

# DP-100T01

# Designing and Implementing a Data Science Solution on Azure

1 http://www.microsoft.com/trademarks

**MICROSOFT LICENSE TERMS**

**MICROSOFT INSTRUCTOR-LED COURSEWARE**

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to your use of the content accompanying this agreement which includes the media on which you received it, if any. These license terms also apply to Trainer Content and any updates and supplements for the Licensed Content unless other terms accompany those items. If so, those terms apply.

**BY ACCESSING, DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT ACCESS, DOWNLOAD OR USE THE LICENSED CONTENT.**

**If you comply with these license terms, you have the rights below for each license you acquire.**

1. **DEFINITIONS.**
    1. “Authorized Learning Center” means a Microsoft Imagine Academy (MSIA) Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
    2. “Authorized Training Session” means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.
    3. “Classroom Device” means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center’s training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
    4. “End User” means an individual who is (i) duly enrolled in and attending an Authorized Training Session or Private Training Session, (ii) an employee of an MPN Member (defined below), or (iii) a Microsoft full-time employee, a Microsoft Imagine Academy (MSIA) Program Member, or a Microsoft Learn for Educators – Validated Educator.
    5. “Licensed Content” means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
    6. “Microsoft Certified Trainer” or “MCT” means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.
    7. “Microsoft Instructor-Led Courseware” means the Microsoft-branded instructor-led training course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics, or Microsoft Business Group courseware.
    8. “Microsoft Imagine Academy (MSIA) Program Member” means an active member of the Microsoft Imagine Academy Program.
    9. “Microsoft Learn for Educators – Validated Educator” means an educator who has been validated through the Microsoft Learn for Educators program as an active educator at a college, university, community college, polytechnic or K-12 institution.
    10. “Microsoft Learning Competency Member” means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
    11. “MOC” means the “Official Microsoft Learning Product” instructor-led courseware known as Microsoft Official Course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies.
    12. “MPN Member” means an active Microsoft Partner Network program member in good standing.

13. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
14. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
15. "Trainer" means (i) an academically accredited educator engaged by a Microsoft Imagine Academy Program Member to teach an Authorized Training Session, (ii) an academically accredited educator validated as a Microsoft Learn for Educators – Validated Educator, and/or (iii) a MCT.
16. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.

2. **USE RIGHTS.** The Licensed Content is licensed, not sold. The Licensed Content is licensed on a **one copy per user basis**, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

- 2.1 Below are five separate sets of use rights. Only one set of rights apply to you.

  1. **If you are a Microsoft Imagine Academy (MSIA) Program Member:**

     1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
     2. For each license you acquire on behalf of an End User or Trainer, you may either:

        1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User who is enrolled in the Authorized Training Session, and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
        2. provide one (1) End User with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
        3. provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content.

     3. For each license you acquire, you must comply with the following:

        1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
        2. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
        3. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End

User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified Trainers who have in-depth knowledge of and experience with the Microsoft technology that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Authorized Training Sessions,
6. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
7. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

2. **If you are a Microsoft Learning Competency Member:**
   1. Each license acquire may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you.  If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
   2. For each license you acquire on behalf of an End User or MCT, you may either:
      1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Authorized Training Session and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware provided, **or**
      2. provide one (1) End User attending the Authorized Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
      3. you will provide one (1) MCT with the unique redemption code and instructions on how they can access one (1) Trainer Content.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. you will ensure that each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
      3. you will ensure that each End User provided with a hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

4. you will ensure that each MCT teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. you will only use qualified MCTs who also hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Authorized Training Sessions using MOC,
6. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
7. you will only provide access to the Trainer Content to MCTs.

3. **If you are a MPN Member:**
   1. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
   2. For each license you acquire on behalf of an End User or Trainer, you may either:
      1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Private Training Session, and only immediately prior to the commencement of the Private Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
      2. provide one (1) End User who is attending the Private Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
      3. you will provide one (1) Trainer who is teaching the Private Training Session with the unique redemption code and instructions on how they can access one (1) Trainer Content.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. you will ensure that each End User attending an Private Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Private Training Session,
      3. you will ensure that each End User provided with a hard copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
      4. you will ensure that each Trainer teaching an Private Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Private Training Session,

5. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Private Training Sessions,

6. you will only use qualified MCTs who hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Private Training Sessions using MOC,

7. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and

8. you will only provide access to the Trainer Content to Trainers.

4. **If you are an End User:**
For each license you acquire, you may use the Microsoft Instructor-Led Courseware solely for your personal training use. If the Microsoft Instructor-Led Courseware is in digital format, you may access the Microsoft Instructor-Led Courseware online using the unique redemption code provided to you by the training provider and install and use one (1) copy of the Microsoft Instructor-Led Courseware on up to three (3) Personal Devices. You may also print one (1) copy of the Microsoft Instructor-Led Courseware. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

5. **If you are a Trainer.**

   1. For each license you acquire, you may install and use one (1) copy of the Trainer Content in the form provided to you on one (1) Personal Device solely to prepare and deliver an Authorized Training Session or Private Training Session, and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Trainer Content. You may not install or use a copy of the Trainer Content on a device you do not own or control. You may also print one (1) copy of the Trainer Content solely to prepare for and deliver an Authorized Training Session or Private Training Session.

   2. If you are an MCT, you may customize the written portions of the Trainer Content that are logically associated with instruction of a training session in accordance with the most recent version of the MCT agreement.

   3. If you elect to exercise the foregoing rights, you agree to comply with the following: (i) customizations may only be used for teaching Authorized Training Sessions and Private Training Sessions, and (ii) all customizations will comply with this agreement. For clarity, any use of "customize" refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

- 2.2 **Separation of Components.** The Licensed Content is licensed as a single unit and you may not separate their components and install them on different devices.
- 2.3 **Redistribution of Licensed Content.** Except as expressly provided in the use rights above, you may not distribute any Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.
- 2.4 **Third Party Notices.** The Licensed Content may include third party code that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code are included for your information only.
- 2.5 **Additional Terms.** Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to your use of that respective component and supplements the terms described in this agreement.

3. **LICENSED CONTENT BASED ON PRE-RELEASE TECHNOLOGY.** If the Licensed Content's subject matter is based on a pre-release version of Microsoft technology ("**Pre-release**"), then in addition to the other provisions in this agreement, these terms also apply:
   1. **Pre-Release Licensed Content.** This Licensed Content subject matter is on the Pre-release version of the Microsoft technology. The technology may not work the way a final version of the technology will and we may change the technology for the final version. We also may not release a final version. Licensed Content based on the final version of the technology may not contain the same information as the Licensed Content based on the Pre-release version. Microsoft is under no obligation to provide you with any further content, including any Licensed Content based on the final version of the technology.
   2. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft technology, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its technology, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.
   3. **Pre-release Term.** If you are an Microsoft Imagine Academy Program Member, Microsoft Learning Competency Member, MPN Member, Microsoft Learn for Educators – Validated Educator, or Trainer, you will cease using all copies of the Licensed Content on the Pre-release technology upon (i) the date which Microsoft informs you is the end date for using the Licensed Content on the Pre-release technology, or (ii) sixty (60) days after the commercial release of the technology that is the subject of the Licensed Content, whichever is earliest ("**Pre-release term**"). Upon expiration or termination of the Pre-release term, you will irretrievably delete and destroy all copies of the Licensed Content in your possession or under your control.
4. **SCOPE OF LICENSE.** The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
   - access or allow any individual to access the Licensed Content if they have not acquired a valid license for the Licensed Content,
   - alter, remove or obscure any copyright or other protective notices (including watermarks), branding or identifications contained in the Licensed Content,
   - modify or create a derivative work of any Licensed Content,
   - publicly display, or make the Licensed Content available for others to access or use,
   - copy, print, install, sell, publish, transmit, lend, adapt, reuse, link to or post, make available or distribute the Licensed Content to any third party,
   - work around any technical limitations in the Licensed Content, or
   - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation.
5. **RESERVATION OF RIGHTS AND OWNERSHIP.** Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property

laws and treaties.  Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content.

6. **EXPORT RESTRICTIONS.** The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.
7. **SUPPORT SERVICES.** Because the Licensed Content is provided "as is", we are not obligated to provide support services for it.
8. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon termination of this agreement for any reason, you will immediately stop all use of and delete and destroy all copies of the Licensed Content in your possession or under your control.
9. **LINKS TO THIRD PARTY SITES.**  You may link to third party sites through the use of the Licensed Content.  The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites.  Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites.  Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.
10. **ENTIRE AGREEMENT.** This agreement, and any additional terms for the Trainer Content, updates and supplements are the entire agreement for the Licensed Content, updates and supplements.
11. **APPLICABLE LAW.**
    1. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
    2. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.
12. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.
13. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT AND ITS RESPECTIVE AFFILIATES GIVES NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT AND ITS RESPECTIVE AFFILIATES EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**
14. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT, ITS RESPECTIVE AFFILIATES AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO US$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.**

This limitation applies to

- anything related to the Licensed Content, services, content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential, or other damages.

**Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.**

**Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.**

**EXONÉRATION DE GARANTIE.** Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contre-façon sont exclues.

**LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES.** Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 $ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers; et.
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

**EFFET JURIDIQUE.** Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Revised April 2019

# Contents

# Module 0 Welcome

## Welcome to the Course

### Course Introduction

Welcome to this course on Designing and Implementing Data Science Solutions on Microsoft Azure.

In this course, you will learn how to use Azure Machine Learning to operate machine learning workloads in the cloud. As you work through the material and hands-on exercises in this course, you will build on your existing data science and machine learning knowledge and learn how to leverage cloud services to perform machine learning at scale.

The course assumes that you are familiar with Python, and have experience of training machine learning models using common frameworks such as:

- Scikit-Learn
- PyTorch
- TensorFlow

After completing the course, you will be able to:

- Create an Azure Machine Learning workspace, and manage compute, data, and coding environments for machine learning workloads
- Use Azure Machine Learning for "no-code" machine learning model training and deployment.
- Create and run experiments that log metrics and train machine learning models.
- Create and manage datastores and datasets, and use data in machine learning experiments.
- Create and manage compute resources, and use them to run machine learning experiments at scale in the cloud.
- Use Pipelines to orchestrate machine learning operations.
- Deploy predictive models as real-time or batch inference services, and consume them from client applications.

- Find the optimal model for your data by using hyperparameter tuning and automated machine learning.
- Apply principles and techniques that support responsible machine learning practices.
- Monitor usage and data drift for deployed models.

## Course Agenda

This course includes the following modules

### Module 1: Getting Started with Azure Machine Learning

In this module, you will learn how to provision an Azure Machine Learning workspace and use it to manage machine learning assets such as data, compute, model training code, logged metrics, and trained models. You will learn how to use the web-based Azure Machine Learning *studio* interface as well as the Azure Machine Learning SDK and developer tools like Visual Studio Code and Jupyter Notebooks to work with the assets in your workspace.

### Module 2: Visual Tools for Machine Learning

This module introduces the *Automated Machine Learning* and *Designer* visual tools, which you can use to train, evaluate, and deploy machine learning models without writing any code.

### Module 3: Running Experiments and Training Models

In this module, you will get started with *experiments* that encapsulate data processing and model training code, and use them to train machine learning models.

### Module 4: Working with Data

Data is a fundamental element in any machine learning workload, so in this module, you will learn how to create and manage datastores and datasets in an Azure Machine Learning workspace, and how to use them in model training experiments.

### Module 5: Working with Compute

One of the key benefits of the cloud is the ability to leverage compute resources on demand, and use them to scale machine learning processes to an extent that would be infeasible on your own hardware. In this module, you'll learn how to manage experiment *environments* that ensure consistent runtime consistency for experiments, and how to create and use compute targets for experiment runs.

### Module 6: Orchestrating Machine Learning Workflows

Now that you understand the basics of running workloads as experiments that leverage data assets and compute resources, it's time to learn how to orchestrate these workloads as *pipelines* of connected steps. Pipelines are key to implementing an effective Machine Learning Operationalization (ML Ops) solution in Azure, so you'll explore how to define and run them in this module.

## Module 7: Deploying and Consuming Models

Models are designed to help decision making through predictions, so they're only useful when deployed and available for an application to consume. In this module learn how to deploy models for real-time inferencing, and for batch inferencing.

## Module 8: Training Optimal Models

By this stage of the course, you've learned the end-to-end process for training, deploying, and consuming machine learning models; but how do you ensure your model produces the best predictive outputs for your data? In this module, you'll explore how you can use the azure Machine Learning SDK to apply hyperparameter tuning and automated machine learning, and find the best model for your data.

## Module 9: Responsible Machine Learning

Data scientists have a duty to ensure they analyze data and train machine learning models responsibly; respecting individual privacy, mitigating bias, and ensuring transparency. This module explores some considerations and techniques for applying responsible machine learning principles.

## Module 10: Monitoring Models

After a model has been deployed, it's important to understand how the model is being used in production, and to detect any degradation in its effectiveness due to *data drift*. This module describes techniques for monitoring models and their data.

# Lab Environment

This course includes extensive hands-on activities designed to help you learn by working with Azure Machine Learning. To complete the labs in this course, you will need:

- A modern web browser - for example, Microsoft Edge.
- The lab files for this course, which are published online at **https://aka.ms/mslearn-dp100**.
- A **Microsoft Azure**[1] subscription.

If you are taking this course with a Microsoft Learning Partner, you can use an "Azure Pass" to claim a free temporary Azure subscription. Redeem your Azure Pass code at **https://www.microsoftazurepass.com**, signing in with a Microsoft account that hasn't been used to redeem an Azure Pass previously.

You can complete the labs on your own computer. In some classes, a hosted environment may be available - check with your instructor.

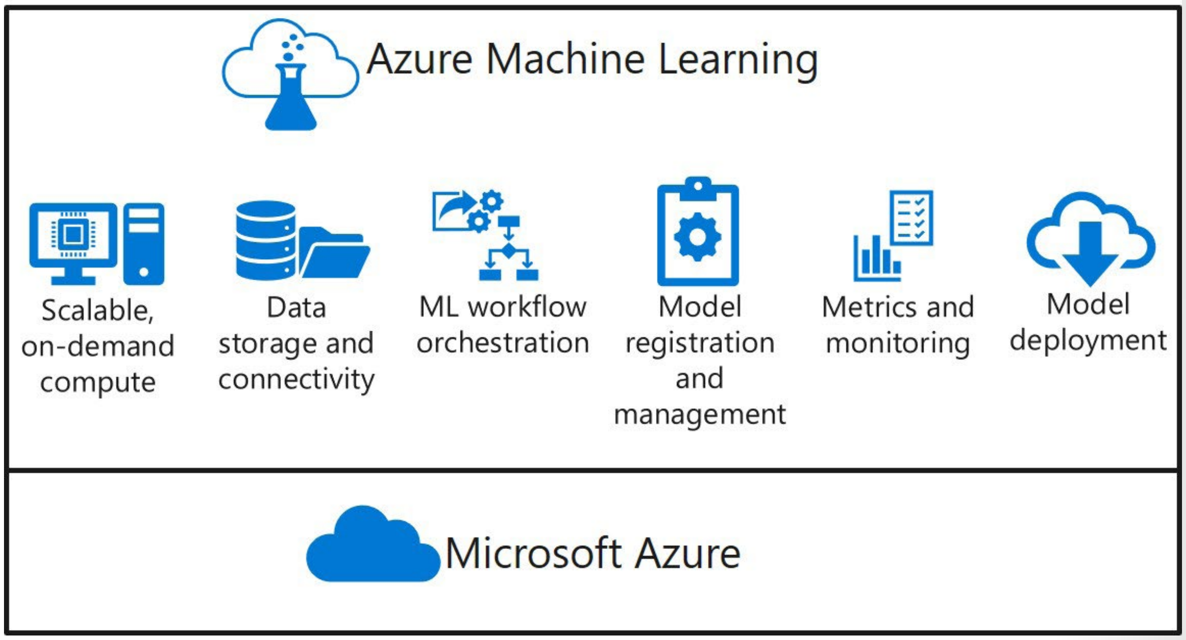
1 https://azure.microsoft.com

# Module 1 Getting Started with Azure Machine Learning

## Introduction to Azure Machine Learning

### What is Azure Machine Learning?

Azure Machine Learning is a platform for operating machine learning workloads in the cloud.


Azure Machine Learning
Scalable, on-demand compute
Data storage and connectivity
ML workflow orchestration
Model registration and management
Metrics and monitoring
Model deployment
Microsoft Azure

Built on the Microsoft Azure cloud platform, Azure Machine Learning enables you to manage:

- Scalable on-demand compute for machine learning workloads.
- Data storage and connectivity to ingest data from a wide range sources.

- Machine learning workflow orchestration to automate model training, deployment, and management processes.
- Model registration and management, so you can track multiple versions of models and the data on which they were trained.
- Metrics and monitoring for training experiments, datasets, and published services.
- Model deployment for real-time and batch inferencing.

## Azure Machine Learning in Context

Now that you know a little about the capabilities Azure Machine Learning offers, it can be helpful to understand where those capabilities fit into an overall context for machine learning in Azure. Fundamentally, machine learning is about taking data and using it to train and deliver predictive models that support ML-powered applications.

In Azure, there are many services for storing data, including Azure Storage, Azure Data Lake Store, Azure SQL Database, Azure Cosmos DB, and others. There are also services that you can use to build "big data" processing solutions that transfer and transform data, including Azure Data Factory and Apache Spark engines in Azure HDInsight and Azure Databricks.

Azure also provides a huge array of services you can use to deliver applications for Web, mobile, and IoT devices; including Azure App Service, Azure Functions, and Azure IoT Edge. Azure also offers services for container-based deployment through services like Azure Container Services and Azure Kubernetes Services.

Azure Machine Learning provides a platform for operationalizing the workloads needed to drive the iterative process that enables delivery of machine learning applications built on data. There are three primary types of user that Azure Machine Learning supports in this process:

- Data Scientists, who use their knowledge of statistics and data analytics to conduct analytical experiments and train machine learning models. These users typically work in Python or R, and use frameworks such as Scikit-Learn, PyTorch, and TensorFlow to train machine learning models.
- "Citizen" Data Scientists and App Developers, who don't primarily work in the field of statistical data analysis, but who need to train machine learning models to support applications. These users can take advantage of graphical tools that abstract the underlying complexity of model training.
- Software engineers and operators, who need to operationalize machine learning to support applications and services. Their tasks typically involve using scripts or automated DevOps processes to manage model retraining and deployment, as well as overall application monitoring and troubleshooting.

## Machine Learning Operationalization (ML Ops)

Increasingly, enterprises are seeking to integrate machine learning model training, deployment, and management into existing processes for developing and delivering software. What's become known as "Machine Learning Operationalization", or "ML Ops" has become part of the larger move towards "Development/Operations" or "DevOps".

DevOps is a combination of best practices for team collaboration and operational automation that helps drive efficiency in creating, deploying, and managing software solutions at enterprise scale. Often, data scientists and machine learning specialists work in relative isolation from the software developers who integrate their models into applications and the system administrators who manage the deployed application infrastructure, but by adopting some DevOps principles, these disparate teams can coordinate activities to create an effective overall solution.

Azure Machine Learning provides multiple capabilities that an organization can leverage for ML Ops, and integrates with more general DevOps tools such as Azure DevOps and GitHub. You'll explore these capabilities in more depth in the remainder of this course.

## Azure Machine Learning Workspaces

A workspace is a context for the experiments, data, compute targets, and other assets associated with a machine learning workload.

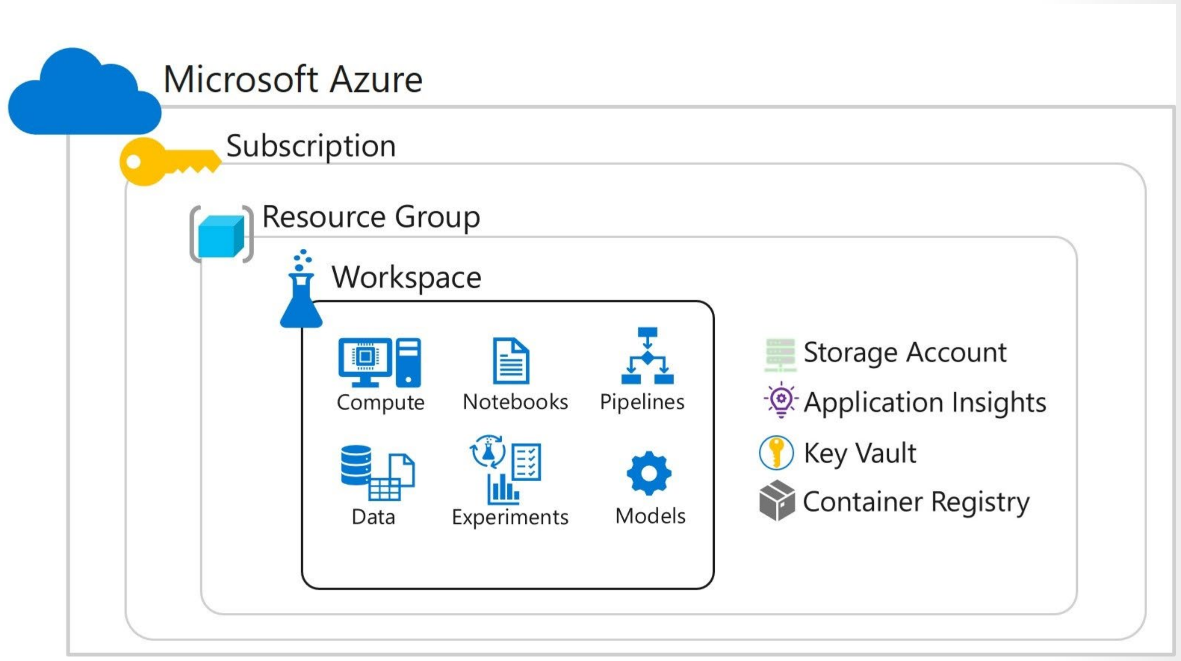
### Workspaces for Machine Learning Assets

A workspace defines the boundary for a set of related machine learning assets. You can use workspaces to group machine learning assets based on projects, deployment environments (for example, test and production), teams, or some other organizing principle. The assets in a workspace include:

- Compute targets for development, training, and deployment.
- Data for experimentation and model training.
- Notebooks containing shared code and documentation.
- Experiments, including run history with logged metrics and outputs.
- Pipelines that define orchestrated multi-step processes.
- Models that you have trained.

### Workspaces as Azure Resources

Workspaces are Azure resources, and as such they are defined within a resource group in an Azure subscription, along with other related Azure resources that are required to support the workspace.



The Azure resources created alongside a workspace include:

- A storage account - used to store files used by the workspace as well as data for experiments and model training.
- An Application Insights instance, used to monitor predictive services in the workspace.
- An Azure Key Vault instance, used to manage secrets such as authentication keys and credentials used by the workspace.
- A container registry, created as-needed to manage containers for deployed models.

### Creating a Workspace

You can create a workspace in any of the following ways:

- In the Microsoft Azure portal, create a new **Machine Learning** resource, specifying the subscription, resource group and workspace name.
- Use the Azure Machine Learning Python SDK to run code that creates a workspace. For example:

```
from azureml.core import Workspace

ws = Workspace.create(name='aml-workspace', 
                  subscription_id='123456-abc-123...',
                  resource_group='aml-resources',
                  create_resource_group=True,
                  location='eastus'
                 )
```

- Use the Azure Command Line Interface (CLI) with the Azure Machine Learning CLI extension. For example, you could use the following command (which assumes a resource group named *aml-resources* has already been created):

```
az ml workspace create -w 'aml-workspace' -g 'aml-resources'
```

- Create an Azure resource Manager (ARM) template. For more information the ARM template format for an Azure Machine Learning workspace, see the **Azure Machine Learning documentation**[1].
- Use the Azure Machine Learning REST interface, see the **Azure Machine Learning documentation**[2].

## Access Controls and Permissions

As with any cloud resource, you need to ensure that only authorized users can access and work with assets in an Azure Machine Learning workspace.

For the most part, access to Azure Machine Learning workspace resources is managed through Azure Active Directory (AAD) role-based access control (RBAC). Typically, a user (or a service principal for an application) is authenticated by Azure Active Directory and receives an access token, this token is then used to request access to individual resources, and access is granted (or denied) based on membership of roles that have permission to perform specific actions. The default roles defined for an Azure Machine

1 https://aka.ms/AA70rq4
2 https://aka.ms/AAaexxy

Learning workspace, and some of their most important permissions, are shown in the following table – but you can add custom roles and set custom permissions.

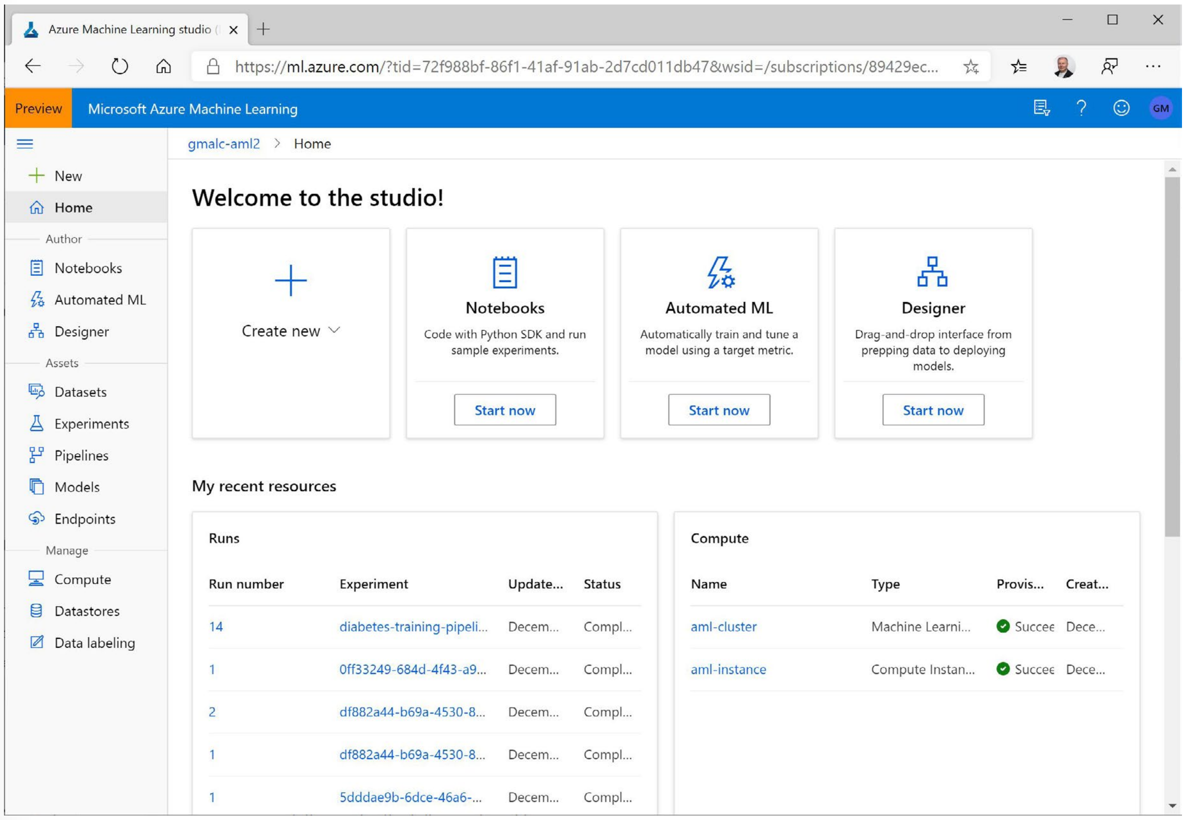
| Permission | Owner | Contributor | Reader |
|---|---|---|---|
| Create workspace | X | X | |
| Share workspace | X | | |
| Create compute target | X | X | |
| Attach compute target | X | X | |
| Attach data stores | X | X | |
| Run experiments | X | X | |
| View runs / metrics | X | X | X |
| Register model | X | X | |
| Create image | X | X | |
| Deploy web service | X | X | |
| View models / images | X | X | X |
| Call web service | X | X | X |

Some resources within an Azure Machine Learning workspace support alternative access methods. For example, compute resources (which we'll explore later) can be configured to allow access via secure shell (SSH); and when you deploy a machine learning model as a service (which again, we'll explore later), you can enable access using an AAD token or by specifying an access key.

## Working with Azure Machine Learning

### Azure Machine Learning studio

You can manage the assets in your Azure Machine Learning workspace in the Azure portal, but as this is a general interface for managing all kinds of resources in Azure, data scientists and other users involved in machine learning operations may prefer to use a more focused, dedicated interface.



Code with Python SDK and run sample experiments.

Automatically train and tune a model using a target metric.

Drag-and-drop interface from prepping data to deploying models.

Runs

| Run number | Experiment | Update... | Status |
|---|---|---|---|
| 14 | diabetes-training-pipeli... | Decem... | Compl... |
| 1 | 0ff33249-684d-4f43-a9... | Decem... | Compl... |
| 2 | df882a44-b69a-4530-8... | Decem... | Compl... |
| 1 | df882a44-b69a-4530-8... | Decem... | Compl... |
| 1 | 5dddae9b-6dce-46a6-... | Decem... | Compl... |

Compute

| Name | Type | Provis... | Creat... |
|---|---|---|---|
| aml-cluster | Machine Learni... | ✓ Succee | Dece... |
| aml-instance | Compute Instan... | ✓ Succee | Dece... |

Azure Machine Learning studio is a web-based tool for managing an Azure Machine Learning workspace. It enables you to create, manage, and view all of the assets in your workspace and provides the following graphical tools:

- *Automated ML*: A wizard interface that enables you to train a model using a combination of algorithms and data preprocessing techniques to find the best model for your data.
- *Designer*: A drag and drop interface for "no code" machine learning model development.

**Note**: A previously released tool named Azure Machine Learning Studio provided a free service for drag and drop machine learning model development. The studio interface for the Azure Machine Learning service includes this capability in the *Designer* tool, as well as other workspace asset management capabilities.

### Using Azure Machine Learning studio

To use Azure Machine Learning studio, use a a web browser to navigate to **https://ml.azure.com** and sign in using credentials associated with your Azure subscription. You can then select the subscription and workspace you want to manage.

## The Azure Machine Learning SDK for Python

While graphical interfaces like Azure Machine Learning studio make it easy to create and manage machine learning assets, it is often advantageous to use a code-based approach to managing resources. By writing scripts to create and manage resources, you can:

- Automate asset creation and configuration to make it repeatable.
- Ensure consistency for resources that must be replicated in multiple environments (for example, development, test, and production)
- Incorporate machine learning asset configuration into developer operations (*DevOps*) workflows, such as continuous integration / continuous deployment (CI/CD) pipelines.

Azure Machine Learning provides software development kits (SDKs) for Python and R, which you can use to create, manage, and use assets in an Azure Machine Learning workspace.

**Note**: This course focuses on the Python SDK because it has broader capabilities than the R SDK, which is in preview at the time of writing.

### Installing the Azure Machine Learning SDK for Python

You can install the Azure Machine Learning SDK for Python by using the `pip` package management utility, as shown in the following code sample:

```
pip install azureml-sdk
```

The SDK is installed using the Python pip utility, and consists of the main **azureml-sdk** package as well as numerous other ancillary packages that contain specialized functionality. For example, the **azureml-widgets** package provides support for interactive widgets in a Jupyter notebook environment. To install additional packages, include them in the `pip install` command:

```
pip install azureml-sdk azureml-widgets
```

**More Information**: For more information about installing the Azure Machine Learning SDK for Python, see the **SDK documentation**[3]. Also, you should be aware that the SDK is updated on a regular basis, and review the **release notes for the latest release**[4].

### Connecting to a Workspace

After installing the SDK package in your Python environment, you can write code to connect to your workspace and perform machine learning operations. The easiest way to connect to a workspace is to use a workspace configuration file, which includes the Azure subscription, resource group, and workspace details as shown here:

```
{
    "subscription_id": "1234567-abcde-890-fgh...",
    "resource_group": "aml-resources",
    "workspace_name": "aml-workspace"
}
```

3 https://aka.ms/AA70rq7
4 https://aka.ms/AA70zel

**Tip**: You can download a configuration file for a workspace from the **Overview** page of its blade in the Azure portal or from Azure Machine Learning studio.

To connect to the workspace using the configuration file, you can use the **from_config** method of the **Workspace** class in the SDK, as shown here:

```
from azureml.core import Workspace

ws = Workspace.from_config()
```

By default, the **from_config** method looks for a file named **config.json** in the folder containing the Python code file, but you can specify another path if necessary.

As an alternative to using a configuration file, you can use the **get** method of the **Workspace** class with explicitly specified subscription, resource group, and workspace details as shown here - though the configuration file technique is generally preferred due to its greater flexibility when using multiple scripts:

```
from azureml.core import Workspace

ws = Workspace.get(name='aml-workspace',
                   subscription_id='1234567-abcde-890-fgh...',
                   resource_group='aml-resources')
```

Whichever technique you use, if there is no current active session with your Azure subscription, you will be prompted to authenticate.

### Working with the Workspace Class

The **Workspace** class is the starting point for most code operations. For example, you can use its **compute_targets** attribute to retrieve a dictionary object containing the compute targets defined in the workspace, like this:

```
for compute_name in ws.compute_targets:
    compute = ws.compute_targets[compute_name]
    print(compute.name, ":", compute.type)
```

The SDK contains a rich library of classes that you can use to create, manage, and use many kinds of asset in an Azure Machine Learning workspace.

**More Information**: For more information about the Azure Machine Learning SDK, see the **SDK documentation**[5].

## The Azure Machine Learning CLI Extension

The Azure command-line interface (CLI) is a cross-platform command-line tool for managing Azure resources. The Azure Machine Learning CLI extension is an additional package that provides commands for working with Azure Machine Learning.

5 https://aka.ms/AA70zeq

## Installing the Azure Machine Learning CLI Extension

To install the Azure Machine Learning CLI extension, you must first install the Azure CLI. See the **full installation instructions for all supported platforms**[6] for more details.

After installing the Azure CLI, you can add the Azure Machine Learning CLI extension by running the following command:

```
az extension add -n azure-cli-ml
```

## Using the Azure Machine Learning CLI Extension

To use the Azure Machine Learning CLI extension, run the `az ml` command with the appropriate parameters for the action you want to perform. For example, to list the compute targets in a workspace, run the following command:

```
az ml computetarget list -g 'aml-resources' -w 'aml-workspace'
```

**Note**: In the code sample above, the **-g** parameter specifies the name of the resource group in which the Azure Machine Learning workspace specified in the **-w** parameter is defined. These parameters are shortened aliases for **–resource-group** and **–workspace-name**.

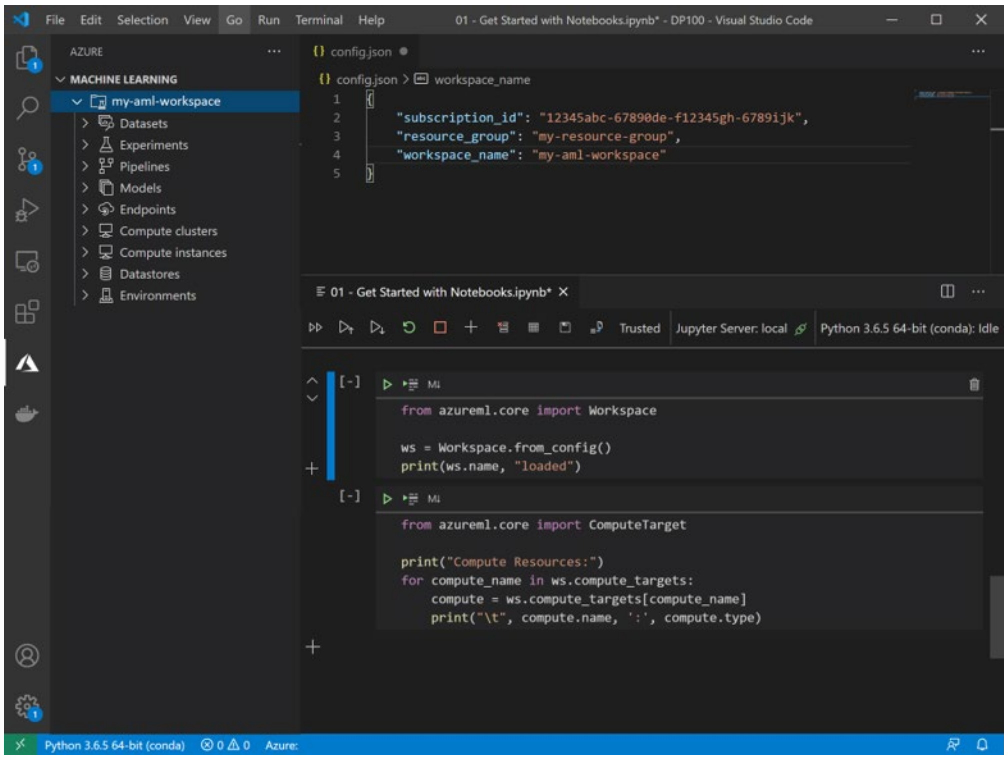
**More Information**: For more information about the Azure Machine Learning CLI Extension, see the **documentation**[7].

## The Azure Machine Learning Extension for Visual Studio Code

Visual Studio Code is a lightweight code editing environment for Microsoft Windows, Apple macOS, and Linux. It provides a visual interface for many kinds of code, including Microsoft C#, Javascript, Python and others; as well as intellisense and syntax formatting for common data formats such as JSON and XML.

Visual Studio Code's flexibility is based on the ability to install modular *extensions* that add syntax checking, debugging, and visual management interfaces for specific workloads. For example, the Microsoft Python extension adds support for writing and running Python code in scripts or notebooks within the Visual Studio Code interface.

6 https://aka.ms/AA70zet
7 https://aka.ms/AA70zeu


File Edit Selection View Go Run Terminal Help
01 - Get Started with Notebooks.ipynb* - DP100 - Visual Studio Code
AZURE
MACHINE LEARNING
my-aml-workspace
Datasets
Experiments
Pipelines
Models
Endpoints
Compute clusters
Compute instances
Datastores
Environments
config.json
config.json > workspace_name
01 - Get Started with Notebooks.ipynb*
Trusted
Jupyter Server: local
Python 3.6.5 64-bit (conda): Idle
Python 3.6.5 64-bit (conda)
Azure:

```
{
    "subscription_id": "12345abc-67890de-f12345gh-6789ijk",
    "resource_group": "my-resource-group",
    "workspace_name": "my-aml-workspace"
}
```

```
[-]
from azureml.core import Workspace

ws = Workspace.from_config()
print(ws.name, "loaded")
```

```
[-]
from azureml.core import ComputeTarget

print("Compute Resources:")
for compute_name in ws.compute_targets:
    compute = ws.compute_targets[compute_name]
    print("\t", compute.name, ':', compute.type)
```

## The Azure Machine Learning Extension

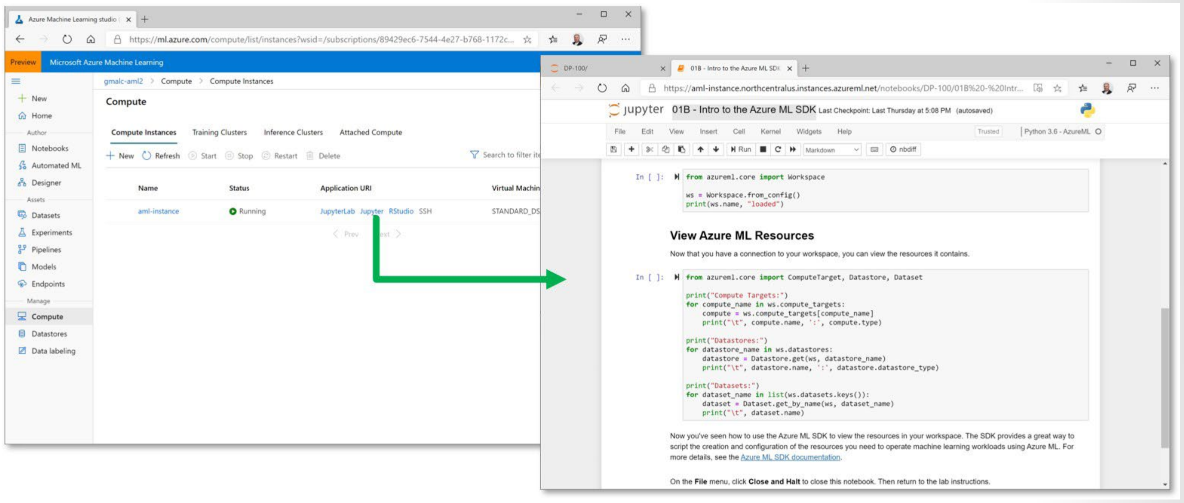
The Azure Machine Learning Extension for Visual Studio Code provides a graphical interface for working with assets in an Azure Machine Learning workspace. You can combine the capabilities of the Azure Machine Learning and Python extensions to manage a complete end-to-end machine learning workload in Azure Machine Learning from the Visual Studio Code environment.

**More Information**: For more information about using the Azure Machine Learning extension for Visual Studio Code, see the **documentation**[8].

## Compute Instances

Azure Machine Learning includes the ability to create *Compute Instances* in a workspace to provide a development environment that is managed with all of the other assets in the workspace.

8 https://docs.microsoft.com/azure/machine-learning/how-to-manage-resources-vscode



```
In [ ]:  from azureml.core import Workspace

         ws = Workspace.from_config()
         print(ws.name, "loaded")
```

**View Azure ML Resources**

Now that you have a connection to your workspace, you can view the resources it contains.

```
In [ ]:  from azureml.core import ComputeTarget, Datastore, Dataset

         print("Compute Targets:")
         for compute_name in ws.compute_targets:
             compute = ws.compute_targets[compute_name]
             print("\t", compute.name, ':', compute.type)

         print("Datastores:")
         for datastore_name in ws.datastores:
             datastore = Datastore.get(ws, datastore_name)
             print("\t", datastore.name, ':', datastore.datastore_type)

         print("Datasets:")
         for dataset_name in list(ws.datasets.keys()):
             dataset = Dataset.get_by_name(ws, dataset_name)
             print("\t", dataset.name)
```

Now you've seen how to use the Azure ML SDK to view the resources in your workspace. The SDK provides a great way to script the creation and configuration of the resources you need to operate machine learning workloads using Azure ML. For more details, see the Azure ML SDK documentation.

On the **File** menu, click **Close and Halt** to close this notebook. Then return to the lab instructions.

Compute Instances include Jupyter Notebook and JupyterLab installations that you can use to write and run code that uses the Azure Machine Learning SDK to work with assets in your workspace.

You can choose a VM image that provides the compute specification you need, from small CPU-only VMs to large GPU-enabled workstations. Because the VMs are hosted in Azure, you only pay for the compute resources when they are running; so you can create a VM to suit your needs, and stop it when your workload has completed to minimize costs.

You can store notebooks independently in workspace storage, and open them in any VM.

# Lab: Create an Azure Machine Learning Workspace

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will create and explore an Azure Machine Learning workspace.

## Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. Complete the **Create an Azure Machine Learning Workspace** exercise.

# Module Review

## Knowledge Check

In this module, you learned about the capabilities of Azure machine Learning and how to provision an Azure Machine Learning workspace and get started with Azure Machine Learning studio.

Use the following review questions to check your learning.

## Question 1

*Which of the following Azure resources are created alongside an Azure Machine Learning workspace?*

☐ Storage Account, Key Vault, and Application Insights

☐ Databricks workspace, Storage Account, and Key Vault

☐ Key Vault, Databricks workspace, and Application Insights

☐ Application Insights, Storage Account, and Databricks workspace

## Question 2

*Which of the following provides a web interface for managing assets in a workspace?*

☐ Azure Machine Learning studio

☐ Azure Cognitive Services

☐ Azure Synapse Analytics

## Question 3

*Which Visual Studio Code extension enables integrated management of workspace assets?*

☐ Python

☐ Azure Machine Learning

☐ Jupyter Notebooks

# Answers

**Question 1**

Which of the following Azure resources are created alongside an Azure Machine Learning workspace?

■ Storage Account, Key Vault, and Application Insights

□ Databricks workspace, Storage Account, and Key Vault

□ Key Vault, Databricks workspace, and Application Insights

□ Application Insights, Storage Account, and Databricks workspace

*Explanation*
*When you create an Azure Machine Learning workspace, storage account, key vault, and application insights resources are also created.*

**Question 2**

Which of the following provides a web interface for managing assets in a workspace?

■ Azure Machine Learning studio

□ Azure Cognitive Services

□ Azure Synapse Analytics

*Explanation*
*Azure Machine Learning studio provides a web-based portal for managing resources in a workspace.*

**Question 3**

Which Visual Studio Code extension enables integrated management of workspace assets?

□ Python

■ Azure Machine Learning

□ Jupyter Notebooks

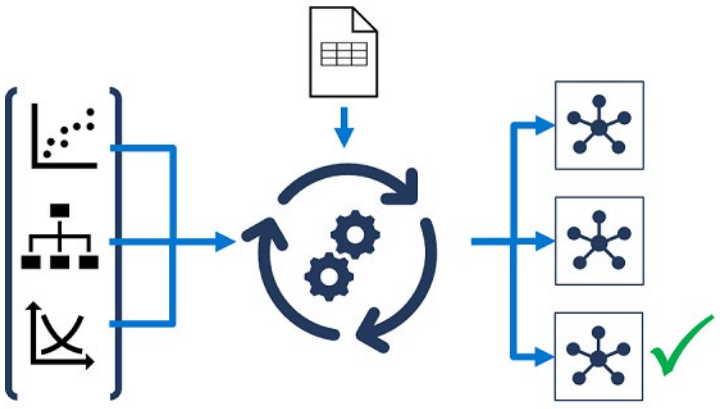
*Explanation*
*The Azure Machine Learning extension in Visual Studio Code provides an integrated interface for managing workspace assets.*

# Module 2 Visual Tools for Machine Learning

## Automated Machine Learning

### What is Automated Machine Learning?

*Automated Machine Learning* enables you to try multiple algorithms and preprocessing transformations with your data. This, combined with scalable cloud-based compute makes it possible to find the best performing model for your data without the huge amount of time-consuming manual trial and error that would otherwise be required.

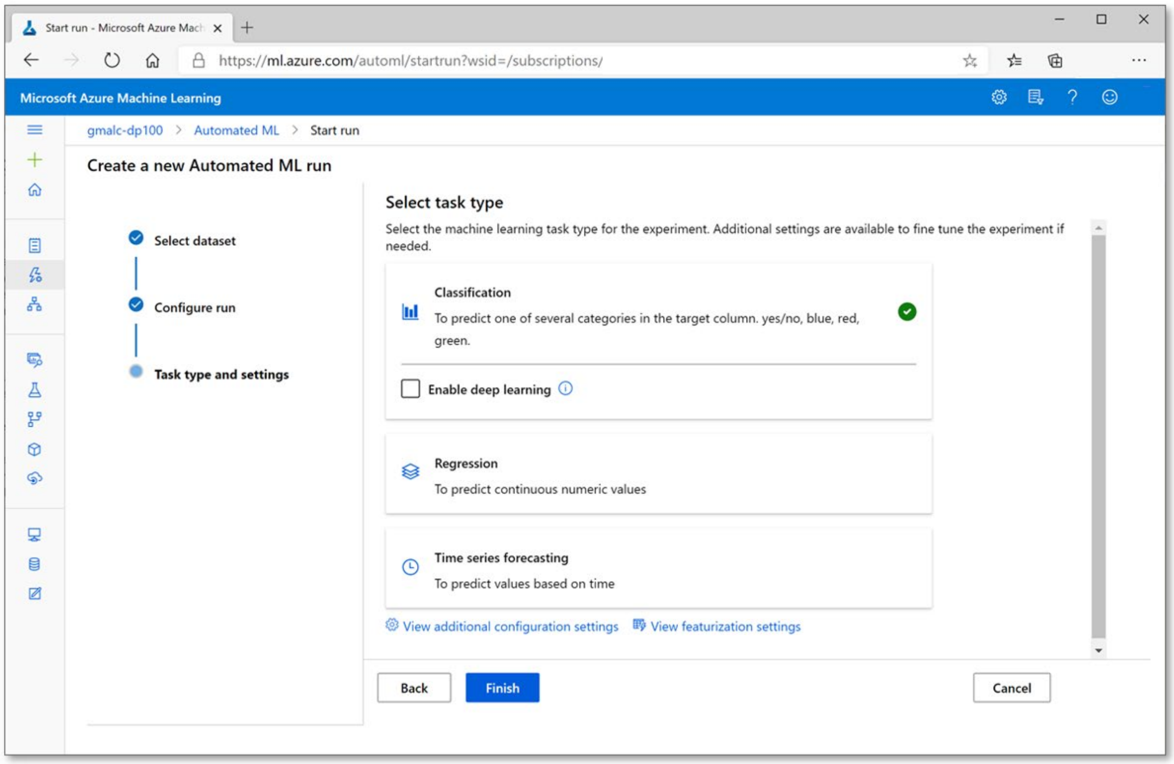


The automated machine learning capability in Azure Machine Learning supports supervised machine learning models; in other words, models for which the training data includes known label values. You can use automated machine learning to train models for:

- Classification (predicting categories or classes)
- Regression (predicting numeric values)
- Time series forecasting (regression with a time-series element, enabling you to predict numeric values at a future point in time)

The “best” performing model is identified based on the metric you specify.

## Automated ML in Azure Machine Learning studio


Start run - Microsoft Azure Mach
https://ml.azure.com/automl/startrun?wsid=/subscriptions/
Microsoft Azure Machine Learning
gmalc-dp100 > Automated ML > Start run
Create a new Automated ML run
Select dataset
Configure run
Task type and settings
Back
Finish
Cancel

Select task type

Select the machine learning task type for the experiment. Additional settings are available to fine tune the experiment if needed.

Classification
To predict one of several categories in the target column. yes/no, blue, red, green.

Enable deep learning

Regression
To predict continuous numeric values

Time series forecasting
To predict values based on time

View additional configuration settings
View featurization settings

There are three main steps to run an automated machine learning experiment using the user interface:

1. Specify a dataset containing the features and label data you want to use to train the model.
2. Configure the automated machine learning experiment run – including its name, the target label you want to train a model to predict, and the compute target on which to run the experiment.
3. Select the task type (classification, regression, or time-series), configuration settings, and featurization settings you want to apply in the experiment.

Note: You can also use the Azure Machine Learning SDK to run automated machine learning experiments – we'll explore this later in the course.

## Configuration and Featurization

You can use the following configuration settings to control the automated machine learning experiment:

- The primary metric for which you want to optimize the model – for example, **Accuracy**. Different task types support different metrics.
- Whether or not to generate feature importance explanations for the best model.
- Blocking algorithms that you do not want the experiment to try.
- Exit criterion to stop the experiment after a maximum amount of time or when a specific metric threshold is achieved.

- The validation technique used to split training and test data to evaluate model performance.
- The number of concurrent training runs.

Automated machine learning also supports data pre-processing, or featurization. It always tries various data normalization/scaling techniques and applies data guardrails to mitigate unbalanced data (for example, datasets where there are a significantly larger number of observations with one label value than another). You can optionally apply additional featurization techniques to individual columns, including dropping high-cardinality features (which tend not to be predictive), imputing missing values, encoding categorical features, and deriving features (for example, by splitting dates into day, month, and year features).

## Lab: Use Automated Machine Learning

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use automated machine learning to train a model.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

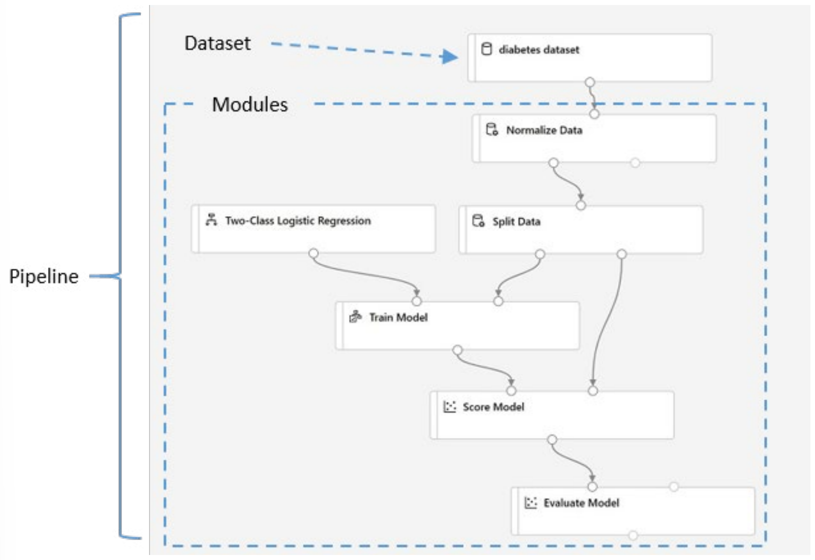
1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Use Automated Machine Learning** exercise.

# Azure Machine Learning Designer

## What is Azure Machine Learning Designer?

Azure Machine Learning Designer is a graphical environment for creating machine learning models, and publishing them as services that can be consumed by client applications.

In Azure Machine Learning Designer, you define a dataflow for training a machine learning model as a *pipeline*. This pipeline includes all of the steps that are required to ingest and process the data, before using it to train a model. Each step is independently executable, and can be run on any valid compute target. The pipeline will manage the flow of execution, starting compute targets for each step as required.



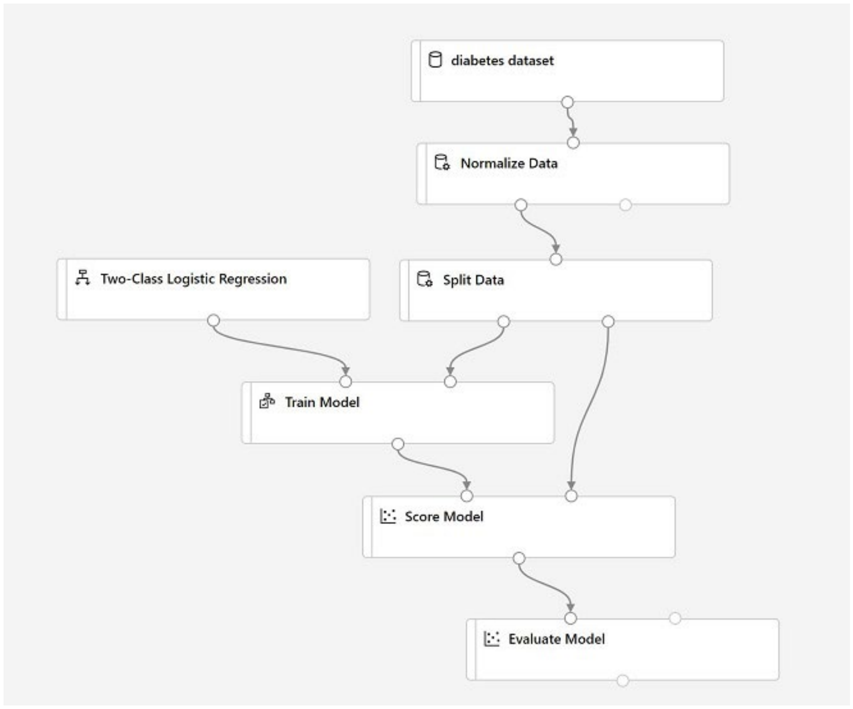
Most pipelines to train models begin with a *dataset* from which the training data is ingested, and then each step in the pipeline is encapsulated in a module with *inputs* and *outputs* through which the data flows.

The main benefit of using the designer is that it allows for "no-code" development of machine learning solutions through its drag-and-drop interface. The tool includes a wide range of pre-defined modules for data ingestion, feature selection and engineering, model training, and validation. Additionally, there are modules that enable you to add custom Python, R, and SQL logic to a data flow.

To run a designer pipeline, you must create a **Compute Cluster** in your workspace and specify it as the compute target for the pipeline.

**Note**: Don't worry too much about compute targets for the moment - you'll explore them in more detail later in the course.

## Training Pipelines



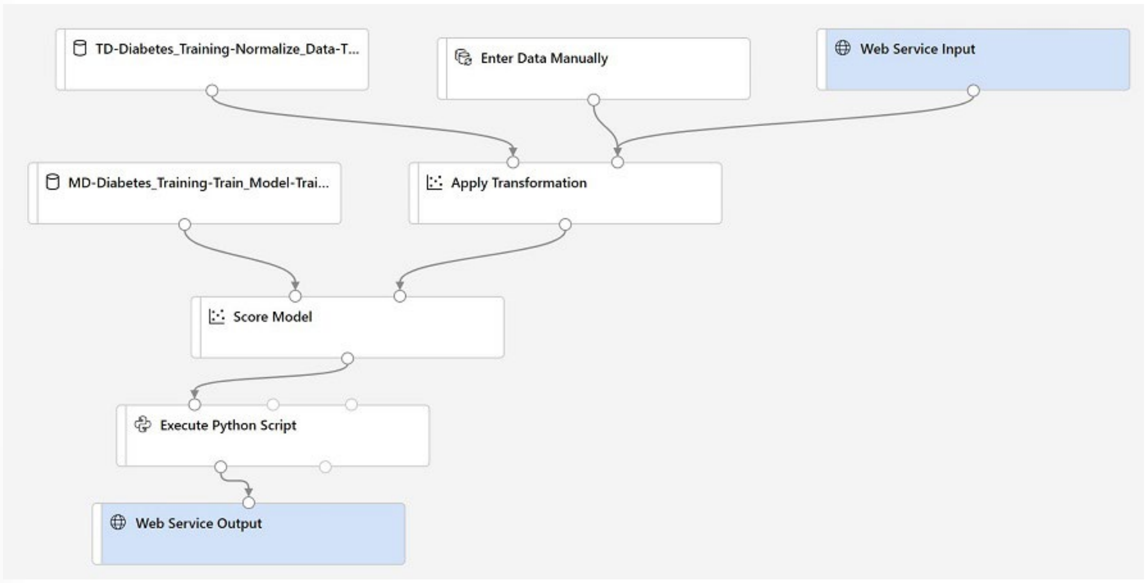
Azure Machine Learning Designer includes modules for training, scoring, and evaluating machine learning models. The specific details vary depending on the kind of model you are implementing, but the general approach is the same:

1. Use an algorithm module to specify the type of model to be trained. Azure Machine Learning Designer supports a range of algorithms for both supervised learning (classification and regression), and unsupervised learning (clustering).
2. Train the model by fitting the algorithm to the training data. For supervised learning algorithms, you must use the **Train Model** module and specify the label to be predicted from the features in the training data. For unsupervised clustering, you must use the **Train Clustering Model** module.
3. For a supervised learning algorithm, you typically split the data into a training set and a validation set, so after training the model with the training set you can use a **Score Model** module to predict labels for the validation set and evaluate the model. For an unsupervised clustering model, this step may be replaced by using the **Assign Data to Clusters** module to cluster a validation dataset.
4. You can evaluate model performance by using the **Evaluate Model** module to view metrics generated by scoring the test data. The specific metrics and associated visualizations vary depending on the type of model - for example, a binary classification model produces metrics for *accuracy*, *precision*, and *recall* as well as a *Receiver Operator Characteristic* (ROC) chart; while a regression model produces metrics such as *Root Mean Squared Error* (RMSE) and *Coefficient of Determination* (usually referred to as $R^2$).

The **Evaluate Model** module has two inputs, enabling you to train two models of the same type and evaluate them side-by-side to compare performance metrics.

## Inference Pipelines

Having trained a model using a *training pipeline*, you can use it to create an *inference pipeline* for either real-time or batch prediction.



An inference pipeline encapsulates the steps required to use the trained model in a web service that predicts labels for new data. It differs from the training pipeline in the following respects:

- A web service input defining an interface for new data to be scored is added to the beginning of real-time inference pipelines. By default, this is based on the schema of the training dataset.
- Steps that rely on statistics from the training data (such as feature normalization or categorical encoding) are encapsulated in transformation datasets that are applied to new data.
- The trained model is encapsulated in a dataset, removing the algorithm and model training modules.
- A Web service output containing the scored results is added at the end of real-time inferencing pipelines to define the output returned to applications consuming the service.

## Modifying the Inference Pipeline

Before deploying an inference pipeline as a web service, you may want to make some changes to it. For example:

- For supervised learning models, consider replacing the training dataset at the beginning of the pipeline with an alternative data definition that does not include the label column. This has the effect of removing the label column from the web service input schema, which is more intuitive for client application developers (who would otherwise need to submit a value for the label that they want the model to predict).
- If you choose to remove the label column from the input schema, ensure it is not explicitly referenced in any other modules in the pipeline, as this will cause a runtime exception.
- Remove any modules that are not required - for example, if the training pipeline includes an **Evaluate Model** module, it will be included by default in the inference pipeline, even though it is not used.
- Consider filtering the output columns. The **Score Model** module returns its input data as well as the scored label and probability columns, so by default the web service will return all of these to the client

application. The web service may be more efficient if you filter these to include only the required output, such as a row identifier and the scored label and probability.

- Consider adding *parameters*, which can be passed by calling applications to add flexibility to the pipeline. Typically, parameters are used to enable a choice of data sources to be used in the pipeline.

## Publishing a Service Endpoint

After creating and modifying an inference pipeline, you can publish an endpoint through which client applications can consume it as a web service.

### Deploying a Real-Time Inference Pipeline

For real-time inferencing, you must deploy the pipeline as a service on an Azure Kubernetes Services (AKS) compute target. The deployed pipeline service can then be accessed through an HTTP REST endpoint.

### Publishing a Batch Inference Pipeline

If you have created a batch inference pipeline, you can publish an HTTP endpoint through which the pipeline can be initiated. It will run on the Azure Machine Learning training compute target you have selected for the inference pipeline.

**Note**: It's important to note that batch inference pipelines are run on *training* compute, even when published as consumable services.

## Lab: Use Azure Machine Learning Designer

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use Azure Machine Learning designer to train and publish a model.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Use Azure Machine Learning designer** exercise.

## Module Review

### Knowledge Check

In this lesson, you learned how to publish an Azure Machine Learning Designer pipeline as an inference service.

Use the following review questions to check your learning.

## Question 1

*You want to use automated machine learning with car sales data to train a machine learning model that predicts the price of a car based on its make, model, engine size, and mileage. What task type should you select?*

☐ Classification

☐ Regression

☐ Time-series

## Question 2

*You are creating a training pipeline using a dataset that has multiple numeric columns. You want to transform the numeric columns so that the values are all on a similar scale. Which module should you add to the pipeline?*

☐ Select Columns in a Dataset

☐ Clean Missing Data

☐ Normalize Data

## Answers

**Question 1**

You want to use automated machine learning with car sales data to train a machine learning model that predicts the price of a car based on its make, model, engine size, and mileage. What task type should you select?

☐ Classification

■ Regression

☐ Time-series

*Explanation*
*Because the model must predict a numeric value with no time-series element, you should select the Regression task.*

**Question 2**

You are creating a training pipeline using a dataset that has multiple numeric columns. You want to transform the numeric columns so that the values are all on a similar scale. Which module should you add to the pipeline?

☐ Select Columns in a Dataset

☐ Clean Missing Data

■ Normalize Data

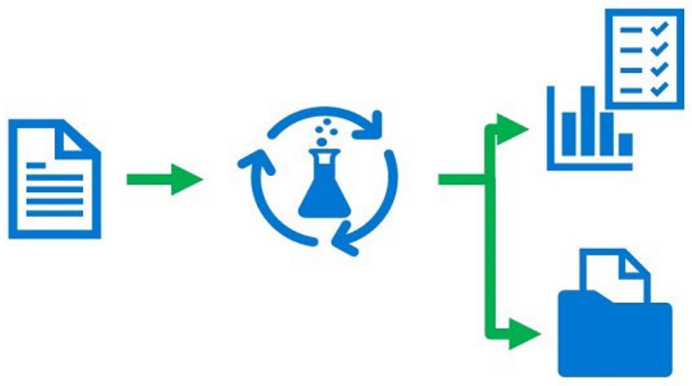
*Explanation*
*To standardize numeric features on the same scale, use the Normalize Data module.*

# Module 3 Running Experiments and Training Models

## Introduction to Experiments

### What is an Experiment?

Like any scientific discipline, data science involves running *experiments*; typically to explore data or to build and evaluate predictive models. In Azure Machine Learning, an experiment is a named process, usually the running of a script or a pipeline, that can generate metrics and outputs and be tracked in the Azure Machine Learning workspace. Additionally, metadata for the experiment run, and events that occur during the run are logged. The logged metadata includes the context for the experiment – for example, if the experiment is based on a script that is stored in a GitHub repo, metadata about the GitHub branch is included in the logged metadata.



An experiment can be run multiple times, with different data, code, or settings; and Azure Machine Learning tracks each run, enabling you to view run history and compare results for each run.

## The Experiment Run Context

When you submit an experiment, you use its *run context* to initialize and end the experiment run that is tracked in Azure Machine Learning, as shown in the following code sample:

```
from azureml.core import Experiment

# create an experiment variable
experiment = Experiment(workspace = ws, name = "my-experiment")

# start the experiment
run = experiment.start_logging()

# experiment code goes here

# end the experiment
run.complete()
```

After the experiment run has completed, you can view the details of the run in the **Experiments** tab in Azure Machine Learning studio.

## Running an Experiment Inline

You can use the Python SDK to run an Azure Machine Learning experiment inline, for example in a notebook. The key steps you need to perform in your code are:

1. Create or retrieve a named experiment in your workspace.
2. Start a run of the experiment – retrieving an active **Run** object.
3. Use the **Run** object to log metrics you want to review later.
4. Save or upload files to the run's **outputs** folder so that they are stored in the run history.
5. Complete the run

After completing the run, you can view its logs, metrics and outputs in Azure Machine Learning studio or use the SDK to retrieve them in code. There's also a widget that you can use to display run details in a notebook.

### Logging Metrics

Every experiment generates log files that include the messages that would be written to the terminal during interactive execution. This enables you to use simple `print` statements to write messages to the log. However, if you want to record named metrics for comparison across runs, you can do so by using the **Run** object; which provides a range of logging functions specifically for this purpose. These include:

- **log**: Record a single named value.
- **log_list**: Record a named list of values.
- **log_row**: Record a row with multiple columns.
- **log_table**: Record a dictionary as a table.
- **log_image**: Record an image file or a plot.

**More Information**: For more information about logging metrics during experiment runs, see **Monitor Azure ML experiment runs and metrics**[1] in the Azure Machine Learning documentation.

For example, following code records the number of observations (records) in a CSV file:

```
from azureml.core import Experiment
import pandas as pd

# Create an Azure ML experiment in your workspace
experiment = Experiment(workspace=ws, name='my-experiment')

# Start logging data from the experiment
run = experiment.start_logging()

# load the data and count the rows
data = pd.read_csv('data.csv')
row_count = (len(data))

# Log the row count
run.log('observations', row_count)

# Complete the experiment
run.complete()
```

## Retrieving and Viewing Logged Metrics

You can view the  metrics logged by an experiment run in Azure Machine Learning studio or by using the **RunDetails** widget in a notebook, as shown here:

```
from azureml.widgets import RunDetails

RunDetails(run).show()
```

You can also retrieve the metrics using the **Run** object's **get_metrics** method, which returns a JSON representation of the metrics, as shown here:

```
import json

# Get logged metrics
metrics = run.get_metrics()
print(json.dumps(metrics, indent=2))
```

The previous code produces output similar to this:

```
{
  "observations": 15000
}
```

1 https://aka.ms/AA70zf6

### Experiment Output Files

In addition to logging metrics, an experiment can generate output files. Often these are trained machine learning models, but you can save any sort of file and make it available as an output of your experiment run. The output files of an experiment are saved in its **outputs** folder.

The technique you use to add files to the outputs of an experiment depend on how your running the experiment. The examples shown so far control the experiment lifecycle inline in your code, and when taking this approach you can upload local files to the run's **outputs** folder by using the **Run** object's **upload_file** method in your experiment code as shown here:

```
run.upload_file(name='outputs/sample.csv', path_or_stream='./sample.csv')
```

When running an experiment in a remote compute context (which we'll discuss later in this course), any files written to the **outputs** folder in the compute context are automatically uploaded to the run's **outputs** folder when the run completes.

Whichever approach you use to run your experiment, you can retrieve a list of output files from the **Run** object like this:

```
import json

files = run.get_file_names()
print(json.dumps(files, indent=2))
```

The previous code produces output similar to this:

```
[
  "outputs/sample.csv"
]
```

## Running a Script as an Experiment

You can run an experiment inline using the **start_logging** method of the **Experiment** object, but it's more common to encapsulate the experiment logic in a script and run the script as an experiment. The script can be run in any valid compute context, making this a more flexible solution for running experiments at scale.

### Writing an Experiment Script

An experiment script is just a Python code file that contains the code you want to run in the experiment. To access the experiment run context (which is needed to log metrics) the script must import the **azureml.core.Run** class and call its **get_context** method. The script can then use the run context to log metrics, upload files, and complete the experiment, as shown here:

```
from azureml.core import Run
import pandas as pd
import matplotlib.pyplot as plt
import os

# Get the experiment run context
run = Run.get_context()
```

```
# load the diabetes dataset
data = pd.read_csv('data.csv')

# Count the rows and log the result
row_count = (len(data))
run.log('observations', row_count)

# Save a sample of the data
os.makedirs('outputs', exist_ok=True)
data.sample(100).to_csv("outputs/sample.csv", index=False, header=True)

# Complete the run
run.complete()
```

The experiment script is saved in a folder along with any other files on which it depends. For example, you could save this script as **experiment.py** in a folder named **experiment_folder**. Since the script includes code to load training data from **data.csv**, this file should also be saved in the folder.

### Running an Experiment Script

To run a script as an experiment, you must define a *script configuration* that defines the script to be run and the Python environment in which to run it. This is implemented by using a **ScriptRunConfig** object.

For example, the following code could be used to run an experiment based on a script in the **experiment_files** folder (which must also contain any files used by the script, such as the *data.csv* file in previous script code example):

```
from azureml.core import Experiment, ScriptRunConfig

# Create a script config
script_config = ScriptRunConfig(source_directory='experiment_folder',
                                script='experiment.py')

# submit the experiment
experiment = Experiment(workspace=ws, name='my-experiment')
run = experiment.submit(config=script_config)
run.wait_for_completion(show_output=True)
```

**Note**: An implicitly created **RunConfiguration** object defines the Python environment for the experiment, including the packages available to the script. The default environment includes standard Python libraries like **numpy** and **pandas** as well as the **azureml-defaults** package, which contains the classes needed to work with the experiment run context. If your script depends on packages that are not included in the default environment, you must associate the **ScriptRunConfig** with an **Environment** object that makes use of a **CondaDependencies** object to specify the Python packages required. Runtime environments are discussed in more detail later in this course.

## Using MLflow

MLflow is an Open Source library for managing machine learning experiments, and includes a tracking component for logging. If your organization already uses MLflow, you can continue to use it to track metrics in Azure Machine Learning.

## Using MLflow Inline

You can use MLflow to run an inline experiment, as shown in this example:

```
from azureml.core import Experiment
import pandas as pd
import mlflow

# Set the MLflow tracking URI to the workspace
mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())

# Create an Azure ML experiment in your workspace
experiment = Experiment(workspace=ws, name='my-experiment')
mlflow.set_experiment(experiment.name)

# start the MLflow experiment
with mlflow.start_run():

    print("Starting experiment:", experiment.name)

    # load the data and count the rows
    data = pd.read_csv('data.csv')
    row_count = (len(data))

    # Log the row count
    mlflow.log_metric('observations', row_count)
```

Note that the code explicitly sets the MLflow tracking URI to the tracking endpoint provided by the Azure Machine Learning workspace.

## Using MLflow in a Script

To use MLflow in a script-based experiment, th script must include code to start an MLflow run and log metrics; and the **ScriptRunConfig** for the experiment must include an environment in which the **mlflow** and **azureml-mlflow** packages are installed.

## Script

```
import pandas as pd
import mlflow

# start the MLflow experiment
with mlflow.start_run():

    print("Starting experiment:", experiment.name)

    # load the data and count the rows
    data = pd.read_csv('data.csv')
    row_count = (len(data))

    # Log the row count
    mlflow.log_metric('observations', row_count)
```

### Code to initiate the experiment run

```
from azureml.core import Experiment, ScriptRunConfig, Environment
from azureml.core.conda_dependencies import CondaDependencies

# Create a Python environment for the experiment
mlflow_env = Environment("mlflow-env")

# Ensure the required packages are installed
packages = CondaDependencies.create(conda_packages=['pandas','pip'],
                                    pip_packages=['mlflow','azureml-mlflow'])
mlflow_env.python.conda_dependencies = packages

# Create a script config
script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=mlflow_env)

# submit the experiment
experiment = Experiment(workspace=ws, name='mlflow-script')
run = experiment.submit(config=script_config)
```

**More Information**: For more information about using MLflow with Azure Machine Learning, see **Track metrics and deploy models with MLflow and Azure Machine Learning**[2] in the documentation.

## Lab: Run Experiments

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the Azure Machine Learning SDK to run experiments and log metrics.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Run experiments** exercise.

2 https://aka.ms/AA70zf7

# Training and Registering Models

## Training a Model in a Script

You can use a **ScriptRunConfig** to run a script-based experiment that trains a machine learning model.

### Writing a Script to Train a Model

When using an experiment to train a model, your script should save the trained model in the **outputs** folder. For example, the following script trains a model using Scikit-Learn, and saves it in the **outputs** folder using the **joblib** package:

```
from azureml.core import Run
import pandas as pd
import numpy as np
import joblib
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Get the experiment run context
run = Run.get_context()

# Prepare the dataset
diabetes = pd.read_csv('data.csv')
X, y = data[['Feature1','Feature2','Feature3']].values, data['Label'].
values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

# Train a logistic regression model
reg = 0.1
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_
train)

# calculate accuracy
y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
run.log('Accuracy', np.float(acc))

# Save the trained model
os.makedirs('outputs', exist_ok=True)
joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()
```

To prepare for an experiment that trains a model, a script like this is created and saved in a folder. For example, you could save this script as **training_script.py** in a folder named **training_folder**. Since the script includes code to load training data from **data.csv**, this file should also be saved in the folder.

### Running the Script as an Experiment

To run the script, create a **ScriptRunConfig** that references the folder and script file. You generally also need to define a Python (Conda) environment that includes any packages required by the script. In this example, the script uses Scikit-Learn so you must create an environment that includes that. The script also uses Azure Machine Learning to log metrics, so you need to remember to include the **azureml-defaults** package in the environment.

```
from azureml.core import Experiment, ScriptRunConfig, Environment
from azureml.core.conda_dependencies import CondaDependencies

# Create a Python environment for the experiment
sklearn_env = Environment("sklearn-env")

# Ensure the required packages are installed
packages = CondaDependencies.create(conda_packages=['scikit-learn','pip'],
                                    pip_packages=['azureml-defaults'])
sklearn_env.python.conda_dependencies = packages

# Create a script config
script_config = ScriptRunConfig(source_directory='training_folder',
                                script='training.py',
                                environment=sklearn_env)

# Submit the experiment
experiment = Experiment(workspace=ws, name='training-experiment')
run = experiment.submit(config=script_config)
run.wait_for_completion()
```

## Using Script Arguments

You can increase the flexibility of script-based experiments by using arguments to set variables in the script.

### Working with Script Arguments

To use parameters in a script, you must use a library such as **argparse** to read the arguments passed to the script and assign them to variables. For example, the following script reads an argument named **–reg-rate**, which is used to set the regularization rate hyperparameter for the logistic regression algorithm used to train a model.

```
from azureml.core import Run
import argparse
import pandas as pd
import numpy as np
import joblib
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Get the experiment run context
run = Run.get_context()
```

```
# Set regularization hyperparameter
parser = argparse.ArgumentParser()
parser.add_argument('--reg-rate', type=float, dest='reg_rate', default=0.01)
args = parser.parse_args()
reg = args.reg_rate

# Prepare the dataset
diabetes = pd.read_csv('data.csv')
X, y = data[['Feature1','Feature2','Feature3']].values, data['Label'].
values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

# Train a logistic regression model
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_
train)

# calculate accuracy
y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
run.log('Accuracy', np.float(acc))

# Save the trained model
os.makedirs('outputs', exist_ok=True)
joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()
```

## Passing Arguments to an Experiment Script

To pass parameter values to a script being run in an experiment, you need to provide an **arguments** value containing a list of comma-separated arguments and their values to the **ScriptRunConfig**, like this:

```
# Create a script config
script_config = ScriptRunConfig(source_directory='training_folder',
                                script='training.py',
                                arguments = ['--reg-rate', 0.1],
                                environment=sklearn_env)
```

# Registering a Trained Model

After running an experiment that trains a model you can use a reference to the **Run** object to retrieve its outputs, including the trained model.

## Retrieving Model Files

After an experiment run has completed, you can use the run objects **get_file_names** method to list the files generated. Standard practice is for  scripts that train models to save them in the run's **outputs** folder.

You can also use the run object's **download_file** and **download_files** methods to download output files to the local file system.

```
# "run" is a reference to a completed experiment run

# List the files generated by the experiment
for file in run.get_file_names():
    print(file)

# Download a named file
run.download_file(name='outputs/model.pkl', output_file_path='model.pkl')
```

## Registering a Model

Model registration enables you to track multiple versions of a model, and retrieve models for *inferencing* (predicting label values from new data). When you register a model, you can specify a name, description, tags, framework (such as Scikit-Learn or PyTorch), framework version, custom properties, and other useful metadata. Registering a model with the same name as an existing model automatically creates a new version of the model, starting with 1 and increasing in units of 1.

To register a model from a local file, you can use the **register** method of the **Model** object as shown here:

```
from azureml.core import Model

model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model',
                       tags={'data-format': 'CSV'},
                       model_framework=Model.Framework.SCIKITLEARN,
                       model_framework_version='0.20.3')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model',
                    tags={'data-format': 'CSV'},
                    model_framework=Model.Framework.SCIKITLEARN,
                    model_framework_version='0.20.3')
```

## Viewing Registered Models

You can view registered models in Azure Machine Learning studio. You can also use the **Model** object to retrieve details of registered models like this:

```
from azureml.core import Model

for model in Model.list(ws):
```

```
# Get model name and auto-generated version
print(model.name, 'version:', model.version)
```

## Lab: Train Models

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the Azure Machine Learning SDK to train and register a model.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Train models** exercise.

## Module Review

In this module, you learned how to run an experiment in Azure Machine Learning, and how to use an experiment to train a machine learning model.

Use the following review questions to check your learning.

### Question 1

*You are using the Azure Machine Learning Python SDK to write code for an experiment.*
*You need to record metrics from each run of the experiment, and be able to retrieve them easily from each run.*
*What should you do?*

☐ Add print statements to the experiment code to print the metrics.

☐ Use the log methods of the Run class to record named metrics.

☐ Save the experiment data in the outputs folder.

### Question 2

*You want to use a script-based experiment to train a PyTorch model, setting the batch size and learning rate hyperparameters to different values each time the experiment runs.*
*What should you do?*

☐ Create multiple script files – one for each batch size and learning rate combination you want to use.

☐ Set the batch_size and learning_rate properties of the ScriptRunConfig before running the experiment.

☐ Add arguments for batch size and learning rate to the script, and set them in the arguments property of the ScriptRunConfig

# Answers

**Question 1**

You are using the Azure Machine Learning Python SDK to write code for an experiment.
You need to record metrics from each run of the experiment, and be able to retrieve them easily from each run.
What should you do?

- ☐ Add print statements to the experiment code to print the metrics.
- ■ Use the log methods of the Run class to record named metrics.
- ☐ Save the experiment data in the outputs folder.

*Explanation*
*To record metrics in an experiment run, use the Run.log methods.*

**Question 2**

You want to use a script-based experiment to train a PyTorch model, setting the batch size and learning rate hyperparameters to different values each time the experiment runs.
What should you do?

- ☐ Create multiple script files – one for each batch size and learning rate combination you want to use.
- ☐ Set the batch_size and learning_rate properties of the ScriptRunConfig before running the experiment.
- ■ Add arguments for batch size and learning rate to the script, and set them in the arguments property of the ScriptRunConfig

*Explanation*
*To use variable values in a script, pass them as arguments.*

# Module 4 Working with Data

## Working with Datastores

### What are Datastores?

In Azure Machine Learning, *datastores* are abstractions for cloud data sources. They encapsulate the information required to connect to data sources. You can access datastores directly in code by using the Azure Machine Learning SDK, and use it to upload or download data.

### Types of Datastore

Azure Machine Learning supports the creation of datastores for multiple kinds of Azure data source, including:

- Azure Storage (blob and file containers)
- Azure Data Lake stores
- Azure SQL Database
- Azure Databricks file system (DBFS)

**Note**: For a full list of supported datastores, see the **Azure Machine Learning documentation**[1].

### Built-in Datastores

Every workspace has two built-in datastores (an Azure Storage blob container, and an Azure Storage file container) that are used as system storage by Azure Machine Learning. There's also a third datastore that gets added to your workspace if you make use of the open datasets provided as samples (for example, by creating a designer pipeline based on a sample dataset)

In most machine learning projects, you will likely need to work with data sources of your own - either because you need to store larger volumes of data than the built-in datastores support, or because you need to integrate your machine learning solution with data from existing applications.

1 https://aka.ms/AA70zfl

## Working with Datastores

To add a datastore to your workspace, you can register it using the graphical interface in Azure Machine Learning studio, or you can use the Azure Machine Learning SDK. For example, the following code registers an Azure Storage blob container as a datastore named **blob_data**.

```
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()

# Register a new datastore
blob_ds = Datastore.register_azure_blob_container(workspace=ws,
                                                  datastore_name='blob_
data',
                                                  container_name='data_con-
tainer',
                                                  account_name='az_store_
acct',
                                                  account_key='123456ab-
cde789…')
```

### Managing Datastores

You can view and manage datastores in Azure Machine Learning Studio, or you can use the Azure Machine Learning SDK. For example, the following code lists the names of each datastore in the workspace.

```
for ds_name in ws.datastores:
    print(ds_name)
```

You can get a reference to any datastore by using the **Datastore.get()** method as shown here:

```
blob_store = Datastore.get(ws, datastore_name='blob_data')
```

The workspace always includes a *default* datastore (initially, this is the built-in **workspaceblobstore** datastore), which you can retrieve by using the **get_default_datastore()** method of a **Workspace** object, like this:

```
default_store = ws.get_default_datastore()
```

## Considerations for Datastores

When planning for datastores, consider the following guidelines:

- When using Azure blob storage, *premium* level storage may provide improved I/O performance for large datasets. However, this option will increase cost and may limit replication options for data redundancy.
- When working with data files, although CSV format is very common, Parquet format generally results in better performance.

- You can access any datastore by name, but you may want to consider changing the default datastore (which is initially the built-in **workspaceblobstore** datastore).

To change the default datastore, use the **set_default_datastore()** method:

```
ws.set_default_datastore('blob_data')
```

# Working with Datasets

## What are Datasets?

*Datasets* are versioned packaged data objects that can be easily consumed in experiments and pipelines. Datasets are the recommended way to work with data, and are the primary mechanism for advanced Azure Machine Learning capabilities like data labeling and data drift monitoring.

### Types of Dataset

Datasets are typically based on files in a datastore, though they can also be based on URLs and other sources. You can create the following types of dataset:

- **Tabular**: The data is read from the dataset as a table. You should use this type of dataset when your data is consistently structured and you want to work with it in common tabular data structures, such as Pandas dataframes.
- **File**: The dataset presents a list of file paths that can be read as though from the file system. Use this type of dataset when your data is unstructured, or when you need to process the data at the file level (for example, to train a convolutional neural network from a set of image files).

You can create datasets from individual files or multiple file paths. The paths can include wildcards (for example, */files/*.csv*) making it possible to encapsulate data from a large number of files in a single dataset.

## Creating and Registering Datasets

You can create a dataset and work with it immediately, and you can then *register* the dataset in the workspace to make it available for use in experiments and data processing pipelines later.

You can create datasets by using the visual interface in Azure Machine Learning studio, or you can use the Azure Machine Learning SDK.

### Creating and Registering Tabular Datasets

To create a tabular dataset using the SDK, use the **from_delimited_files** method of the **Dataset.Tabular** class, like this:

```
from azureml.core import Dataset

blob_ds = ws.get_default_datastore()
csv_paths = [(blob_ds, 'data/files/current_data.csv'),
             (blob_ds, 'data/files/archive/*.csv')]
tab_ds = Dataset.Tabular.from_delimited_files(path=csv_paths)
tab_ds = tab_ds.register(workspace=ws, name='csv_table')
```

The dataset in this example includes data from two file paths within the default datastore:

- The **current_data.csv** file in the **data/files** folder.
- All .csv files in the **data/files/archive/** folder.

After creating the dataset, the code registers it in the workspace with the name **csv_table**.

## Creating and Registering File Datasets

To create a file dataset using the SDK, use the **from_files** method of the **Dataset.File** class, like this:

```
from azureml.core import Dataset

blob_ds = ws.get_default_datastore()
file_ds = Dataset.File.from_files(path=(blob_ds, 'data/files/images/*.jpg'))
file_ds = file_ds.register(workspace=ws, name='img_files')
```

The dataset in this example includes all .jpg files in the **data/files/images** path within the default datastore:

After creating the dataset, the code registers it in the workspace with the name **img_files**.

## Retrieving a Registered Dataset

After registering a dataset, you can retrieve it by using any of the following techniques:

- The **datasets** dictionary attribute of a **Workspace** object.
- The **get_by_name** or **get_by_id** method of the **Dataset** class.

Both of these techniques are shown in the following code:

```
import azureml.core
from azureml.core import Workspace, Dataset

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Get a dataset from the workspace datasets collection
ds1 = ws.datasets['csv_table']

# Get a dataset by name from the datasets class
ds2 = Datasets.get_by_name(ws, 'img_files')
```

# Working with Tabular Datasets

You can read data directly from a tabular dataset by converting it into a Pandas or Spark dataframe:

```
df = tab_ds.to_pandas_dataframe()
# code to work with dataframe goes here, for example:
print(df.head())
```

## Passing a Tabular Dataset to an Experiment Script

When you need to access a dataset in an experiment script, you must pass the dataset to the script. There are two ways you can do this.

## Use a Script Argument

You can pass a tabular dataset as a script argument. When you take this approach, the argument received by the script is the unique ID for the dataset in your workspace. In the script, you can then get the workspace from the run context and use it to retrieve the dataset by it's ID.

## ScriptRunConfig

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pan-
das]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', tab_ds],
                                environment=env)
```

## Script

```
from azureml.core import Run, Dataset

parser.add_argument('--ds', type=str, dest='dataset_id')
args = parser.parse_args()

run = Run.get_context()
ws = run.experiment.workspace
dataset = Dataset.get_by_id(ws, id=args.dataset_id)
data = dataset.to_pandas_dataframe()
```

## Use a Named Input

Alternatively, you can pass a tabular dataset as a *named input*. In this approach, you use the **as_named_input** method of the dataset to specify a name for the dataset. Then in the script, you can retrieve the dataset by name from the run context's **input_datasets** collection without needing to retrieve it from the workspace. Note that if you use this approach, you still need to include a script argument for the dataset, even though you don’t actually use it to retrieve the dataset.

## ScriptRunConfig

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pan-
das]'])
env.python.conda_dependencies = packages
```

```
script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', tab_ds.as_named_in-
put('my_dataset')],
                                environment=env)
```

### Script

```
from azureml.core import Run

parser.add_argument('--ds', type=str, dest='ds_id')
args = parser.parse_args()

run = Run.get_context()
dataset = run.input_datasets['my_dataset']
data = dataset.to_pandas_dataframe()
```

## Working with File Datasets

When working with a file dataset, you can use the **to_path()** method to return a list of the file paths encapsulated by the dataset:

```
for file_path in file_ds.to_path():
    print(file_path)
```

### Passing a File Dataset to an Experiment Script

Just as with a Tabular dataset, there are two ways you can pass a file dataset to a script. However, there are some key differences in the way that the dataset is passed.

### Use a Script Argument

You can pass a file dataset as a script argument. Unlike with a tabular dataset, you must specify a mode for the file dataset argument, which can be **as_download** or **as_mount**. This provides an access point that the script can use to read the files in the dataset. In most cases, you should use **as_download**, which copies the files to a temporary location on the compute where the script is being run. However, if you are working with a large amount of data for which there may not be enough storage space on the experiment compute, use **as_mount** to stream the files directly from their source.

### ScriptRunConfig

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pan-
das]'])
env.python.conda_dependencies = packages
```

```
script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', file_ds.as_download()],
                                environment=env)
```

## Script

```
from azureml.core import Run
import glob

parser.add_argument('--ds', type=str, dest='ds_ref')
args = parser.parse_args()
run = Run.get_context()

imgs = glob.glob(ds_ref + "/*.jpg")
```

## Use a Named Input

You can also pass a file dataset as a *named input*. In this approach, you use the **as_named_input** method of the dataset to specify a name before specifying the access mode. Then in the script, you can retrieve the dataset by name from the run context's **input_datasets** collection and read the files from there. As with tabular datasets, if you use a named input, you still need to include a script argument for the dataset, even though you don't actually use it to retrieve the dataset.

## ScriptRunConfig

```
env = Environment('my_env')
packages = CondaDependencies.create(conda_packages=['pip'],
                                    pip_packages=['azureml-defaults',
                                                  'azureml-dataprep[pan-
das]'])
env.python.conda_dependencies = packages

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                arguments=['--ds', file_ds.as_named_in-
put('my_ds').as_download()],
                                environment=env)
```

## Script

```
from azureml.core import Run
import glob

parser.add_argument('--ds', type=str, dest='ds_ref')
args = parser.parse_args()
run = Run.get_context()
```

```
dataset = run.input_datasets['my_ds']
imgs= glob.glob(dataset + "/*.jpg")
```

## Dataset Versioning

Datasets can be *versioned*, enabling you to track historical versions of datasets that were used in experiments, and reproduce those experiments with data in the same state.

### Creating a New Version of a Dataset

You can create a new version of a dataset by registering it with the same name as a previously registered dataset and specifying the **create_new_version** property:

```
img_paths = [(blob_ds, 'data/files/images/*.jpg'),
             (blob_ds, 'data/files/images/*.png')]
file_ds = Dataset.File.from_files(path=img_paths)
file_ds = file_ds.register(workspace=ws, name='img_files', create_new_ver-
sion=True)
```

In this example, the .png files in the **images** folder have been added to the definition of the **img_paths** dataset example used in the previous topic.

### Retrieving a Specific Dataset version

You can retrieve a specific version of a dataset by specifying the **version** parameter in the **get_by_name** method of the **Dataset** class.

```
img_ds = Dataset.get_by_name(workspace=ws, name='img_files', version=2)
```

## Lab: Work with Data

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will work with datastores and datasets.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Work with data** exercise.

## Module Review

In this module, you learned how to work with *datastores* and *datasets*.

Use the following review questions to check your learning.

## Question 1

*You have a reference to a Workspace named ws.*
*Which code retrieves the default datastore for the workspace?*

☐ default_ds = Datastore.get(ws, 'default')

☐ default_ds = ws.Datastores[0]

☐ default_ds = ws.get_default_datastore()

## Question 2

*A datastore contains a CSV file of structured data that you want to use as a Pandas dataframe.*
*Which kind of dataset should you create to make it easy to do this?*

☐ A file dataset

☐ A tabular dataset

## Question 3

*You want a script to stream data directly from a file dataset.*
*Which mode should you use?*

☐ as_mount()

☐ as_download()

☐ as_upload()

## Answers

**Question 1**

You have a reference to a Workspace named ws.
Which code retrieves the default datastore for the workspace?

☐ default_ds = Datastore.get(ws, 'default')

☐ default_ds = ws.Datastores[0]

■ default_ds = ws.get_default_datastore()

*Explanation*
*To get the default datastore, use the Workspace.get_default_datasetore method.*

**Question 2**

A datastore contains a CSV file of structured data that you want to use as a Pandas dataframe.
Which kind of dataset should you create to make it easy to do this?

☐ A file dataset

■ A tabular dataset

*Explanation*
*Use a tabular dataset for structured data that you want to work with in a Pandas dataframe.*

**Question 3**

You want a script to stream data directly from a file dataset.
Which mode should you use?

■ as_mount()

☐ as_download()

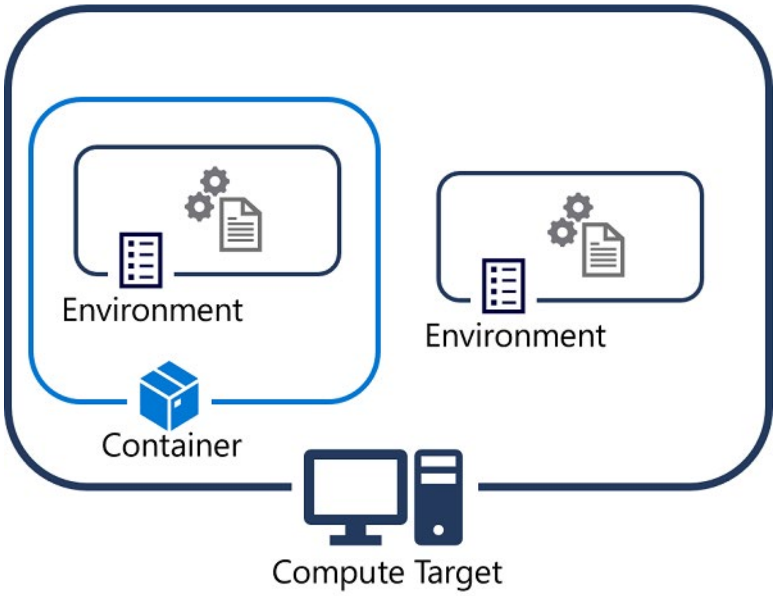
☐ as_upload()

*Explanation*
*Use as_mount() mode to stream data directly from its source.*

# Module 5 Working with Compute

## Environments

### Run Contexts for Experiments



Python code runs in the context of a *virtual environment* that defines the version of the Python runtime to be used as well as the installed packages available to the code. In most Python installations, packages are installed and managed in environments using **Conda** or **pip**.

To improve portability, we usually create environments in docker containers that are in turn be hosted in compute targets, such as your development computer, virtual machines, or clusters in the cloud.

### Explicitly Creating Environments

There are multiple ways to create environments in Azure Machine Learning.

## Creating an Environment from a Specification File

You can use a Conda or pip specification file to define the packages required in a Python evironment, and use it to create an **Environment** object.

For example, you could save the following Conda configuration settings in a file named **conda.yml**:

```
name: py_env
dependencies:
  - numpy
  - pandas
  - scikit-learn
  - pip:
    - azureml-defaults
```

Then, you could use the following code creates an Azure Machine Learning environment from the saved specification file:

```
from azureml.core import Environment

env = Environment.from_conda_specification(name='training_environment',
                                           file_path='./conda.yml')
```

## Creating an Environment from an Existing Conda Environment

If you have an existing Conda environment defined on your workstation, you can use it to define an Azure Machine Learning environment:

```
from azureml.core import Environment

env = Environment.from_existing_conda_environment(name='training_environ-
ment',
                                                  conda_environment_
name='py_env')
```

## Creating an Environment by Specifying Packages

You can define an environment by specifying the Conda and pip packages you need in a **CondaDependencies** object, like this:

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

env = Environment('training_environment')
deps = CondaDependencies.create(conda_packages=['scikit-learn','pan-
das','numpy', 'pip'],
                                pip_packages=['azureml-defaults'])
env.python.conda_dependencies = deps
```

**Tip**: It's best practice to use conda to install pip if you plan to also install pip packages.

## Configuring Environment Containers

Usually, environments for experiment script are created in containers. The following code configures a script-based experiment to host the **env** environment created previously in a container (this is the default unless you use a **DockerConfiguration** with a **use_docker** attribute of **False**, in which case the environment is created directly in the compute target)

```
from azureml.core import Experiment, ScriptRunConfig
from azureml.core.runconfig import DockerConfiguration

docker_config = DockerConfiguration(use_docker=True)

script_config = ScriptRunConfig(source_directory='my_folder',
                                script='my_script.py',
                                environment=env,
                                docker_runtime_config=docker_config)
```

Azure Machine Learning uses a library of base images for containers, choosing the appropriate base for the compute target you specify (for example, including Cuda support for GPU-based compute). If you have created custom container images and registered them in a container registry, you can override the default base images and use your own by modifying the attributes of the environment's **docker** property..

```
env.docker.base_image='my-base-image'
env.docker.base_image_registry='myregistry.azurecr.io/myimage'
```

Alternatively, you can have an image created on-demand based on the base image and additional settings in a dockerfile.

```
env.docker.base_image = None
env.docker.base_dockerfile = './Dockerfile'
```

By default, Azure machine Learning handles Python paths and package dependencies. If your image already includes an installation of Python with the dependencies you need, you can override this behavior by setting **python.user_managed_dependencies** to **True** and setting an explicit Python path for your installation.

```
env.python.user_managed_dependencies=True
env.python.interpreter_path = '/opt/miniconda/bin/python'
```

## Registering and Reusing Environments

After you've created an environment, you can register it in your workspace and reuse it for future experiments that have the same Python dependencies.

### Registering an Environment

Use the **register** method of an **Environment** object to register an environment:

```
env.register(workspace=ws)
```

You can view the registered environments in your workspace like this:

```
from azureml.core import Environment

env_names = Environment.list(workspace=ws)
for env_name in env_names:
    print('Name:',env_name)
```

## Retrieving and using an Environment

You can retrieve a registered environment by using the **get** method of the **Environment** class, and then assign it to a **ScriptRunConfig**.

For example, the following code sample retrieves the *training_environment* registered environment, and assigns it to an estimator:

```
from azureml.core import ScriptRunConfig, Environment

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=training_env)
```

When an experiment based on the estimator is run, Azure Machine Learning will look for an existing environment that matches the definition, and if none is found a new environment will be created based on the registered environment specification.

## Curated Environments

Azure Machine Learning includes a selection of pre-defined *curated* environments that reflect common usage scenarios. These include environments that are pre-configured with package dependencies for common frameworks, such as Scikit-Learn, PyTorch, Tensorflow, and others.

Curated environments are registered in all Azure Machine Learning workspaces with a name that begins ***AzureML-***.

**Note**: You can't register your own environments with an “AzureML-” prefix.

To view curated environments and the dependencies they contain, you can run the following code:

```
from azureml.core import Environment

envs = Environment.list(workspace=ws)
for env in envs:
    if env.startswith("AzureML"):
        print("Name",env)
        print("packages", envs[env].python.conda_dependencies.serialize_to_
string())
```

## Compute Targets

## Compute Options for Experiment Runs

Azure Machine Learning supports multiple types of compute for experimentation and training. This enables you to select the most appropriate type of compute target for your particular needs.

- **Local compute** - You can specify a *local* compute target for most processing tasks in Azure Machine Learning. This runs the experiment on the same compute target as the code used to initiate the experiment, which may be your physical workstation or a virtual machine such as an Azure Machine Learning *compute instance* on which you are running a notebook. Local compute is generally a great choice during development and testing with low to moderate volumes of data.
- **Compute clusters** - For experiment workloads with high scalability requirements, you can use Azure Machine Learning compute clusters; which are multi-node clusters of Virtual Machines that automatically scale up or down to meet demand. This is a cost-effective way to run experiments that need to handle large volumes of data or use parallel processing to distribute the workload and reduce the time it takes to run.
- **Attached compute** - If you already use an Azure-based compute environment for data science, such as a virtual machine or an Azure Databricks cluster, you can attach it to your Azure Machine Learning workspace and use it as a compute target for certain types of workload.

**Note**: In Azure Machine Learning studio, you may have spotted that there's another type of compute named *inference clusters*. This kind of compute represents an Azure Kubernetes Service cluster and can only be used to deploy trained models as inferencing services. We'll explore deployment later, but for now we'll focus on compute for experiments and model training.

The ability to assign experiment runs to specific compute targets helps you implement a flexible data science ecosystem in the following ways:

- Code can be developed and tested on local or low-cost compute, and then moved to more scalable compute for production workloads.
- You can run individual processes on the compute target that best fits its needs. For example, by using GPU-based compute to train deep learning models, and switching to lower-cost CPU-only compute to test and register the trained model.

One of the core benefits of cloud computing is the ability to manage costs by paying only for what you use. In Azure Machine Learning, you can take advantage of this principle by defining compute targets that:

- Start on-demand and stop automatically when no longer required.
- Scale automatically based on workload processing needs.

## Creating a Compute Cluster

The most common ways to create a compute cluster are to use the **Compute** page in Azure Machine Learning studio, or to use the Azure Machine Learning SDK. Additionally, you can create compute targets using the Azure Machine Learning extension in Visual Studio Code, or by using the Azure command line interface (CLI) extension for Azure Machine Learning.

## Creating a Compute Cluster with the SDK

A *managed* compute target is one that is managed by Azure Machine Learning, such as an Azure Machine Learning training cluster.

To create an Azure Machine Learning compute cluster compute target, use the **azureml.core.compute.ComputeTarget** class and the **AmlCompute** class, like this:

```
from azureml.core import Workspace
from azureml.core.compute import ComputeTarget, AmlCompute

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'aml-cluster'

# Define compute configuration
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_
DS11_V2',
                                                       min_nodes=0, max_
nodes=4,
                                                       vm_priority='lowpri-
ority')

# Create the compute
aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)
aml_cluster.wait_for_completion(show_output=True)
```

In this example, a cluster with up to four nodes that is based on the STANDARD_DS11_v2 virtual machine image will be created. The priority for the virtual machines (VMs) can be set to dedicated, meaning they are reserved for use in this cluster, or to lowpriority, which has a lower cost but means that the VMs can be preempted if a higher priority workload requires the compute. Additionally, there are options to enable SSH access, connect the cluster to a virtual network, and manage the identity the cluster should use for access to other Azure resources.

**Note**: For a full list of **AmlCompute** configuration options, see the **AmlCompute class**[1] SDK documentation.

## Checking for an Existing Compute Target

In many cases, you will want to check for the existence of a compute target, and only create a new one if there isn't already one with the specified name. To accomplish this, you can catch the **ComputeTargetException** exception, like this:

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

compute_name = 'aml-cluster'

# Check if the compute target exists
```

1 https://aka.ms/AA70zfq

```
try:
    aml_cluster = ComputeTarget(workspace=ws, name=compute_name)
    print('Found existing cluster.')
except ComputeTargetException:
    # If not, create it
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_
DS11_V2',
                                                           max_nodes=4)
    aml_cluster = ComputeTarget.create(ws, compute_name, compute_config)

aml_cluster.wait_for_completion(show_output=True)
```

**More Information**: For more information about creating compute targets, see **Set up and use compute targets for model training**[2] in the Azure Machine Learning documentation.

## Attaching Azure Databricks Compute

An *unmanaged* compute target is one that is defined and managed outside of the Azure Machine Learning workspace; for example, an Azure virtual machine or an Azure Databricks cluster.

The code to attach an existing unmanaged compute target is similar to the code used to create a managed compute target, except that you must use the **ComputeTarget.attach()** method to attach the existing compute based on its target-specific configuration settings.

For example, the following code can be used to attach an existing Azure Databricks cluster:

```
from azureml.core import Workspace
from azureml.core.compute import ComputeTarget, DatabricksCompute

# Load the workspace from the saved config file
ws = Workspace.from_config()

# Specify a name for the compute (unique within the workspace)
compute_name = 'db_cluster'

# Define configuration for existing Azure Databricks cluster
db_workspace_name = 'db_workspace'
db_resource_group = 'db_resource_group'
db_access_token = '1234-abc-5678-defg-90...' # Get this from the Databricks
workspace
db_config = DatabricksCompute.attach_configuration(resource_group=db_re-
source_group,
                                                   workspace_name=db_work-
space_name,
                                                   access_token=db_access_
token)

# Create the compute
databricks_compute = ComputeTarget.attach(ws, compute_name, db_config)
databricks_compute.wait_for_completion(True)
```

2 https://aka.ms/AA70rrg

## Using Compute Targets

After you've created or attached compute targets in your workspace, you can use them to run specific workloads; such as experiments.

To use a particular compute target, you can specify it in the appropriate parameter for an experiment run configuration or estimator. For example, the following code configures an estimator to use the compute target named *aml-cluster*:

```
from azureml.core import Environment, ScriptRunConfig

compute_name = 'aml-cluster'

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=env,
                                compute_target=compute_name)
```

When an experiment is submitted, the run will be queued while the *aml-cluster* compute target is started and the specified environment created on it, and then the run will be processed on the compute environment.

Instead of specifying the name of the compute target, you can specify a **ComputeTarget** object, like this:

```
from azureml.core import Environment, ScriptRunConfig
from azureml.core.compute import ComputeTarget

compute_name = "aml-cluster"

training_cluster = ComputeTarget(workspace=ws, name=compute_name)

training_env = Environment.get(workspace=ws, name='training_environment')

script_config = ScriptRunConfig(source_directory='my_dir',
                                script='script.py',
                                environment=env,
                                compute_target=training_cluster)
```

## Lab: Work with Compute

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will work with environments and compute targets.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.

2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Work with compute** exercise.

## Module Review

In this module, you learned how to work with *environments* and *compute targets* to define an execution context for experiments.

Use the following review questions to check your learning.

### Question 1

*You need to create an environment from a Conda configuration (.yml) file.*
*Which method of the Environment class should you use?*

- ☐ create
- ☐ create_from_conda_specification
- ☐ create_from_existing_conda_environment

### Question 2

*You need to run a training script on compute that scales on-demand from 0 to 3 GPU-based nodes.*
*Which kind of compute target should you create?*

- ☐ Compute Instance
- ☐ Compute Cluster
- ☐ Inference Cluster

### Question 3

*Which ScriptRunConfig parameter causes the script to run on a compute cluster named train-cluster?*

- ☐ arguments=['--AmlCluster', 'train-cluster']
- ☐ environment='train-cluster'
- ☐ compute_target='train-cluster'

## Answers

**Question 1**

You need to create an environment from a Conda configuration (.yml) file.
Which method of the Environment class should you use?

☐ create

■ create_from_conda_specification

☐ create_from_existing_conda_environment

*Explanation*
*Use the create_from_conda_specification method to create an environment from a configuration file. The create method requires you to explicitly specify conda and pip packages, and the create_from_existing_conda_environment requires an existing environment on the computer.*

**Question 2**

You need to run a training script on compute that scales on-demand from 0 to 3 GPU-based nodes.
Which kind of compute target should you create?

☐ Compute Instance

■ Compute Cluster

☐ Inference Cluster

*Explanation*
*Use a compute cluster to create multiple nodes of GPU-enabled VMs that are started automatically as needed.*

**Question 3**

Which ScriptRunConfig parameter causes the script to run on a compute cluster named train-cluster?

☐ arguments=['--AmlCluster', 'train-cluster']

☐ environment='train-cluster'

■ compute_target='train-cluster'

*Explanation*
*Use the compute_target parameter to set the compute target.*

# Module 6 Orchestrating Machine Learning Workflows

## Introduction to Pipelines

## What is a Pipeline?

In Azure Machine Learning, a *pipeline* is a workflow of machine learning tasks in which each task is implemented as a *step*.

**Note**: The term *pipeline* is used extensively in machine learning, often with different meanings. For example, in Scikit-Learn, you can define pipelines that combine data preprocessing transformations with a training algorithm; and in Azure DevOps, you can define a build or release pipeline to perform the build and configuration tasks required to deliver software. The focus of this module is on Azure Machine Learning pipelines, which encapsulate steps that can be run as an experiment. However, bear in mind that it's perfectly feasible to have an Azure DevOps pipeline with a task that that initiates an Azure Machine Learning pipeline, which in turn includes a step that trains a model based on a Scikit-Learn pipeline!

Steps can be arranged sequentially or in parallel, enabling you to build sophisticated flow logic to orchestrate machine learning operations. Each step can be run on a specific compute target, making it possible to combine different types of processing as required to achieve an overall goal.

### Pipelines as Executable Processes

A pipeline can be executed as a process by running the pipeline as an experiment. Each step in the pipeline runs on its allocated compute target as part of the overall experiment run.

You can publish a pipeline as a REST endpoint, enabling client applications to initiate a pipeline run. You can also define a schedule for a pipeline, and have it run automatically at periodic intervals.

## Pipeline Steps

An Azure Machine Learning pipeline consists of one or more *steps* that perform tasks. There are many kinds of step supported by Azure Machine Learning pipelines, each with its own specialized purpose and configuration options.

## Types of Step

Common kinds of step in an Azure Machine Learning pipeline include:

- **PythonScriptStep**: Runs a specified Python script.
- **DataTransferStep**: Uses Azure Data Factory to copy data between data stores.
- **DatabricksStep**: Runs a notebook, script, or compiled JAR on a databricks cluster.
- **AdlaStep**: Runs a U-SQL job in Azure Data Lake Analytics.
- **ParallelRunStep** - Runs a Python script as a distributed task on multiple compute nodes.

**Note**: For a full list of supported step types, see **azure.pipeline.steps package documentation**[1].

## Defining Steps in a Pipeline

To create a pipeline, you must first define each step and then create a pipeline that includes the steps. The specific configuration of each step depends on the step type. For example the following code defines two **PythonScriptStep** steps to prepare data, and then train a model.

```
from azureml.pipeline.steps import PythonScriptStep

# Step to run a Python script
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster')

# Step to train a model
step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'train_model.py',
                         compute_target = 'aml-cluster')
```

After defining the steps, you can assign them to a pipeline, and run it as an experiment:

```
from azureml.pipeline.core import Pipeline
from azureml.core import Experiment

# Construct the pipeline
train_pipeline = Pipeline(workspace = ws, steps = [step1,step2])

# Create an experiment and run the pipeline
experiment = Experiment(workspace = ws, name = 'training-pipeline')
pipeline_run = experiment.submit(train_pipeline)
```

1 https://aka.ms/AA70rrh

## Passing Data Between Steps

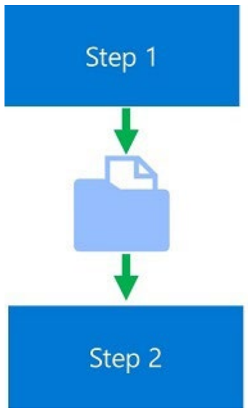
Often, a pipeline line includes at least one step that depends on the output of a preceding step. For example, you might use a step that runs a python script to preprocess some data, which must then be used in a subsequent step to train a model.

### The OutputFileDatasetConfig Object

The **OutputFileDatasetConfig** object is a special kind of dataset that:

- References a location in a datastore for interim storage of data.
- Creates a data dependency between pipeline steps.

You can view a **OutputFileDatasetConfig** object as an intermediary store for data that must be passed from a step to a subsequent step.



### OutputFileDatasetConfig Step Inputs and Outputs

To use a **OutputFileDatasetConfig** object to pass data between steps, you must:

1. Define a named **OutputFileDatasetConfig** object that references a location in a datastore. If nor explicit datastore is specified, the default datastore is used.
2. Pass the **OutputFileDatasetConfig** object as a script argument in steps that run scripts.
3. Include code in those scripts to write to the **OutputFileDatasetConfig** argument as an output or read it as an input.

For example, the following code defines a **OutputFileDatasetConfig** object that for the preprocessed data that must be passed between the steps.

```
from azureml.data import OutputFileDatasetConfig
from azureml.pipeline.steps import PythonScriptStep, EstimatorStep

# Get a dataset for the initial data
raw_ds = Dataset.get_by_name(ws, 'raw_dataset')

# Define a PipelineData object to pass data between steps
data_store = ws.get_default_datastore()
prepped_data = OutputFileDatasetConfig('prepped')
```

```
# Step to run a Python script
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster',
                         # Script arguments include PipelineData
                         arguments = ['--raw-ds', raw_ds.as_named_in-
put('raw_data'),
                                      '--out_folder', prepped_data])

# Step to run an estimator
step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'train_model.py',
                         compute_target = 'aml-cluster',
                         # Pass as script argument
                         arguments=['--training-data', prepped_data.as_in-
put()])
```

In the scripts themselves, you can obtain a reference to the **OutputFileDatasetConfig** object from the script argument, and use it like a local folder.

```
# code in data_prep.py
from azureml.core import Run
import argparse
import os

# Get the experiment run context
run = Run.get_context()

# Get arguments
parser = argparse.ArgumentParser()
parser.add_argument('--raw-ds', type=str, dest='raw_dataset_id')
parser.add_argument('--out_folder', type=str, dest='folder')
args = parser.parse_args()
output_folder = args.folder

# Get input dataset as dataframe
raw_df = run.input_datasets['raw_data'].to_pandas_dataframe()

# code to prep data (in this case, just select specific columns)
prepped_df = raw_df[['col1', 'col2', 'col3']]

# Save prepped data to the PipelineData location
os.makedirs(output_folder, exist_ok=True)
output_path = os.path.join(output_folder, 'prepped_data.csv')
prepped_df.to_csv(output_path)
```

## Pipeline Step Reuse

Pipelines with multiple long-running steps can take a significant time to complete, so Azure Machine Learning includes some default caching and reuse features to reduce this time.

### Managing Step Output Reuse

By default the step output from a previous pipeline run is reused without re-running the step as long as the script, source directory, and other parameters for the step have not changed. This can significantly reduce the time it takes to run a pipeline; however it can lead to stale results when changes to downstream data sources have not been accounted for.

To control reuse for an individual step, you can set the **allow_reuse** parameter in the step configuration, like this:

```
step1 = PythonScriptStep(name = 'prepare data',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster',
                         runconfig = run_config,
                         inputs=[raw_ds.as_named_input('raw_data')],
                         outputs=[prepped_data],
                         arguments = ['--folder', prepped_data]),
                         # Disable step reuse
                         allow_reuse = False)
```

### Forcing All Steps to Run

When you have multiple steps, you can force all of them to run regardless of individual reuse configuration by setting the **regenerate_outputs** parameter when submitting the pipeline experiment:

```
pipeline_run = experiment.submit(train_pipeline, regenerate_outputs=True)
```

## Publishing and Running Pipelines

## Pipeline Endpoints

After you have created a pipeline, you can publish it to create a REST endpoint through which the pipeline can be run on demand.

### Publishing a Pipeline

To publish a pipeline, you can call its **publish** method, as shown here:

```
published_pipeline = pipeline.publish(name='training_pipeline',
                                      description='Model training
pipeline',
                                      version='1.0')
```

Alternatively, you can call the **publish_pipeline** method on a successful run of the pipeline:

```
# Get the most recent run of the pipeline
pipeline_experiment = ws.experiments.get('training-pipeline')
run = list(pipeline_experiment.get_runs())[0]

# Publish the pipeline from the run
published_pipeline = run.publish_pipeline(name='training_pipeline',
                                          description='Model training
pipeline',
                                          version='1.0')
```

After the pipeline has been published, you can view it in Azure Machine Learning studio. You can also determine the URI of its endpoint like this:

```
rest_endpoint = published_pipeline.endpoint
print(rest_endpoint)
```

### Using a Published Pipeline

To initiate a published endpoint, you make an HTTP request to its REST endpoint, passing an authorization header with a token for a service principal with permission to run the pipeline, and a JSON payload specifying the experiment name. The pipeline is run asynchronously, so the response from a successful REST call includes the run ID. You can use this to track the run in Azure Machine Learning studio.

For example, the following Python code makes a REST request to run a pipeline and displays the returned run ID.

```
import requests

response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "run_training_pipeline"})
run_id = response.json()["Id"]
```

```
print(run_id)
```

## Pipeline Parameters

You can increase the flexibility of a pipeline by defining parameters.

### Defining Parameters for a Pipeline

To define parameters for a pipeline, create a **PipelineParameter** object for each parameter, and specify each parameter in at least one step.

For example, you could use the following code to include a parameter for a regularization rate in the Python script used by a pipeline step:

```
from azureml.pipeline.core.graph import PipelineParameter

reg_param = PipelineParameter(name='reg_rate', default_value=0.01)

...

step2 = PythonScriptStep(name = 'train model',
                         source_directory = 'scripts',
                         script_name = 'data_prep.py',
                         compute_target = 'aml-cluster',
                         # Pass parameter as script argument
                         arguments=['--in_folder', prepped_data,
                                    '--reg', reg_param],
                         inputs=[prepped_data])
```

**Note**: You must define parameters for a pipeline before publishing it.

### Running a Pipeline with a Parameter

After you publish a parameterized pipeline, you can pass parameter values in the JSON payload for the REST interface:

```
response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "run_training_pipeline",
                               "ParameterAssignments": {"reg_rate": 0.1}})
```

## Scheduling Pipelines

After you have published a pipeline, you can initiate it on demand through its REST endpoint, or you can have the pipeline run automatically based on a periodic schedule or in response to data updates.

### Scheduling a Pipeline for Periodic Intervals

To schedule a pipeline to run at periodic intervals, you must define a **ScheduleRecurrence** that determines the run frequency, and use it to create a **Schedule**.

For example, the following code schedules a daily run of a published pipeline.

```
from azureml.pipeline.core import ScheduleRecurrence, Schedule

daily = ScheduleRecurrence(frequency='Day', interval=1)
pipeline_schedule = Schedule.create(ws, name='Daily Training',
                                        description='trains model every
day',
                                        pipeline_id=published_pipeline.id,
                                        experiment_name='Training_Pipe-
line',
                                        recurrence=daily)
```

## Triggering a Pipeline Run on Data Changes

To schedule a pipeline to run whenever data changes, you must create a **Schedule** that monitors a specified path on a datastore, like this:

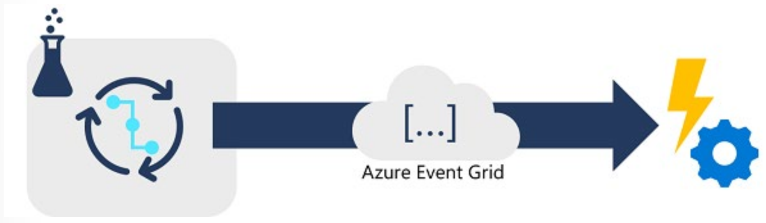
```
from azureml.core import Datastore
from azureml.pipeline.core import Schedule

training_datastore = Datastore(workspace=ws, name='blob_data')
pipeline_schedule = Schedule.create(ws, name='Reactive Training',
                                    description='trains model on data
change',
                                    pipeline_id=published_pipeline_id,
                                    experiment_name='Training_Pipeline',
                                    datastore=training_datastore,
                                    path_on_datastore='data/training')
```

# Event-Driven Workflows

So far we've explored ways to automate machine learning operations through pipelines.

Azure Machine Learning integrates with Azure Event Grid to support event-driven workflows that go beyond pipelines to create a framework for triggering actions in response to specific events.



Azure Machine Learning events include:

- Run completion
- Run failure
- Model registration
- Model deployment

- Data drift detection

You can use events trigger actions such as:

- Azure Functions
- Azure Logic Apps
- Azure Event Hubs
- Azure Data Factory pipelines
- Generic webhooks

For example, you can define an event based on the completion of a pipeline run that trains a model, and use that event to trigger an action such as an Azure Function that deploys the model as a service in a staging environment for testing. Or you might create an event that fires when a pipeline run fails, and use an Azure Logic App to send a notification email to an operator.

## Lab: Create a Pipeline

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will create and publish a pipeline.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Create a pipeline** exercise.

## Module Review

In this module, you learned how to create pipelines and publish them as REST services.

Use the following review questions to check your learning.

### Question 1

*What type of object should you use to pass data between pipeline steps?*

☐ Datastore

☐ Dataset

☐ OutputFileDatasetConfig

## Question 2

*You plan to use the Schedule.create method to create a schedule for a published pipeline. What kind of object must you create first to configure how frequently the pipeline runs?*

- ☐ ScheduleRecurrence
- ☐ Datastore
- ☐ PipelineParameter

## Answers

**Question 1**

What type of object should you use to pass data between pipeline steps?

☐ Datastore

☐ Dataset

■ OutputFileDatasetConfig

*Explanation*
*Use an OutputFileDatasetConfig to pass data between steps.*

**Question 2**

You plan to use the Schedule.create method to create a schedule for a published pipeline. What kind of object must you create first to configure how frequently the pipeline runs?

■ ScheduleRecurrence

☐ Datastore

☐ PipelineParameter

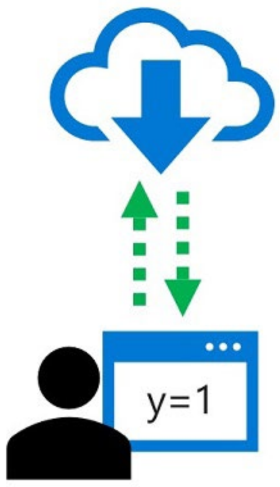
*Explanation*
*You need a ScheduleRecurrence object to create a schedule that runs at a regular interval.*

# Module 7 Deploying and Consuming Models

## Real-time Inferencing

### What is Real-Time Inferencing?

In machine learning, *inferencing* refers to the use of a trained model to predict labels for new data on which the model has not been trained. Often, the model is deployed as part of a service that enables applications to request immediate, or *real-time*, predictions for individual or small numbers of data observations.



In Azure Machine learning, you can create real-time inferencing solutions by deploying a model as a real-time service, hosted in a containerized platform such as Azure Kubernetes Services (AKS).

### Deploying a Real-Time Inferencing Service

You can deploy a model as a real-time web service to several kinds of compute target, including local compute, an Azure Machine Learning compute instance,  an Azure Container Instance (ACI), an Azure

Kubernetes Service (AKS) cluster, an Azure Function, or an Internet of Things (IoT) module. Azure Machine Learning uses *containers* as a deployment mechanism, packaging the model and the code to use it as an image that can be deployed to a container in your chosen compute target.

**Note**: Deployment to a local service, a compute instance, or an ACI is a good choice for testing and development. For production, you should deploy to a target that meets the specific performance, scalability, and security needs of your application architecture.

To deploy a model as a real-time inferencing service, you must perform the following tasks:

## 1. Register a trained model

After successfully training a model, you must register it in your Azure Machine Learning workspace. Your real-time service will then be able to load the model when required.

To register a model from a local file, you can use the **register** method of the **Model** object as shown here:

```
from azureml.core import Model

classification_model = Model.register(workspace=ws,
                       model_name='classification_model',
                       model_path='model.pkl', # local path
                       description='A classification model')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model')
```

## 2. Define an inference configuration

The model will be deployed as a service that consist of:

- A script to load the model and return predictions for submitted data.
- An environment in which the script will be run.

You must therefore define the script and environment for the service.

## Create an entry script

Create the *entry script* (sometimes referred to as the *scoring script*) for the service as a Python (.py) file. It must include two functions:

- **init()**: Called when the service is initialized.
- **run(raw_data)**: Called when new data is submitted to the service.

Typically, you use the **init** function to load the model from the model registry, and use the **run** function to generate predictions from the input data. The following example script shows this pattern:

```
import json
import joblib
import numpy as np
import os

# Called when the service is loaded
def init():
    global model
    # Get the path to the registered model file and load it
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'model.pkl')
    model = joblib.load(model_path)

# Called when a request is received
def run(raw_data):
    # Get the input data as a numpy array
    data = np.array(json.loads(raw_data)['data'])
    # Get a prediction from the model
    predictions = model.predict(data)
    # Return the predictions as any JSON serializable format
    return predictions.tolist()
```

Save the script in a folder so you can easily identify it later. For example, you might save the script above as *score.py* in a folder named *service_files*.

## Create an environment

Your service requires a Python environment in which to run the entry script, which you can define by creating an **Environment** that contains the required packages:

```
from azureml.core import Environment

service_env = Environment(name='service-env')
python_packages = ['scikit-learn', 'numpy'] # whatever packages your entry
script uses
for package in python_packages:
    service_env.python.conda_dependencies.add_pip_package(package)
```

## Combine the script and environment in an InferenceConfig

After creating the entry script and environment, you can combine them in an **InferenceConfig** for the service like this:

```
from azureml.core.model import InferenceConfig

classifier_inference_config = InferenceConfig(source_directory = 'service_
files',
                                              entry_script="score.py",
                                              environment=service_env)
```

## 3. Define a Deployment Configuration

Now that you have the entry script and environment, you need to configure the compute to which the service will be deployed. If you are deploying to an AKS cluster, you must create the cluster and a compute target for it before deploying:

```
from azureml.core.compute import ComputeTarget, AksCompute

cluster_name = 'aks-cluster'
compute_config = AksCompute.provisioning_configuration(location='eastus')
production_cluster = ComputeTarget.create(ws, cluster_name, compute_config)
production_cluster.wait_for_completion(show_output=True)
```

With the compute target created, you can now define the deployment configuration, which sets the target-specific compute specification for the containerized deployment:

```
from azureml.core.webservice import AksWebservice

classifier_deploy_config = AksWebservice.deploy_configuration(cpu_cores = 1,
                                                               memory_gb =
1)
```

The code to configure an ACI deployment is similar, except that you do not need to explicitly create an ACI compute target, and you must use the **deploy_configuration** class from the **azureml.core.webservice.AciWebservice** namespace. Similarly, you can use the **azureml.core.webservice.LocalWebservice** namespace to configure a local Docker-based service.

**Note**: To deploy a model to an Azure Function, you do not need to create a deployment configuration. Instead, you need to package the model based on the type of function trigger you want to use. This functionality is in preview at the time of writing. For more details, see **Deploy a machine learning model to Azure Functions**[1] in the Azure Machine Learning documentation.

## 4. Deploy the Model

After all of the configuration is prepared, you can deploy the model. The easiest way to do this is to call the **deploy** method of the **Model** class, like this:

```
from azureml.core.model import Model

service = Model.deploy(workspace=ws,
                       name = 'classifier-service',
                       models = [classification_model],
                       inference_config = classifier_inference_config,
                       deployment_config = classifier_deploy_config,
                       deployment_target = production_cluster)
service.wait_for_deployment(show_output = True)
```

For ACI or local services, you can omit the **deployment_target** parameter (or set it to **None**).

1 https://aka.ms/AA70rrn

**More Information**: For more information about deploying models with Azure Machine Learning, see **Deploy models with Azure Machine Learning**[2] in the documentation.

## Consuming a Real-time Inferencing Service

After deploying a real-time service, you can consume it from client applications to predict labels for new data cases.

### Using the Azure Machine Learning SDK

For testing, you can use the Azure Machine Learning SDK to call a web service through the **run** method of a **WebService** object that references the deployed service. Typically, you send data to the **run** method in JSON format with the following structure:

```
{
  "data":[
      [0.1,2.3,4.1,2.0], // 1st case
      [0.2,1.8,3.9,2.1],  // 2nd case,
      ...
  ]
}
```

The response from the **run** method is a JSON collection with a prediction for each case that was submitted in the data. The following code sample calls a service and displays the response:

```
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Call the web service, passing the input data
response = service.run(input_data = json_data)

# Get the predictions
predictions = json.loads(response)

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i]), predictions[i] )
```

### Using a REST Endpoint

In production, most client applications will not include the Azure Machine Learning SDK, and will consume the service through its REST interface. You can determine the endpoint of a deployed service in

2 https://aka.ms/AA70zfv

Azure machine Learning studio, or by retrieving the **scoring_uri** property of the **Webservice** object in the SDK, like this:

```
endpoint = service.scoring_uri
print(endpoint)
```

With the endpoint known, you can use an HTTP POST request with JSON data to call the service. The following example shows how to do this using Python:

```
import requests
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Set the content type in the request headers
request_headers = { 'Content-Type':'application/json' }

# Call the service
response = requests.post(url = endpoint,
                         data = json_data,
                         headers = request_headers)

# Get the predictions from the JSON response
predictions = json.loads(response.json())

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i]), predictions[i] )
```

## Authentication

In production, you will likely want to restrict access to your services by applying authentication. There are two kinds of authentication you can use:

- **Key**: Requests are authenticated by specifying the key associated with the service.
- **Token**: Requests are authenticated by providing a JSON Web Token (JWT).

By default, authentication is disabled for ACI services, and set to key-based authentication for AKS services (for which primary and secondary keys are automatically generated). You can optionally configure an AKS service to use token-based authentication (which is not supported for ACI services).

Assuming you have an authenticated session established with the workspace, you can retrieve the keys for a service by using the **get_keys** method of the **WebService** object associated with the service:

```
primary_key, secondary_key = service.get_keys()
```

For token-based authentication, your client application needs to use service-principal authentication to verify its identity through Azure Active Directory (Azure AD) and call the **get_token** method of the service to retrieve a time-limited token.

To make an authenticated call to the service's REST endpoint, you must include the key or token in the request header like this:

```
import requests
import json

# An array of new data cases
x_new = [[0.1,2.3,4.1,2.0],
         [0.2,1.8,3.9,2.1]]

# Convert the array to a serializable list in a JSON document
json_data = json.dumps({"data": x_new})

# Set the content type in the request headers
request_headers = { "Content-Type":"application/json",
                    "Authorization":"Bearer " + key_or_token }

# Call the service
response = requests.post(url = endpoint,
                         data = json_data,
                         headers = request_headers)

# Get the predictions from the JSON response
predictions = json.loads(response.json())

# Print the predicted class for each case.
for i in range(len(x_new)):
    print (x_new[i]), predictions[i] )
```

## Troubleshooting a Real-Time Inferencing Service

There are a lot of elements to a real-time service deployment, including the trained model, the runtime environment configuration, the scoring script, the container image, and the container host. Troubleshooting a failed deployment, or an error when consuming a deployed service can be complex.

### Check the Service State

As an initial troubleshooting step, you can check the status of a service by examining its **state**:

```
from azureml.core.webservice import AksWebservice

# Get the deployed service
service = AksWebservice(name='classifier-service', workspace=ws)

# Check its state
print(service.state)
```

**Note**: To view the **state** of a service, you must use the compute-specific service type (for example **AksWebservice**) and not a generic **WebService** object.

For an operational service, the state should be *Healthy*.

### Review Service Logs

If a service is not healthy, or you are experiencing errors when using it, you can review its logs:

```
print(service.get_logs())
```

The logs include detailed information about the provisioning of the service, and the requests it has processed; and can often provide an insight into the cause of unexpected errors.

### Deploy to a Local Container

Deployment and runtime errors can be easier to diagnose by deploying the service as a container in a local Docker instance, like this:

```
from azureml.core.webservice import LocalWebservice

deployment_config = LocalWebservice.deploy_configuration(port=8890)
service = Model.deploy(ws, 'test-svc', [model], inference_config, deploy-
ment_config)
```

You can then test the locally deployed service using the SDK:

```
print(service.run(input_data = json_data))
```

You can then troubleshoot runtime issues by making changes to the scoring file that is referenced in the inference configuration, and reloading the service without redeploying it (something you can only do with a local service):

```
service.reload()
print(service.run(input_data = json_data))
```

## Lab: Create a Real-time Inference Service

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will deploy a model as a real-rime inference service.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

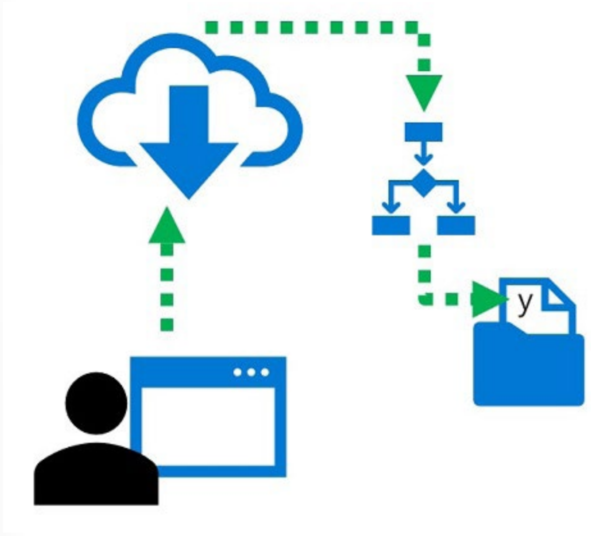
1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.

3. Complete the **Create a real-time inference service** exercise.

## Batch Inferencing

## What is Batch Inferencing?

In many production scenarios, long-running tasks that operate on large volumes of data are performed as *batch* operations. In machine learning, *batch inferencing* is used to apply a predictive model to multiple cases asynchronously - usually writing the results to a file or database.



In Azure Machine Learning, you can implement batch inferencing solutions by creating a pipeline that includes a step to read the input data, load a registered model, predict labels, and write the results as its output. Typically, this step is performed using a **ParallelRunStep**, enabling large volumes of data to be scored in a distributed fashion with the results collated as a single output.

## Creating a Batch Inferencing Pipeline

To create a batch inferencing pipeline, perform the following tasks:

### 1. Register the Model

Just as with a real-time inferencing service, to use a trained model in a batch inferencing pipeline, you must register it in your Azure Machine Learning workspace.

To register a model from a local file, you can use the **register** method of the **Model** object as shown here:

```
from azureml.core import Model

classification_model = Model.register(workspace=ws,
                                      model_name='classification_model',
                                      model_path='model.pkl', # local path
                                      description='A classification model')
```

Alternatively, if you have a reference to the **Run** used to train the model, you can use its **register_model** method as shown here:

```
run.register_model( model_name='classification_model',
                    model_path='outputs/model.pkl', # run outputs path
                    description='A classification model')
```

## 2. Create an Scoring Script

Just like a real-time inferencing service, a batch inferencing service requires a scoring script to load the model and use it to predict new values. It must include two functions:

- **init()**: Called when the pipeline is initialized.
- **run(mini_batch)**: Called for each batch of data to be processed.

Typically, you use the **init** function to load the model from the model registry, and use the **run** function to generate predictions from each batch of data and return the results. The following example script shows this pattern:

```
import os
import numpy as np
from azureml.core import Model
import joblib

def init():
    # Runs when the pipeline step is initialized
    global model

    # load the model
    model_path = Model.get_model_path('classification_model')
    model = joblib.load(model_path)

def run(mini_batch):
    # This runs for each batch
    resultList = []

    # process each file in the batch
    for f in mini_batch:
        # Read comma-delimited data into an array
        data = np.genfromtxt(f, delimiter=',')
        # Reshape into a 2-dimensional array for model input
        prediction = model.predict(data.reshape(1, -1))
        # Append prediction to results
        resultList.append("{}: {}".format(os.path.basename(f), predic-
tion[0]))
    return resultList
```

## 3. Create a Pipeline with a ParallelRunStep

Azure Machine Learning provides a type of pipeline step specifically for performing parallel batch inferencing. Using the **ParallelRunStep** class, you can read batches of files from a **File** dataset and write the processing output to a **OutputFileDatasetConfig**. Additionally, you can set the **output_action** setting for the step to "append_row", which will ensure that all instances of the step being run in parallel will collate their results to a single output file named *parallel_run_step.txt*:

```
from azureml.pipeline.steps import ParallelRunConfig, ParallelRunStep
from azureml.data import OutputFileDatasetConfig
from azureml.pipeline.core import Pipeline

# Get the batch dataset for input
batch_data_set = ws.datasets['batch-data']

# Set the output location
default_ds = ws.get_default_datastore()
output_dir = OutputFileDatasetConfig(name='inferences')

# Define the parallel run step step configuration
parallel_run_config = ParallelRunConfig(
    source_directory='batch_scripts',
    entry_script="batch_scoring_script.py",
    mini_batch_size="5",
    error_threshold=10,
    output_action="append_row",
    environment=batch_env,
    compute_target=aml_cluster,
    node_count=4)

# Create the parallel run step
parallelrun_step = ParallelRunStep(
    name='batch-score',
    parallel_run_config=parallel_run_config,
    inputs=[batch_data_set.as_named_input('batch_data')],
    output=output_dir,
    arguments=[],
    allow_reuse=True
)
# Create the pipeline
pipeline = Pipeline(workspace=ws, steps=[parallelrun_step])
```

## 4. Run the pipeline and Retrieve the Step Output

After your pipeline has been defined, you can run it and wait for it to complete. Then you can retrieve the parallel_run_step.txt file from the output of the step to view the results.

```
from azureml.core import Experiment

# Run the pipeline as an experiment
pipeline_run = Experiment(ws, 'batch_prediction_pipeline').submit(pipeline)
pipeline_run.wait_for_completion(show_output=True)

# Get the outputs from the first (and only) step
prediction_run = next(pipeline_run.get_children())
prediction_output = prediction_run.get_output_data('inferences')
```

```
prediction_output.download(local_path='results')

# Find the parallel_run_step.txt file
for root, dirs, files in os.walk('results'):
    for file in files:
        if file.endswith('parallel_run_step.txt'):
            result_file = os.path.join(root,file)

# Load and display the results
df = pd.read_csv(result_file, delimiter=":", header=None)
df.columns = ["File", "Prediction"]
print(df)
```

## Publishing a Batch Inferencing Service

Just like any pipeline, you can publish a batch inferencing pipeline as a REST service:

```
published_pipeline = pipeline_run.publish_pipeline(name='Batch_Prediction_
Pipeline',
                                                   description='Batch 
pipeline',
                                                   version='1.0')
rest_endpoint = published_pipeline.endpoint
```

Once published, you can use the service endpoint to initiate a batch inferencing job:

```
import requests

response = requests.post(rest_endpoint,
                         headers=auth_header,
                         json={"ExperimentName": "Batch_Prediction"})
run_id = response.json()["Id"]
```

You can also schedule the published pipeline to have it run automatically:

```
from azureml.pipeline.core import ScheduleRecurrence, Schedule

weekly = ScheduleRecurrence(frequency='Week', interval=1)
pipeline_schedule = Schedule.create(ws, name='Weekly Predictions',
                                        description='batch inferencing',
                                        pipeline_id=published_pipeline.id,
                                        experiment_name='Batch_Prediction',
                                        recurrence=weekly)
```

## Lab: Create a Batch Inference Service

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will deploy a model as a batch inference service.

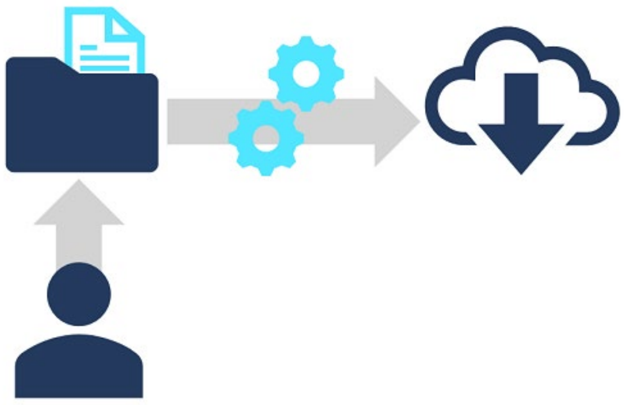
## Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Create a batch inference service** exercise.

## Continuous Integration and Delivery

### What is Continuous Integration and Delivery (CI/CD)?



CI/CD is a core element of DevOps. While implementation details can vary across organizations, the principles tend to be consistent:

- Code and other assets for application solutions are managed in a source control system – often based on the popular Git framework. Common source control systems include Azure Repos (either standalone or as part of an Azure DevOps service) and GitHub.
- Developers push updates to the source control system, triggering build and release processes that automate validation, integration, testing, and deployment of software applications. These processes may be implemented as Azure Pipelines (again, available as a standalone service or as a component of Azure DevOps) or GitHub actions.

### Azure Machine Learning and Azure Pipelines

Azure Pipelines is a commonly used mechanism for automating software build and release processes.

At its simplest, Azure Pipelines provides a way to define a process that runs on-demand or in response to a source control event (such as pushing an update to a code file). You can use Azure CLI commands or Python scripts in a pipeline, making it an idea way to automate Azure Machine Learning operations; such as training, registering, or deploying a model.

Additionally, you can install the Machine Learning extension for Azure pipelines to provide explicit integration with Azure Machine Learning. This enables you to monitor a workspace for new model registration events, and run a pipeline in response – for example to check the predictive performance of the newly registered model, and if it performs better than a previously deployed model, deploy the new one. You can also leverage Azure Machine Learning specific tasks in an Azure pipeline to run a published Azure Machine Learning pipeline, profile a model (to determine the appropriate deployment environment based on expected utilization), or deploy a model as a service.

## Azure Machine Learning and GitHub Actions

When using GitHub as a source control repository, you can use GitHub actions to automate processes. For example, you could create a GitHub action that runs an Azure Machine learning pipeline when a developer pushes new code to a branch in the GitHub repo.

GitHub actions that are specific to Azure Machine Learning include:

- **aml-run** (runs an experiment or pipeline)
- **aml-registermodel** (registers a trained model)
- **aml-deploy** (deploys a model as a service)

## Module Review

In this module, you learned how to deploy a model to real-time and batch inference services.

Use the following review questions to check your learning.

### Question 1

*You want to deploy the model as a containerized real-time service with high scalability and token-based security.*
*What kind of deployment target should you use?*

☐ An Azure Container Instance (ACI)

☐ An Azure Kubernetes Service (AKS) inference cluster

☐ A multi-node compute cluster with GPUs

### Question 2

*Which functions must the entry script for a real-time service implement?*

☐ init and run

☐ main and score

☐ load and predict

### Question 3

*You want to implement a batch inference pipeline that distributes scoring on multiple nodes.*
*Which kind of pipeline step should you use?*

☐ PythonScriptStep

☐ AdlaStep

☐ ParallelRunStep

## Answers

**Question 1**

You want to deploy the model as a containerized real-time service with high scalability and token-based security.
What kind of deployment target should you use?

☐ An Azure Container Instance (ACI)

■ An Azure Kubernetes Service (AKS) inference cluster

☐ A multi-node compute cluster with GPUs

*Explanation*
*An AKS inference cluster offers the best scalability and security for production services.*

**Question 2**

Which functions must the entry script for a real-time service implement?

■ init and run

☐ main and score

☐ load and predict

*Explanation*
*A scoring (or entry) script must implement init and run functions.*

**Question 3**

You want to implement a batch inference pipeline that distributes scoring on multiple nodes.
Which kind of pipeline step should you use?

☐ PythonScriptStep

☐ AdlaStep

■ ParallelRunStep

*Explanation*
*You should use a ParallelRunStep step to run the scoring script in parallel.*

# Module 8 Training Optimal Models

## Hyperparameter Tuning

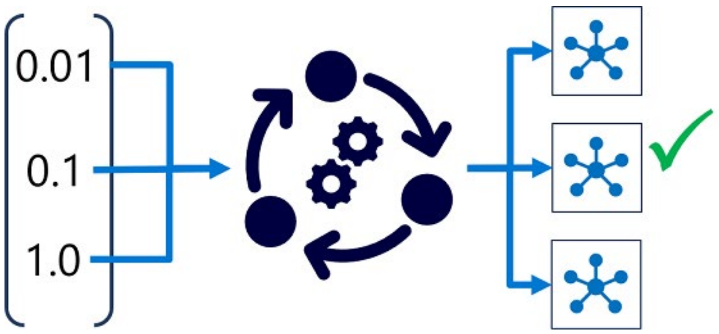
### What is Hyperparameter Tuning?

In machine learning, models are trained to predict unknown labels for new data based on correlations between known labels and features found in the training data. Depending on the algorithm used, you may need to specify *hyperparameters* to configure how the model is trained. For example, the *logistic regression* algorithm uses a *regularization rate* hyperparameter to counteract overfitting; and deep learning techniques for convolutional neural networks (CNNs) use hyperparameters like *learning rate* to control how weights are adjusted during training, and *batch size* to determine how many data items are included in each training batch.

**Why *hyper*parameter?**

Machine Learning is an academic field with its own particular terminology. Data scientists refer to the values determined from the training features as *parameters*, so a different term is required for values that are used to configure training behaviour but which are ***not*** derived from the training data - hence the term *hyperparameter*.

The choice of hyperparameter values can significantly affect the resulting model, making it important to select the best possble values for your particular data and predictive performance goals.



## Tuning Hyperparameters

Hyperparameter tuning is accomplished by training the multiple models, using the same algorithm and training data but different hyperparameter values. The resulting model from each training run is then evaluated to determine the performance metric for which you want to optimize (for example, *accuracy*), and the best-performing model is selected.

In Azure Machine Learning, you achieve this through an experiment that consists of a *hyperdrive* run, which initiates a child run for each hyperparameter combination to be tested. Each child run uses a training script with parameterized hyperparameter values to train a model, and logs the target performance metric achieved by the trained model.

# Hyperparameter Search Space

The set of hyperparameter values tried during hyperparameter tuning is known as the *search space*. The definition of the range of possible values that can be chosen depends on the type of hyperparameter.

## Discrete Hyperparameters

Some hyperparameters require *discrete* values - in other words, you must select the value from a particular set of possibilities. You can define a search space for a discrete parameter using a **choice** from a list of explicit values, which you can define as a Python **list** (`choice([10,20,30])`), a **range** (`choice(range(1,10)])`), or an arbitrary set of comma-separated values (`choice(30,50,100)`)

You can also select discrete values from any of the following discrete distributions:

- qnormal
- quniform
- qlognormal
- qloguniform

## Continuous Hyperparameters

Some hyperparameters are *continuous* - in other words you can use any value along a scale. To define a search space for these kinds of value, you can use any of the following distribution types:

- normal
- uniform
- lognormal
- loguniform

## Defining a Search Space

To define a search space for hyperparameter tuning, create a dictionary with the appropriate parameter expression for each named hyperparameter. For example, the following search space indicates that the **batch_size** hyperparameter can have the value 20, 50, or 100; and the **learning_rate** hyperparameter can have any value from a normal distribution with a mean of 10 and a standard deviation of 3.

```
from azureml.train.hyperdrive import choice, normal

param_space = {
```

```
                '--batch_size': choice(16, 32, 64),
                '--learning_rate': normal(10, 3)
             }
```

## Hyperparameter Sampling

The specific values used in a hyperparameter tuning run depend on the type of *sampling* used.

### Grid Sampling

Grid sampling can only be employed when all hyperparameters are discrete, and is used to try every possible combination of parameters in the search space.

For example, in the following code, grid sampling is used to try every possible combination of discrete *batch_size* and *learning_rate* value:

```
from azureml.train.hyperdrive import GridParameterSampling, choice

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': choice(0.01, 0.1, 1.0)
              }

param_sampling = GridParameterSampling(param_space)
```

### Random Sampling

Random sampling is used to randomly select a value for each hyperparameter, which can be a mix of discrete and continuous values:

```
from azureml.train.hyperdrive import RandomParameterSampling, choice,
normal

param_space = {
                 '--batch_size': choice(16, 32, 64),
                 '--learning_rate': normal(10, 3)
              }

param_sampling = RandomParameterSampling(param_space)
```

### Bayesian Sampling

Bayesian sampling chooses hyperparameter values based on the Bayesian optimization algorithm, which tries to select parameter combinations that will result in improved performance from the previous selection:

```
from azureml.train.hyperdrive import BayesianParameterSampling, choice,
uniform

param_space = {
```

```
                  '--batch_size': choice(16, 32, 64),
                  '--learning_rate': uniform(0.5, 0.1)
               }

param_sampling = BayesianParameterSampling(param_space)
```

**Note**: You can only use Bayesian sampling with **choice**, **uniform**, and **quniform** parameter expressions, and you can't combine it with an early-termination policy.

## Early Termination Policy

With a sufficiently large hyperparameter search space, it could take many iterations (child runs) to try every possible combination. Typically, you set a maximum number of iterations, but this could still result in a large number of runs that don't result in a better model than a combination that has already been tried.

To help prevent wasting time, you can set an early termination policy that abandons runs that are unlikely to produce a better result than previously completed runs. The policy is evaluated at an *evaluation_interval* you specify, based on each time the target performance metric is logged. You can also set a *delay_evaluation* parameter to avoid evaluating the policy until a minimum number of iterations have been completed.

**Note**: Early termination is particularly useful for deep learning scenarios where a deep neural network (DNN) is trained iteratively over a number of *epochs*. The training script can report the target metric after each epoch, and if the run is significantly underperforming previous runs after the same number of intervals, it can be abandoned.

### Bandit Policy

You can use a bandit policy to stop a run if the target performance metric underperforms the best run so far by a specified margin.

```
from azureml.train.hyperdrive import BanditPolicy

early_termination_policy = BanditPolicy(slack_amount = 0.2,
                                        evaluation_interval=1,
                                        delay_evaluation=5)
```

This example applies the policy for every iteration after the first five, and abandons runs where the reported target metric is 0.2 or more worse than the best performing run after the same number of intervals.

**Note**: You can also apply a bandit policy using a slack *factor*, which compares the performance metric as a ratio  rather than an absolute value.

### Median Stopping Policy

A median stopping policy abandons runs where the target performance metric is worse than the median of the running averages for all runs.

```
from azureml.train.hyperdrive import MedianStoppingPolicy

early_termination_policy = MedianStoppingPolicy(evaluation_interval=1,
                                                        delay_evaluation=5)
```

## Truncation Selection Policy

A truncation selection policy cancels the lowest performing *X*% of runs at each evaluation interval based on the *truncation_percentage* value you specify for *X*.

```
from azureml.train.hyperdrive import TruncationSelectionPolicy

early_termination_policy = TruncationSelectionPolicy(truncation_percent-
age=10,
                                                     evaluation_interval=1,
                                                     delay_evaluation=5)
```

## Tuning Hyperparameters with Hyperdrive

In Azure Machine Learning, you can tune hyperparameters by running a *Hyperdrive* experiment.

### Creating a Training Script for Hyperparameter Tuning

To run a Hyperdrive experiment, you need to create a training script just the way you would for any other training experiment, except that your script ***must***:

- Include an argument for each hyperparameter you want to vary.
- Log the target performance metric. This is what enables the hyperdrive run to evaluate the performance of the child runs it initiates, and identify the one that produces the best performing model.

For example, the following script trains a logistic regression model using a **–regularization** argument to set the *regularization rate* hyperparameter, and logs the *accuracy* metric with the name **Accuracy**:

```
import argparse
import joblib
from azureml.core import Run
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Get regularization hyperparameter
parser = argparse.ArgumentParser()
parser.add_argument('--regularization', type=float, dest='reg_rate', de-
fault=0.01)
args = parser.parse_args()
reg = args.reg_rate

# Get the experiment run context
run = Run.get_context()

# load the training dataset
data = run.input_datasets['training_data'].to_pandas_dataframe()
```

```
# Separate features and labels, and split for training/validatiom
X = data[['feature1','feature2','feature3','feature4']].values
y = data['label'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)

# Train a logistic regression model with the reg hyperparameter
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_
train)

# calculate and log accuracy
y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
run.log('Accuracy', np.float(acc))

# Save the trained model
os.makedirs('outputs', exist_ok=True)
joblib.dump(value=model, filename='outputs/model.pkl')

run.complete()
```

**Note**: Note that in the Scikit-Learn **LogisticRegression** class, **C** is the *inverse* of the regularization rate; hence C=1/reg.

## Configuring and Running a Hyperdrive Experiment

To prepare the Hyperdrive experiment, you must use a **HyperDriveConfig** object to configure the experiment run, as shown here:

```
from azureml.core import Experiment
from azureml.train.hyperdrive import HyperDriveConfig, PrimaryMetricGoal

# Assumes ws, script_config and param_sampling are already defined

hyperdrive = HyperDriveConfig(run_config=script_config,
                              hyperparameter_sampling=param_sampling,
                              policy=None,
                              primary_metric_name='Accuracy',
                              primary_metric_goal=PrimaryMetricGoal.MAXI-
MIZE,
                              max_total_runs=6,
                              max_concurrent_runs=4)

experiment = Experiment(workspace = ws, name = 'hyperdrive_training')
hyperdrive_run = experiment.submit(config=hyperdrive)
```

## Monitoring and Reviewing Hyperdrive Runs

You can monitor Hyperdrive experiments in Azure Machine Learning studio, or by using the Jupyter Notebooks **RunDetails** widget.

The experiment will initiate a child run for each hyperparameter combination to be tried, and you can retrieve the logged metrics these runs using the following code:

```
for child_run in run.get_children():
    print(child_run.id, child_run.get_metrics())
```

You can also list all runs in descending order of performance like this:

```
for child_run in hyperdrive_.get_children_sorted_by_primary_metric():
    print(child_run)
```

To retrieve the best performing run, you can use the following code:

```
best_run = hyperdrive_run.get_best_run_by_primary_metric()
```

## Lab: Tune Hyperparameters

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will run a hyperdrive experiment to tune hyperparameters.

### Instructions

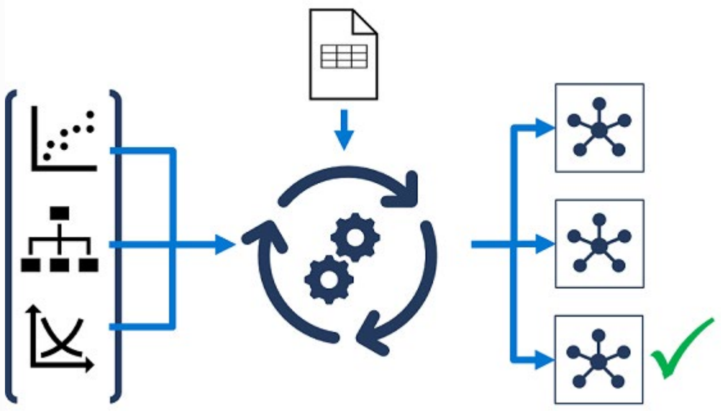
You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Tune hyperparameters** exercise.

## Automated Machine Learning

## Automated Machine Learning - A Reminder

While hyperparameter tuning enables you to try multiple hyperparameter combinations when using a specific algorithm to train a model, *Automated Machine Learning* enables you to try multiple algorithms and preprocessing transformations with your data. This, combined with scalable cloud-based compute makes it possible to find the best performing model for your data without the huge amount of time-consuming manual trial and error that would otherwise be required.



### Automated Machine Learning in Azure Machine Learning

Azure Machine Learning includes support for automated machine learning through a visual interface in Azure Machine Learning studio. You can use the Azure Machine Learning SDK to run automated machine learning experiments.

## Automated Machine Learning Data

Automated machine learning is designed to enable you to simply bring your data, and have Azure Machine Learning figure out how best to train a model from it.

When using the Automated Machine Learning user interface in Azure Machine Learning studio, you can create or select a dataset to be used as the input for your automated machine learning experiment.

When using the SDK to run an automated machine learning experiment, you can submit the data in the following ways:

- Specify a dataset or dataframe of *training data* that includes features and the label to be predicted.
- Optionally, specify a second *validation data* dataset or dataframe that will be used to validate the trained model. if this is not provided, Azure Machine Learning will apply cross-validation using the training data.

## Running an Automated Machine Learning Experiment

To run an automated machine learning experiment, you can either use the user interface in Azure Machine Learning studio, or submit an experiment using the SDK.

### Configuring an Automated Machine Learning Experiment

The user interface provides an intuitive way to select options for your automated machine learning experiment. When using the SDK, you have greater flexibility, and you can set experiment options using the **AutoMLConfig** class, as shown in the following example:

```
from azureml.train.automl import AutoMLConfig

automl_run_config = RunConfiguration(framework='python')
automl_config = AutoMLConfig(name='Automated ML Experiment',
                             task='classification',
                             primary_metric = 'AUC_weighted',
                             compute_target=aml_compute,
                             training_data = train_dataset,
                             validation_data = test_dataset,
                             label_column_name='Label',
                             featurization='auto',
                             iterations=12,
                             max_concurrent_iterations=4)
```

One of the most important settings you must specify is the **primary_metric**. This is the target performance metric for which the optimal model will be determined. Azure Machine Learning supports a set of named metrics for each type of task. To retrieve the list of metrics available for a particular task type, you can use the **get_primary_metrics** function as shown here:

```
from azureml.train.automl.utilities import get_primary_metrics

get_primary_metrics('classification')
```

**More Information**: You can find a full list of primary metrics and their definitions in **Understand automated machine learning results**[1] in the documentation.

### Submitting an Automated Machine Learning Experiment

You can submit an automated machine learning experiment like any other SDK-based experiment:

```
from azureml.core.experiment import Experiment

automl_experiment = Experiment(ws, 'automl_experiment')
automl_run = automl_experiment.submit(automl_config)
```

1 https://aka.ms/AA70rrw

## Monitoring and Reviewing Automated ML Runs

You can monitor automated machine learning experiment runs in Azure Machine Learning studio, or in the Jupyter Notebooks **RunDetails** widget.

### Retrieving the Best Run and its Model

You can easily identify the best run in Azure Machine Learning studio, and download or deploy the model it generated. To accomplish this with the SDK, you can use code like the following example:

```
best_run, fitted_model = automl_run.get_output()
best_run_metrics = best_run.get_metrics()
for metric_name in best_run_metrics:
    metric = best_run_metrics[metric_name]
    print(metric_name, metric)
```

### Exploring Preprocessing Steps

Automated machine learning uses scikit-learn pipelines to encapsulate preprocessing steps with the model. You can view the steps in the fitted model you obtained from the best run using the code above like this:

```
for step_ in fitted_model.named_steps:
    print(step)
```

## Lab: Use Automated Machine Learning from the SDK

**Note**: This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the Azure Machine Learning SDK to run an automated machine learning experiment.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Use automated machine learning from the SDK** exercise.

## Module Review

In this lesson, you learned how to use hyperparameter tuning and automated machine learning to train an optimal model.

Use the following review questions to check your learning.

## Question 1

*You want to try every possible combination of a set of specified discrete values in a hyperparameter tuning experiment.*
*Which kind of sampling should you use?*

- ☐ Grid Sampling
- ☐ Random Sampling
- ☐ Bayesian Sampling

## Question 2

*You want to use automated machine learning to find the model with the best AUC_weighted metric.*
*Which parameter of the AutoMLConfig object should you set?*

- ☐ task='AUC_weighted'
- ☐ label_column_name= 'AUC_weighted'
- ☐ primary_metric='AUC_weighted'

# Answers

**Question 1**

You want to try every possible combination of a set of specified discrete values in a hyperparameter tuning experiment.
Which kind of sampling should you use?

■ Grid Sampling

☐ Random Sampling

☐ Bayesian Sampling

*Explanation*
*You should use a Grid sampling to try every combination of discrete hyperparameter values.*

**Question 2**

You want to use automated machine learning to find the model with the best AUC_weighted metric.
Which parameter of the AutoMLConfig object should you set?

☐ task='AUC_weighted'

☐ label_column_name= 'AUC_weighted'

■ primary_metric='AUC_weighted'

*Explanation*
*To specify the target metric, use the primary_metric parameter.*

# Module 9 Responsible Machine Learning

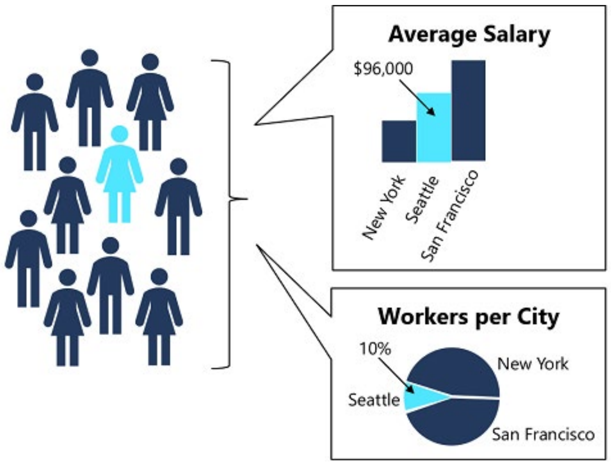
## Differential Privacy

### The Data Privacy Problem

Data science projects, including machine learning projects, involve analysis of data; and often that data includes sensitive personal details that should be kept private.
In practice, most reports that are published from the data include aggregations of the data, which you may think would provide some privacy – after all, the aggregated results do not reveal the individual data values.

However, consider a case where multiple analyses of the data result in reported aggregations that when combined, could be used to  work out information about individuals in the source dataset. Suppose 10 participants share data about their location and salary, from which two reports are produced:

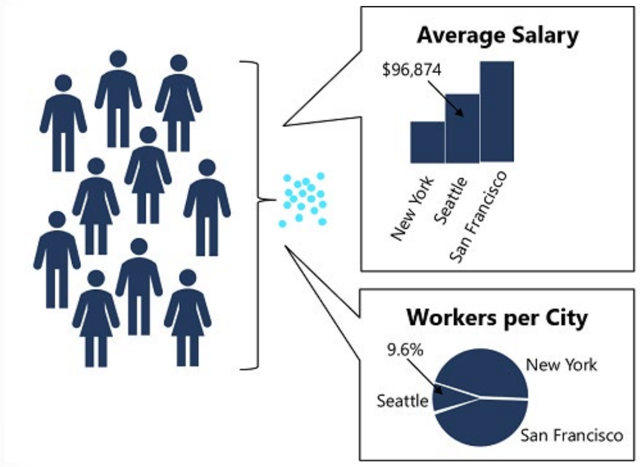
- An aggregated salary report that tells us the average salaries in New York, San Francisco, and Seattle
- A worker location report that tells us that 10% of the study participants (in other words, a single person) is based in Seattle.



From these two reports, we can easily determine the specific salary of the Seattle-based participant. Anyone reviewing both studies who happens to know a person from Seattle who participated, now knows that person's salary.

## What is Differential Privacy?

Differential privacy seeks to protect individual data values by adding statistical "noise" to the analysis process. The math involved in adding the noise is quite complex, but the principle is fairly intuitive – the noise ensures that data aggregations stay statistically consistent with the actual data values allowing for some random variation, but make it impossible to work out the individual values from the aggregated data. In addition, the noise is different for each analysis, so the results are non-deterministic – in other words, two analyses that perform the same aggregation may produce slightly different results.



## Epsilon - The Privacy Loss Parameter

One way that an individual can protect their personal data is simply to not participate in a study – this is known as their "opt-out" option. However, there are a few considerations for this as a solution:

- Even if you opt-out a study may still produce results that affect you. For example, you may choose to opt-out of a study that compares the heart disease diagnoses across a group of people on the basis that doing so may reveal a heart disease diagnosis that causes your health insurance premiums to rise. If the study finds a correlation between people who drink coffee and higher risk of heart disease, and your insurance company knows that you are a coffee drinker, your rate may rise even though you didn't personally participate in the study.
- The benefits of participation in the study may outweigh any negative impact. For example, if you're paid $100 to participate in a study that results in your health insurance rate rising by $10 per year, it will be more than ten years before you make a net loss. This may be a worthwhile tradeoff to you (particularly if your rate may rise as a result of the study even if you don't participate!)
- The only way for the opt-out option to work for every individual, is for every individual not to take part – which makes the whole study pretty pointless!

The amount of variation caused by adding noise is configurable through a parameter called epsilon. This value governs the amount of additional risk that your personal data can be identified through rejecting the opt-out option and participating in a study. The key thing is that it applies this privacy principle for everyone participating in the study. A low epsilon value provides the most privacy, at the expense of less accuracy when aggregating the data. A higher epsilon value results in aggregations that are more true to the actual data distribution, but in which the individual contribution of a single individual to the aggregated value is less obscured by noise.

## Lab: Explore Differential Privacy

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the **SmartNoise** package to explore differential privacy.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Explore differential privacy** exercise.

# Model Interpretability

## Model Interpretability in Azure Machine Learning

As machine learning becomes increasingly integral to decisions that affect health, safety, economic wellbeing, and other aspects of people's lives, it's important to be able to understand how models make predictions; and to be able to explain the rationale for machine learning based decisions while identifying and mitigating bias.

The range of machine learning algorithm types and the nature of how machine learning model training works make this a hard problem to solve, but model interpretability has become a key element of helping to make model predictions explainable.

### The *Interpret-Community* Package

Model interpretability in Azure Machine Learning is based on the open source **Interpret-Community** package, which is itself a wrapper around a collection of *explainers* based on proven and emerging model interpretation algorithms, such as **Shapely Additive Explanations (SHAP)**[1] and **Local Interpretable Model-agnostic Explanations (LIME)**[2]

**More Information**: For more information about the **Interpret-Community** package, see the project GitHub repository at **https://github.com/interpretml/interpret-community/**. For details about its implementation in Azure Machine Learning, see **Model interpretability in Azure Machine Learning**[3] in the Azure Machine Learning documentation.

## Global and Local Feature Importance

Model explainers use statistical techniques to calculate *feature importance*. This enables you to quantify the relative influence each feature in the training dataset has on label prediction. Explainers work by evaluating a test data set of feature cases and the labels the model predicts for them.

### Global Feature Importance

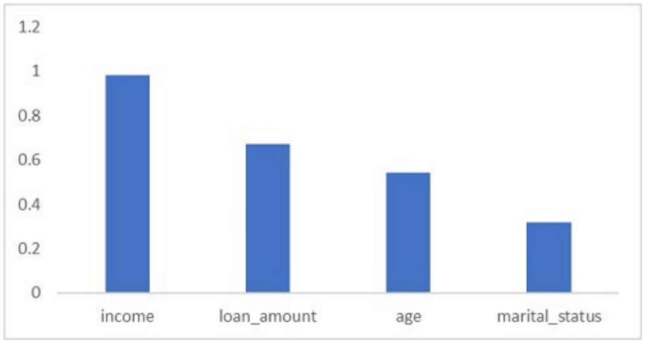
Global feature importance quantifies the relative importance of each feature in the test dataset as a whole. It provides a general comparison of the extent to which each feature in the dataset influences prediction.

For example, a binary classification model to predict loan default risk might be trained from features such as loan amount, income, marital status, and age to predict a label of 1 for loans that are likely to be repaid, and 0 for loans that have a significant risk of default (and therefore shouldn't be approved). An explainer might then use a sufficiently representative test dataset to produce the following global feature importance values:

- **income**: 0.98
- **loan amount**: 0.67
- **age**: 0.54

1 https://github.com/slundberg/shap
2 https://github.com/marcotcr/lime
3 https://aka.ms/AA70rs0

- **marital status** 0.32



It's clear from these values, that in respect to the overall predictions generated by the model for the test dataset, **income** is the most important feature for predicting whether or not a borrower will default on a loan, followed by the **loan amount**, then **age**, and finally **marital status**.

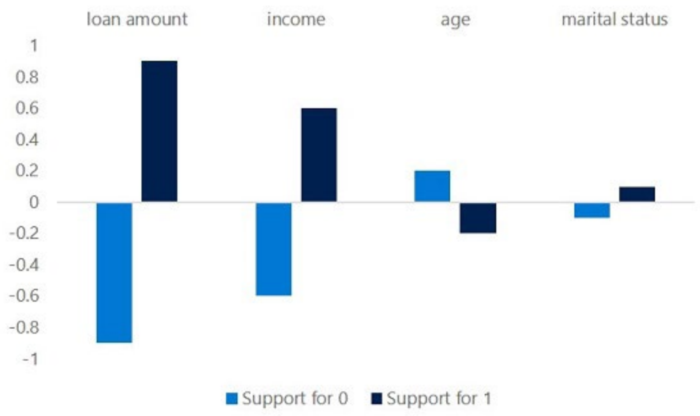
## Local Feature Importance

Local feature importance measures the influence of each feature value for a specific individual prediction.

For example, suppose Sam applies for a loan, which the machine learning model approves (by predicting that Sam won't default on the loan repayment). You could use an explainer to calculate the local feature importance for Sam's application to determine which factors influenced the prediction. You might get a result like this:

| Feature | Support for 0 | Support for 1 |
|---|---|---|
| loan amount | -0.9 | 0.9 |
| income | -0.6 | 0.6 |
| age | 0.2 | -0.2 |
| marital status | -0.1 | 0.1 |

Because this is a *classification* model, each feature gets a local importance value for each possible class, indicating the amount of support for that class based on the feature value. Since this is a *binary* classification model, there are only two possible classes (0 and 1). Each feature's support for one class results in correlatively negative level of support for the other.



In Sam's case, the overall support for class 0 is -1.4, and the support for class 1 is correspondingly 1.4; so support for class 1 is obviously higher than for class 0, and the loan is approved. Note that the most important feature for a prediction of class 1 is **loan amount**, followed by **income** - these are the opposite order from their global feature importance values (which indicate that **income** is the most important factor for the data sample as a whole). There could be multiple reasons why local importance for an individual prediction varies from global importance for the overall dataset; for example,  Sam might have a lower income than average, but the loan amount in this case might be unusually small.

For a multi-class classification model, a local importance values for each possible class is calculated for every feature, with the total across all classes always being 0. For example, a model might predict the species of an iris flower based on features like its petal length, petal width, sepal length, and sepal width. There are three species of iris, so the model predicts one of three class labels (0, 1, or 2) For an individual prediction, the petal length feature might have local importance values of 0.5 for class 0, 0.3 for class 1, and -0.8 for class 2 - indicating that the petal length moderately supports a prediction of class 0, slightly supports a prediction of class 1, and strongly supports a prediction that this particular iris is ***not*** class 2.

For a regression model, there are no classes so the local importance values simply indicate the level of influence each feature has on the predicted scalar label.

## Explainers

You can use the Azure Machine Learning SDK to create explainers for models, even if they were not trained using an Azure Machine Learning experiment.

### Creating an Explainer

To interpret a model, you must install the **azureml-interpret** package and use it to create an explainer. There are multiple types of explainer, including:

- **MimicExplainer** - An explainer that creates a *global surrogate model* that approximates your trained model and can be used to generate explanations. This explainable model must have the same kind of architecture as your trained model (for example, linear or tree-based).
- **TabularExplainer** - An explainer that acts as a wrapper around various SHAP explainer algorithms, automatically choosing the one that is most appropriate for your model architecture.
- **PFIExplainer** - a *Permutation Feature Importance* explainer that analyzes feature importance by shuffling feature values and measuring the impact on prediction performance.

The following code sample shows how to create an instance of each of these explainer types:

```
# MimicExplainer
from interpret.ext.blackbox import MimicExplainer
from interpret.ext.glassbox import DecisionTreeExplainableModel

mim_explainer = MimicExplainer(model=loan_model,
                             initialization_examples=X_test,
                             explainable_model = DecisionTreeExplainable-
Model,
                             features=['loan_amount','income','age','mari-
tal_status'],
                             classes=['reject', 'approve'])


# TabularExplainer
from interpret.ext.blackbox import TabularExplainer
```

```
tab_explainer = TabularExplainer(model=loan_model,
                             initialization_examples=X_test,
                             features=['loan_amount','income','age','mari-
tal_status'],
                             classes=['reject', 'approve'])


# PFIExplainer
from interpret.ext.blackbox import PFIExplainer

pfi_explainer = PFIExplainer(model = loan_model,
                             features=['loan_amount','income','age','mari-
tal_status'],
                             classes=['reject', 'approve'])
```

## Explaining Global Feature Importance

To retrieve global importance values for the features in your model, you call the **explain_global()** method of your explainer to get a global explanation, and then use the **get_feature_importance_dict()** method to get a dictionary of the feature importance values:

```
# MimicExplainer
global_mim_explanation = mim_explainer.explain_global(X_train)
global_mim_feature_importance = global_mim_explanation.get_feature_impor-
tance_dict()


# TabularExplainer
global_tab_explanation = tab_explainer.explain_global(X_train)
global_tab_feature_importance = global_tab_explanation.get_feature_impor-
tance_dict()


# PFIExplainer
global_pfi_explanation = pfi_explainer.explain_global(X_train, y_train)
global_pfi_feature_importance = global_pfi_explanation.get_feature_impor-
tance_dict()
```

**Note**: The code is the same for **MimicExplainer** and **TabularExplainer**. The **PFIExplainer** requires the actual labels that correspond to the test features.

## Explaining Local Feature Importance

To retrieve local feature importance from a **MimicExplainer** or a **TabularExplainer**, you must call the **explain_local()** method of your explainer, specifying the subset of cases you want to explain. Then you can use the **get_ranked_local_names()** and **get_ranked_local_values()** methods to retrieve dictionaries of the feature names and importance values, ranked by importance.

```
# MimicExplainer
local_mim_explanation = mim_explainer.explain_local(X_test[0:5])
local_mim_features = local_mim_explanation.get_ranked_local_names()
```

```
local_mim_importance = local_mim_explanation.get_ranked_local_values()


# TabularExplainer
local_tab_explanation = tab_explainer.explain_local(X_test[0:5])
local_tab_features = local_tab_explanation.get_ranked_local_names()
local_tab_importance = local_tab_explanation.get_ranked_local_values()
```

**Note**: The code is the same for **MimicExplainer** and **TabularExplainer**. The **PFIExplainer** doesn't support local feature importance explanations.

## Adding Explanations to Training Experiments

When you use an estimator or a script to train a model in an Azure Machine Learning experiment, you can create an explainer and upload the explanation it generates to the run output for later analysis.

### Creating an Explanation in the Experiment Script

To create an explanation in the experiment script, you'll need to ensure that the **azureml-interpret** package is installed in the run environment. Then you can use these to create an explanation from your trained model and upload it to the run outputs.

```
# Import Azure ML run library
from azureml.core.run import Run
from azureml.interpret.explanation.explanation_client import Explanation-
Client
from interpret.ext.blackbox import TabularExplainer
# other imports as required

# Get the experiment run context
run = Run.get_context()

# code to train model goes here

# Get explanation
explainer = TabularExplainer(model, X_train, features=features, classes=la-
bels)
explanation = explainer.explain_global(X_test)

# Get an Explanation Client and upload the explanation
explain_client = ExplanationClient.from_run(run)
explain_client.upload_model_explanation(explanation, comment='Tabular
Explanation')

# Complete the run
run.complete()
```

## Viewing the Explanation

You can view the explanation you created for your model in the **Explanations** tab for the run in Azure Machine learning studio.

You can also use the **ExplanationClient** object to download the explanation in Python.

```
from azureml.interpret.explanation.explanation_client import Explanation-
Client

client = ExplanationClient.from_run_id(workspace=ws,
                                       experiment_name=experiment.experi-
ment_name,
                                       run_id=run.id)
explanation = client.download_model_explanation()
feature_importances = explanation.get_feature_importance_dict()
```

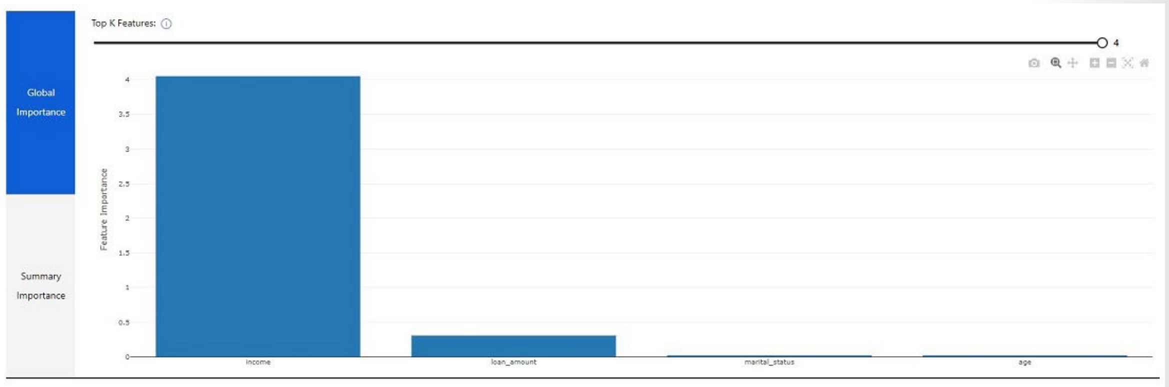
# Visualizing Model Explanations

Model explanations in Azure Machine Learning studio include multiple visualizations that you can use to explore feature importance.

**Note** Visualizations are only available for runs that were configured to generate explanations. When using automated machine learning, only the run producing the best model has explanations generated by default.

## Visualizing Global Feature importance

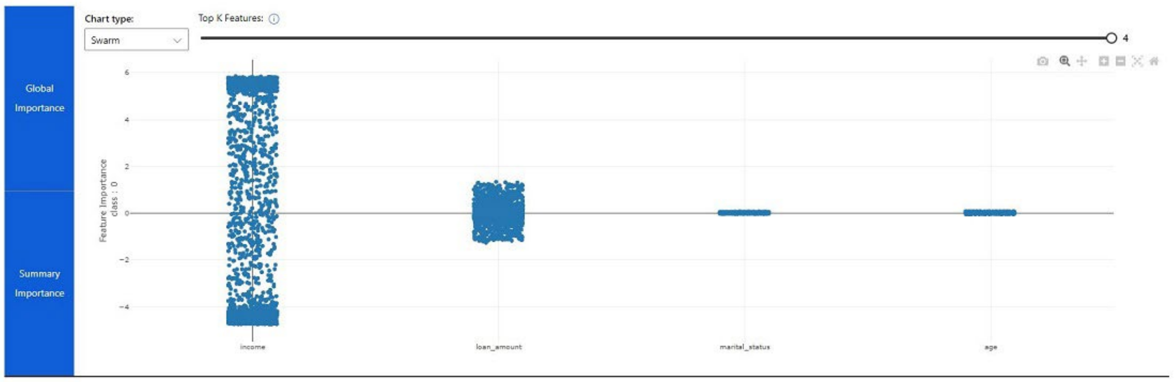
The first visualization on the **Explanations** tab for a run shows global feature importance.



You can use the slider to show only the top *N* features.

## Visualizing Summary Importance

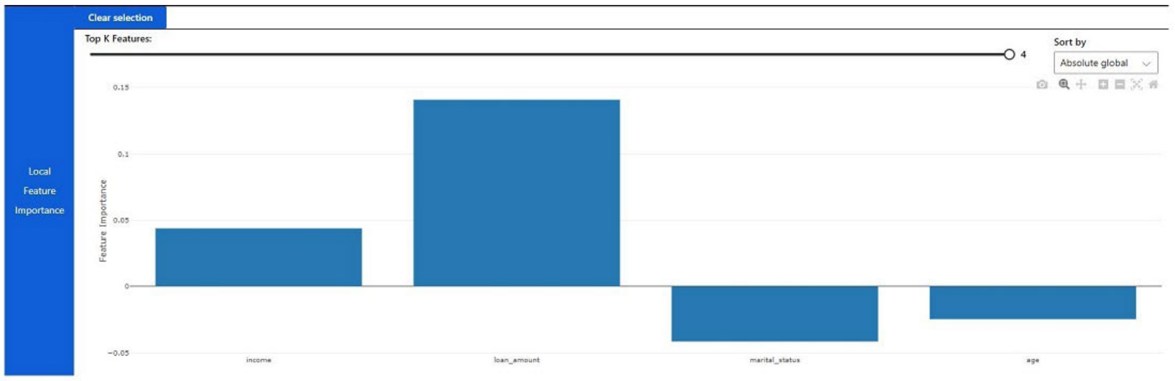
Switching to the **Summary Importance** visualization shows the distribution of individual importance values for each feature across the test dataset.



You can view the features as a *swarm* plot (shown above), a*box* plot, or a *violin* plot.

## Visualizing Local Feature Importance

Selecting an individual data point shows the local feature importance for the case to which the data point belongs.



## Interpretability During Inferencing

In some scenarios, you might want to generate explanations along with predictions from a published model.

## Register a Scoring Explainer with the Model

The first step in this process is to create a *scoring explainer* as a wrapper for your model explainer, and register it in the same workspace as your model.

```
from interpret.ext.blackbox import TabularExplainer
from azureml.interpret.scoring.scoring_explainer import KernelScoringEx-
plainer, save
from azureml.core import Model

# Get a registered model
loan_model - ws.models['loan_model']

tab_explainer = TabularExplainer(model = loan_model,
                              initialization_examples=X_test,
                              features=['loan_amount','income','age','mari-
tal_status'],
                              classes=['reject', 'approve'])

# Create and save a scoring explainer
scoring_explainer = KernelScoringExplainer(tab_explainer, X_test[0:100])
save(scoring_explainer, directory='explainer', exist_ok=True)

# Register the explainer (like a model)
Model.register(ws, model_name='loan_explainer', 'explainer/scoring_explain-
er.pkl')
```

## Create a Scoring Script to Include Explanations

After registering the explainer, you can create a scoring script for a real-time service that loads the explainer and uses it to return explanations along with predictions.

```
import json
import joblib
from azureml.core.model import Model

# Called when the service is loaded
def init():
    global model, explainer
    # load the model
    model_path = Model.get_model_path('loan_model')
    model = joblib.load(model_path)
    # load the explainer
    explainer_path = Model.get_model_path('loan_explainer')
    explainer = joblib.load(explainer_path)

# Called when a request is received
def run(raw_data):
    # Get the input data
    data = np.array(json.loads(raw_data)['data'])
    # Get a prediction from the model
    predictions = model.predict(data)
    # Get explanations
    importance_values = explainer.explain(data)
    # Return the predictions and explanations as JSON
    return {"predictions":predictions.tolist(),"importance":importance_val-
ues}
```

## Deploy the Inferencing Service

With the scoring script created, you can deploy the service - referencing both the predictive model and the explainer.

```
service = Model.deploy(ws, 'loan-svc', [model, explainer], inf_config, dep_
config)
```

## Retrieving Predictions and Explanations

When you consume the service, the JSON returned includes both the predictions and the associated local feature importance values:

```
import json

# New loan application data
x_new = [[55000, 3500, 37, 1]]
json_data = json.dumps({"data": x_new})
response = service.run(input_data = json_data)
print(response)
```

```
{
  'predictions': [1],
  'local_importance': [[[-0.12, -0.15, -0.03, 0.11]],
                       [[0.12, 0.15, 0.03, -0.11]]]
}
```

# Lab: Interpret Models

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the **azureml-interpret** package to create explainers for models.

## Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Interpret models** exercise.

## Fairness

### What is Fairness?

Machine learning models are increasingly used to inform decisions that affect peoples lives. For example, prediction made by a machine learning model might influence:

- Approval for a loan, insurance, or other financial service.
- Acceptance into a school or college course.
- Eligibility for a medical trial or experimental treatment.
- Inclusion in a marketing promotion.
- Selection for employment or promotion.

With such critical decisions in the balance, it's important to have confidence that the machine learning models we rely on predict fairly, and don't result in a negative impact on groups based on:

- Ethnicity
- Gender
- Age
- Physical disability
- other sensitive features

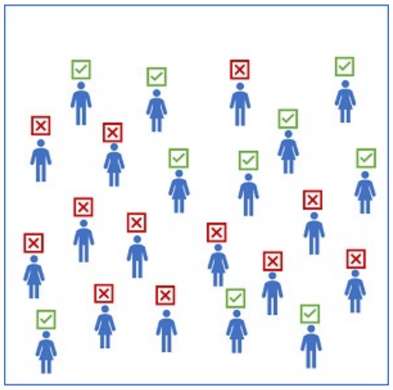
### Evaluating Model Fairness

When we consider the concept of *fairness* in relation to predictions made by machine learning models, it helps to be clear about what we mean by “fair”.

For example, suppose a classification model is used to predict the probability of a successful loan repayment, and therefore influences whether or not the loan is approved. It's likely that the model will be trained using features that reflect characteristics of the applicant, such as:

- Age
- Employment status
- Income
- Savings
- Current debt

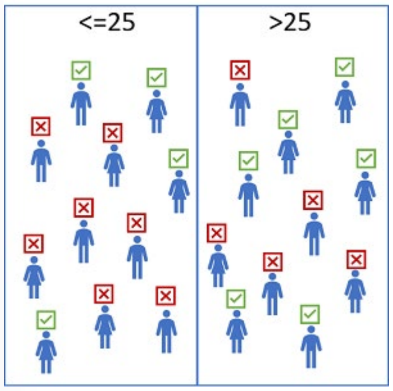
These features are used to train a binary classification model that predicts whether an applicant will repay a loan.



Suppose the model predicts that around 45% of applicants will successfully repay their loans. However, on reviewing loan approval records, you begin to suspect that that fewer loans are approved for applicants aged 25 or younger than for applicants who are over 25. How can you be sure the model is *fair* to applicants in both age groups?

## Measuring disparity in predictions

One way to start evaluating the fairness of a model is to compare *predictions* for each group within a *sensitive feature*. For the loan approval model, *Age* is a sensitive feature that we care about, so we could split the data into subsets for each age group and compare the *selection rate* (the proportion of positive predictions) for each group.



Let's say we find that the model predicts that 36% of applicants aged 25 or younger will repay a loan, but it predicts successful repayments for 54% of applicants aged over 25. There's a disparity in predictions of 18%.

At first glance, this comparison seems to confirm that there's bias in the model that discriminates against younger applicants. However, when you consider the population as a whole, it may be that younger people generally earn less than people more established in their careers, have lower levels of savings and assets, and have a higher rate of defaulting on loans.

The important point to consider here is that just because we want to ensure fairness in regard to age, it doesn't necessarily follow that age is <u>not</u> a factor in loan repayment probability. It's possible that in general, younger people really are less likely to repay a loan than older people. To get the full picture, we need to look a little deeper into the predictive performance of the model for each subset of the population.

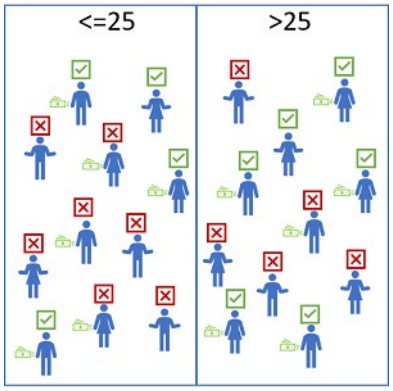
## Measuring disparity in prediction performance

When you train a machine learning model using a supervised technique, like regression or classification, you use metrics achieved against hold-out validation data to evaluate the overall predictive performance of the model. For example, you might evaluate a classification model based on *accuracy*, *precision*, or *recall*.

To evaluate the fairness of a model, you can apply the same predictive performance metric to subsets of the data, based on the sensitive features on which your population is grouped, and measure the disparity in those metrics across the subgroups.

For example, suppose the loan approval model exhibits an overall *recall* metric of 0.67 - in other words, it correctly identifies 67% of cases where the applicant repaid the loan. The question is whether or not the model provides a similar rate of correct predictions for different age groups.

To find out, we group the data based on the sensitive feature (*Age*) and measure the predictive performance metric (*recall*) for those groups. Then we can compare the metric scores to determine the disparity between them.



Let's say that we find that the recall for validation cases where the applicant is 25 or younger is 0.50, and recall for cases where the applicant is over 25 is 0.83. In other words, the model correctly identified 50% of the people in the 25 or younger age group who successfully repaid a loan (and therefore misclassified 50% of them as loan defaulters), but found 83% of loan repayers in the older age group (misclassifying only 17% of them). The disparity in prediction performance between the groups is 33%, with the model predicting significantly more false negatives for the younger age group.

## Mitigating Unfairness

A common approach to mitigation is to use one of the algorithms and constraints to train multiple models, and then compare their performance, selection rate, and disparity metrics to find the optimal model for your needs. Often, the choice of model involves a trade-off between raw predictive performance and fairness - based on your definition of fairness for a given scenario. Generally, fairness is measured by reduction in disparity of feature selection (for example, ensuring that an equal proportion of members from each gender group is approved for a bank loan) or by a reduction in disparity of performance metric (for example, ensuring that a model is equally accurate at identifying repayers and defaulters in each age group).

To train the models for comparison, you use mitigation algorithms to create alternative models that apply *parity constraints*  to produce comparable metrics across sensitive feature groups. The following table describes some common algorithms used to optimize models for fairness.

| Technique | Description | Model type support |
|---|---|---|
| **Exponentiated Gradient** | A *reduction* technique that applies a cost-minimization approach to learning the optimal trade-off of overall predictive performance and fairness disparity | Binary classification and regression |
| **Grid Search** | A simplified version of the Exponentiated Gradient algorithm that works efficiently with small numbers of constraints | Binary classification and regression |
| **Threshold Optimizer** | A *post-processing* technique that applies a constraint to an existing classifier, transforming the prediction as appropriate | Binary classification |

The choice of parity constraint depends on the technique being used and the specific fairness criteria you want to apply. Constraints include:

- **Demographic parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in the selection rate across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that an equal number of positive predictions are made in each group.
- **True positive rate parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in *true positive rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive predictions.
- **False positive rate parity**: Use this constraint with any of the mitigation algorithms to minimize disparity in *false positive rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of false positive predictions.
- **Equalized odds**: Use this constraint with any of the mitigation algorithms to minimize disparity in combined *true positive rate* and *false positive rate* across sensitive feature groups. For example, in a binary classification scenario, this constraint tries to ensure that each group contains a comparable ratio of true positive and false positive predictions.
- **Error rate parity**: Use this constraint with any of the reduction-based mitigation algorithms (**Exponentiated Gradient** and **Grid Search**) to ensure that the error for each sensitive feature group does not deviate from the overall error rate by more than a specified amount.
- **Bounded group loss**: Use this constraint with any of the reduction-based mitigation algorithms to restrict the loss for each sensitive feature group in a *regression* model.

## Lab: Detect and Mitigate Unfairness

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use the **Fairlearn** package to evaluate models for fairness.

## Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Detect and mitigate unfairness** exercise.

# Module Review

In this module, you learned how to apply some principles of responsible machine learning.

Use the following review questions to check your learning.

## Question 1

*In a differential privacy solution, what is the effect of setting an epsilon parameter?*

☐ A lower epsilon reduces the impact of an individual's data on aggregated results, increasing privacy and decreasing accuracy

☐ A lower epsilon reduces the amount of noise added to the data, increasing accuracy and decreasing privacy

## Question 2

*You have trained a model, and you want to quantify the influence of each feature on a specific individual prediction.*
*What kind of feature importance should you examine?*

☐ Global feature importance

☐ Local feature importance

## Question 3

*You are training a binary classification model to support admission approval decisions for a college degree program.*
*How can you evaluate if the model is fair, and doesn't discriminate based on ethnicity?*

☐ Evaluate each trained model with a validation dataset and use the model with the highest accuracy score.

☐ Remove the ethnicity feature from the training dataset.

☐ Compare disparity between selection rates and performance metrics across ethnicities.

# Answers

**Question 1**

In a differential privacy solution, what is the effect of setting an epsilon parameter?

■ A lower epsilon reduces the impact of an individual's data on aggregated results, increasing privacy and decreasing accuracy

☐ A lower epsilon reduces the amount of noise added to the data, increasing accuracy and decreasing privacy

*Explanation*
*In a differential privacy solution, noise is added to the data when generating analyses so that aggregations are statistically consistent but non-deterministic; and individual contributions to the aggregations cannot be determined.*

**Question 2**

You have trained a model, and you want to quantify the influence of each feature on a specific individual prediction.
What kind of feature importance should you examine?

☐ Global feature importance

■ Local feature importance

*Explanation*
*Local importance indicates the influence of features on a specific prediction. Global importance gives an overall indication of feature influence.*

**Question 3**

You are training a binary classification model to support admission approval decisions for a college degree program.
How can you evaluate if the model is fair, and doesn't discriminate based on ethnicity?

☐ Evaluate each trained model with a validation dataset and use the model with the highest accuracy score.

☐ Remove the ethnicity feature from the training dataset.

■ Compare disparity between selection rates and performance metrics across ethnicities.

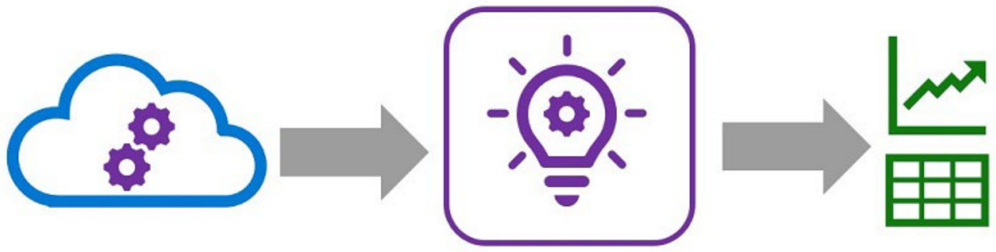
*Explanation*
*By using ethnicity as a sensitive field, and comparing disparity between selection rates and performance metrics for each ethnicity value, you can evaluate the fairness of the model.*

# Module 10 Monitoring Models

## Monitoring Models with Application Insights

## What is Application Insights?

Application Insights is an application performance management service in Microsoft Azure that enables the capture, storage, and analysis of telemetry data from applications.



You can use Application Insights to monitor telemetry from many kinds of application, including applications that are not running in Azure. All that's required is a low-overhead instrumentation package to capture and send the telemetry data to Application Insights. The necessary package is already included in Azure Machine Learning Web services.

## Enabling Application Insights

To log telemetry in application insights from an Azure machine learning service, you must have an Application Insights resource associated with your Azure Machine Learning workspace, and you must configure your service to use it for telemetry logging.

### Associating Application Insights with a Workspace

When you create an Azure Machine Learning workspace, you can select an Azure Application Insights resource to associate with it. If you do not select an existing Application Insights resource, a new one is created in the same resource group as your workspace.

You can determine the Application Insights resource associated with your workspace by viewing the **Overview** page of the workspace blade in the Azure portal, or by using the **get_details()** method of a **Workspace** object:

```
from azureml.core import Workspace

ws = Workspace.from_config()
ws.get_details()['applicationInsights']
```

### Enabling Application Insights for a Service

When deploying a new real-time service, you can enable Application Insights in the deployment configuration for the service:

```
dep_config = AciWebservice.deploy_configuration(cpu_cores = 1,
                                                memory_gb = 1,
                                                enable_app_insights=True)
```

If you want to enable Application Insights for a service that is already deployed, you can modify the deployment configuration for Azure Kubernetes Service (AKS) based services in the Azure portal, or you can update any web service by using the Azure Machine Learning SDK:

```
service = ws.webservices['my-svc']
service.update(enable_app_insights=True)
```

## Capturing and Viewing Application Insights Data

Application Insights automatically captures any information written to the standard output and error logs, and provides a query capability to view data in these logs.

### Writing Log Data

To capture telemetry data for Application insights, you can write any values to the standard output log in the scoring script for your service by using a `print` statement:

```
def init():
    global model
    model = joblib.load(Model.get_model_path('my_model'))
def run(raw_data):
    data = json.loads(raw_data)['data']
    predictions = model.predict(data)
    log_txt = 'Data:' + str(data) + ' - Predictions:' + str(predictions)
    print(log_txt)
    return predictions.tolist()
```

Azure Machine Learning creates a *custom dimension* in the Application Insights data model for the output you write.

## Querying Logs in Application Insights

To analyze captured log data, you can use the Log Analytics query interface for Application Insights in the Azure portal. This interface supports a SQL-like query syntax that you can use to extract fields from logged data, including custom dimensions created by your Azure Machine Learning service.

For example, the following query returns the **timestamp** and **customDimensions.Content** fields from log traces that have a **message** field value of *STDOUT* (indicating the data is in the standard output log) and a **customDimensions.["Service Name"]** field value of *my-svc*:

```
traces
|where message == "STDOUT"
  and customDimensions.["Service Name"] = "my-svc"
| project  timestamp, customDimensions.Content
```

This query returns the logged data as a table:

| timestamp | customDimensions_Content |
|---|---|
| 01/02/2020... | Data:[[1, 2, 2.5, 3.1], [0, 1, 1,7, 2.1]] - Predictions:[0 1] |
| 01/02/2020... | Data:[[3, 2, 1.7, 2.0]] - Predictions:[0] |

**More Information**: For more information about using the Application Insights Log  Analytics query interface, see **Overview of log queries in Azure Monitor**[1] in the Azure Monitor documentation.

# Lab: Monitor a Model

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will use Application Insights to monitor a deployed model.

## Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Monitor a model** exercise.

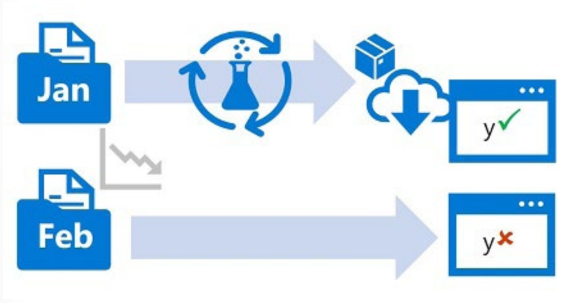
1 https://aka.ms/AA70rs1

# Monitoring Data Drift

## What is Data Drift?

You typically train a model using a historical dataset that is representative of the new data that your model will receive for inferencing. However, over time there may be trends that change the profile of the data, making your model less accurate.

For example, suppose a model is trained to predict the expected gas mileage of an automobile based on the number of cylinders, engine size, weight, and other features. Over time, as car manufacturing and engine technologies advance, the typical fuel-efficiency of vehicles might improve dramatically; making the predictions made by the model trained on older data less accurate.



This change in data profiles between training and inferencing is known as *data drift*, and it can be a significant issue for predictive models used in production. It is therefore important to be able to monitor data drift over time, and retrain models as required to maintain predictive accuracy.

## Creating a Data Drift Monitor

Azure Machine Learning supports data drift monitoring through the use of *datasets*. You can capture new feature data in a dataset and compare it to the dataset with which the model was trained.

### Monitoring Data Drift by Comparing Datasets

It's common for organizations to continue to collect new data after a model has been trained. For example, a health clinic might use diagnostic measurements from previous patients to train a model that predicts diabetes likelihood, but continue to collect the same diagnostic measurements from all new patients. The clinic's data scientists could then periodically compare the growing collection of new data to the original training  data, and identify any changing data trends that might affect model accuracy.

To monitor data drift using registered datasets, you need to register two datasets:

- A *baseline* dataset - usually the original training data.
- A *target* dataset that will be compared to the baseline based on time intervals. This dataset requires a column for each feature you want to compare, and a timestamp column so the rate of data drift can be measured.

[!NOTE]
You can configure a deployed service to collect new data submitted to the model for inferencing, which is saved in Azure blob storage and can be used as a target dataset for data drift monitoring. See **Collect data from models in production**[2] in the Azure Machine Learning documentation for more information.

2 https://aka.ms/AA70zg8

After creating these datasets, you can define a *dataset monitor* to detect data drift and trigger alerts if the rate of drift exceeds a specified threshold. You can create dataset monitors using the visual interface in Azure Machine Learning studio, or by using the **DataDriftDetector** class in the Azure Machine Learning SDK as shown in the following example code:

```
from azureml.datadrift import DataDriftDetector

monitor = DataDriftDetector.create_from_datasets(workspace=ws,
                                                 name='dataset-drift-detec-
tor',
                                                 baseline_data_set=train_
ds,
                                                 target_data_set=new_data_
ds,
                                                 compute_target='aml-clus-
ter',
                                                 frequency='Week',
                                                 feature_
list=['age','height', 'bmi'],
                                                 latency=24)
```

After creating the dataset monitor, you can *backfill* to immediately compare the baseline dataset to existing data in the target dataset, as shown in the following example, which backfills the monitor based on weekly changes in data for the previous six weeks:

```
import datetime as dt

backfill = monitor.backfill( dt.datetime.now() - dt.timedelta(weeks=6), dt.
datetime.now())
```

## Data Drift Schedules and Alerts

When you define a data monitor, you specify a schedule on which it should run. Additionally, you can specify a threshold for the rate of data drift and an operator email address for notifications if this threshold is exceeded.

### Configuring Data Drift Monitor Schedules

Data drift monitoring works by running a comparison at scheduled **frequency**, and calculating data drift metrics for the features in the dataset that you want to monitor. You can define a schedule to run every **Day**, **Week**, or **Month**.

For dataset monitors, you can specify a **latency**, indicating the number of hours to allow for new data to be collected and added to the target dataset. For deployed model data drift monitors, you can specify a **schedule_start** time value to indicate when the data drift run should start (if omitted, the run will start at the current time).

### Configuring Alerts

Data drift is measured using a calculated *magnitude* of change in the statistical distribution of feature values over time. You can expect some natural random variation between the baseline and target datasets, but you should monitor for large changes that might indicate significant data drift.

You can define a **threshold** for data drift magnitude above which you want to be notified, and configure alert notifications by email.

```
alert_email AlertConfiguration('data_scientists@contoso.com')
monitor = DataDriftDetector.create_from_datasets(ws, 'dataset-drift-detec-
tor',
                                                  baseline_data_set, target_
data_set,
                                                  compute_target=cpu_clus-
ter,
                                                  frequency='Week', laten-
cy=2,
                                                  drift_threshold=.3,
                                                  alert_configuration=alert_
email)
```

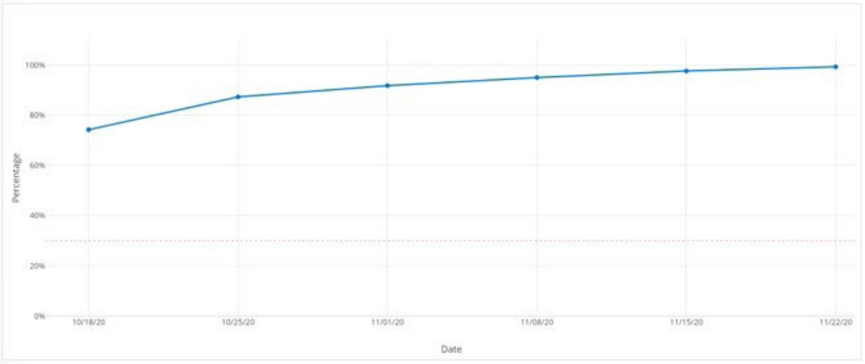
## Reviewing Data Drift

You can view the data drift metrics your monitor has collected in Azure Machine Learning studio.

- To see data drift for datasets, view the **Dataset monitors** tab of the **Datasets** page.
- To see data drift for a deployed model, open the model on the **Models** page and view its **Data drift** tab.

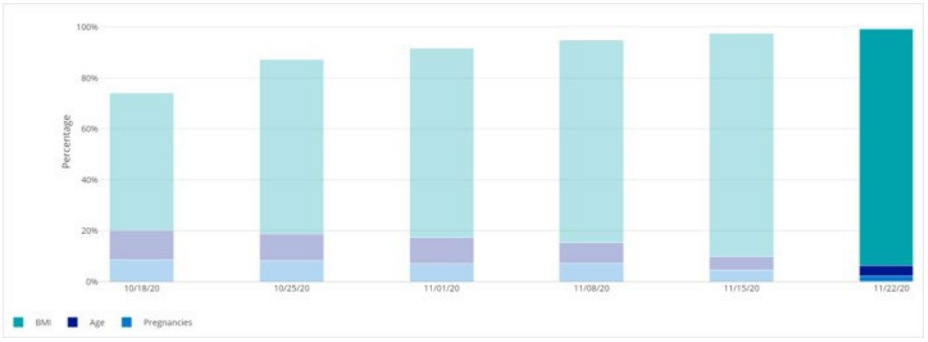
### Data Drift Visualizations

Data drift visualizations for datasets include a line chart showing the overall data drift magnitude trend.



.

Additionally, you can also see details of data drift for individual features.



.

## Lab: Monitor Data Drift

**Note**:  This lab requires an Azure subscription. If you have not already done so, redeem your Azure Pass code to get set up with an Azure subscription.

In this lab, you will monitor a dataset to detect data drift.

### Instructions

You can complete this lab on your own computer, or a hosted lab environment may be available in your class - check with your instructor.

1. Open the lab instructions at **https://aka.ms/mslearn-dp100**.
2. If you have not already done so, complete the **Create an Azure Machine Learning Workspace** exercise.
3. Complete the **Monitor data drift** exercise.

## Module Review

In this module, you learned how to monitor deployed services and detect data drift.

Use the following review questions to check your learning.

### Question 1

*You want to include custom information in the telemetry for your inferencing service, and analyze it using Application Insights.*
*What must you do in your service's entry script?*

☐ Use the Run.log method to log the custom metrics.

☐ Save the custom metrics in the ./outputs folder.

☐ Use a print statement to write the metrics in the STDOUT log.

## Question 2

*You previously trained a model using a training dataset. You want to detect any data drift in the new data collected since the model was trained.*
*What should you do?*

- ☐ Create a new version of the dataset using only the new data; and retrain the model.
- ☐ Add the new data to the existing dataset and enable Application Insights for the service where the model is deployed.
- ☐ Create a new dataset using the new data and a timestamp column; and create a data drift monitor that uses the training dataset as a baseline and the new dataset as a target.

# Answers

**Question 1**

You want to include custom information in the telemetry for your inferencing service, and analyze it using Application Insights.
What must you do in your service's entry script?

- ☐ Use the Run.log method to log the custom metrics.
- ☐ Save the custom metrics in the ./outputs folder.
- ■ Use a print statement to write the metrics in the STDOUT log.

*Explanation*
*To include custom metrics, add print statements to the scoring script so that the custom information is written to the STDOUT log.*

**Question 2**

You previously trained a model using a training dataset. You want to detect any data drift in the new data collected since the model was trained.
What should you do?

- ☐ Create a new version of the dataset using only the new data; and retrain the model.
- ☐ Add the new data to the existing dataset and enable Application Insights for the service where the model is deployed.
- ■ Create a new dataset using the new data and a timestamp column; and create a data drift monitor that uses the training dataset as a baseline and the new dataset as a target.

*Explanation*
*To track changing data trends, create a data drift monitor that uses the training data as a baseline and the new data as a target.*