Herramientas para la Computación en la Nube Dirigida a Inteligencia Artificial

# Tema 3. Creación de modelos de IA en Microsoft Azure con Machine Learning Workspaces

# Índice

# Esquema

## CREACIÓN DE MODELOS DE IA EN MICROSOFT AZURE CON MACHINE LEARNING WORKSPACES

### ENTORNO DE TRABAJO

- **Despliegue del entorno**
  - Azure CLI
  - PowerShell
  - REST API
  - Portal de Azure
- **Elementos del entorno**
  - Roles de usuario
  - Experimentos
  - Datos y fuentes de datos
  - Infraestructura de computación
  - Entornos de entrenamiento
  - Ejecuciones
  - Modelos registrados
  - Despliegues
- **Machine Learning Studio**

### ENTRENAMIENTO DE MODELOS

- **Ingestión de datos**
  - Manual
  - Automatizado
- **Exploración de datos**
  - pandas, matplotlib, seaborn
- **Ingeniería de variables**
  - Etiquetado de imágenes y textos
- **Uso de *pipelines***
  - Ejemplo con PythonScriptStep
- **Entrenamiento de modelos**
  - Infraestructura de ejecución
  - Registro de experimentos/ejecuciones
- **Entrenamiento de redes neuronales**
  - Ejemplo de uso de GPUs
- **Ajuste de hiperparámetros**
  - Ejemplo con HyperDrive

### VALOR AÑADIDO

- **AutoML**
  - Ejemplo con Python SDK
- ***Deep learning* distribuido**
  - Ejemplo con Horovod/MPI

### PRODUCTIZACIÓN

- **Despliegue de modelos**
  - Ejemplo con Python SDK
- **Operación de modelos**
  - Perfilado
  - Generación de trazas
  - *Concept drift*

## 3.1. Introducción y objetivos

En este tema, se describen las funcionalidades que proporciona la plataforma de *machine learning* e IA de Microsoft, **Azure Machine Learning.** Esta plataforma puede utilizarse a través del servicio web Azure Machine Learning Studio. Como paso previo, debe crearse un **entorno de trabajo**, o *workspace,* que albergue todos los recursos necesarios para facilitar la creación de modelos. Aquí se describirán los **elementos** que confirman dicho entorno y los pasos necesarios para ponerlo en marcha.

A lo largo del tema, se describen las **funcionalidades** que proporciona Azure Machine Learning para cada una de las fases del proceso de creación de modelos: ingestión de datos, análisis exploratorio, ingeniería de variables, uso de *pipelines* para entrenar modelos y ajuste de hiperparámetros.

En todos los casos, estas funcionalidades automatizan labores generalmente repetitivas o tediosas, o proporcionan un mecanismo para estructurar mejor el proceso y beneficiarse de las ventajas de la reutilización de código. Las explicaciones incluyen ejemplos de código fuente en lenguaje Python para ilustrar la forma específica en la que pueden utilizarse estas funcionalidades.

También se describen las **capacidades de AutoML,** o *machine learning* automatizado, que ofrece Azure, las posibilidades del *deep learning* en arquitecturas distribuidas, el despliegue automatizado de modelos en servicios y las opciones de monitorización de dichos despliegues. Estas características representan el elemento diferencial de un entorno *cloud,* como Azure, frente a la modalidad *on premise.*

# Ideas clave

Por último, este tema se plantea los siguientes objetivos para el alumno:

- Utilizar eficazmente el entorno que ofrece Microsoft Azure para crear modelos de *machine learning* e IA.
- Explotar las funcionalidades diferenciales que ofrece Azure Machine Learning para agilizar el proceso.
- Utilizar el SDK de Azure Machine Learning para Python como medio para adoptar un enfoque de reutilización de código que agilice la puesta en servicio de múltiples modelos con menos esfuerzo.

## 3.2. Entorno de trabajo o workspace

**Despliegue del entorno de trabajo**

Existen **cuatro opciones** para desplegar un entorno de trabajo *(workspace)* en Azure Machine Learning: el cliente de línea de comandos de Azure, PowerShell, la REST API de Azure o el portal de Azure. Las tres primeras facilitan la **reproducibilidad** de los entornos, por lo que son las opciones preferidas. Con este fin, las tres pueden utilizar **plantillas ARM.** Estas plantillas son ficheros que describen las especificaciones del entorno. El servicio Azure Resource Manager (ARM) es el encargado de implementarlas, creando para ello la infraestructura necesaria en Azure. Lo veremos en los siguientes ejemplos, donde utilizaremos la línea de comandos de Azure.

Por ejemplo, con el siguiente comando se crea un entorno de *machine learning* con el nombre `entorno_ml`, en el `resource group` (carpeta de recursos), en los centros de datos de `West Europe`:

```
$ az ml workspace create -w entorno_ml -g grupo -l westeurope
```

Junto con el entorno se crean los siguientes **recursos:**

- Una **cuenta de almacenamiento,** que almacena código fuente, modelos y experimentos (diferentes configuraciones de modelos).
- Un **Azure Key Vault,** que almacena los datos sensibles (contraseñas, certificados) del entorno y de los sistemas a los que accede: entornos de almacenamiento de datos e infraestructura para entrenar o correr modelos.
- Una configuración de **Application Insights** para monitorizar el consumo de los recursos del entorno, como la CPU de los sistemas o el espacio de almacenamiento disponible.

- Un **Azure Container Registry.** Se trata de un Docker Registry —entorno de almacenamiento de imágenes Docker — en el que se publican las imágenes que implementan los procesos de entrenamiento o predicción.

**Elementos del entorno de trabajo**

Cada *workspace* incorpora los siguientes elementos:

| Nombre | Descripción |
|---|---|
| **Roles de usuario** | Permite especificar qué rol debe tener un usuario para poder ejecutar cada una de las acciones disponibles en el entorno. Por ejemplo, para obtener predicciones a través de un *endpoint* es necesario el rol owner, contributor o uno personalizado que incluya el permiso `/workspaces/services/aks/score/action` |
| **Experimentos** | Un experimento es la definición de un proceso de entrenamiento con una configuración determinada (con diferentes hiperparámetros o algoritmos). Guardar cada experimento es útil para determinar cuál de ellos ofrece mejores resultados. |
| **Datos y fuentes de datos** | Incluye los conjuntos de datos que se utilizan en el entorno y los sistemas en los que se almacenan. |
| **Infraestructura de computación** | Hace referencia a los sistemas en los que se realizan los procesos de entrenamiento de modelos y entrega de predicciones. Puede ser una máquina virtual o un *cluster* (conjunto de servidores). |
| **Entornos de entrenamiento** | Son contenedores Docker con el *software* necesario para las tareas de desarrollo en el proceso de entrenamiento. En cada proyecto pueden necesitarse distintos paquetes de *software* y versiones, es decir, un entorno distinto. Puede seleccionarse uno predefinido o configurar uno desde cero para adaptarlo a nuestros requerimientos. |

| Nombre | Descripción |
|---|---|
| **Ejecuciones** | Una ejecución es un proceso de entrenamiento completado, con una configuración determinada (experimento). Cada ejecución tiene asociada una configuración, que incluye: el *script* de entrenamiento, el entorno (*environment*) de entrenamiento y la infraestructura utilizada. En cada ejecución se generan: trazas de ejecución (*logs*), métricas del proceso, una copia del código fuente y los ficheros de salida generados al ejecutarlo. |
| **Modelos registrados** | Se refiere al almacenamiento, con una versión asociada, de los modelos generados tras cada proceso de entrenamiento. De esta forma, es posible seleccionar cuál de los modelos creados se despliega en producción para generar predicciones. |
| **Despliegues** | Se refiere a la acción de instalar un modelo en una infraestructura para que atienda solicitudes de predicciones a través de un servicio (*endpoint*). Este servicio puede funcionar en modo *online* (recibe una única petición en cada llamada) o *batch* (recibe muchas peticiones en una única llamada). |
| **Canalizaciones o *pipelines*** | Permiten implementar *workflows* con las actividades que forman parte del proceso de creación de un modelo. De esta forma, es más sencillo introducir automatizaciones en aquellas que son repetitivas. |

Tabla 1. Elementos de un *workspace.* Fuente: elaboración propia.

**Machine Learning Studio**

Una vez creado el *workspace,* podemos empezar a utilizarlo desde el servicio Azure Machine Learning Studio, disponible en el portal de Azure.

En la Figura 1, se muestra la página principal de Machine Learning Studio.

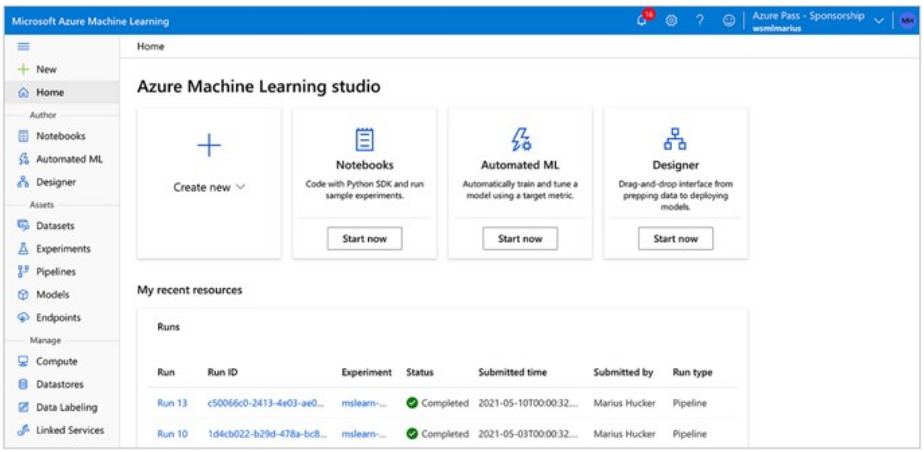



Figura 1. Página principal de Machine Learning Studio. Fuente: Marius, 2022.

Las **principales secciones** del servicio, en el menú de la izquierda, son las siguientes:

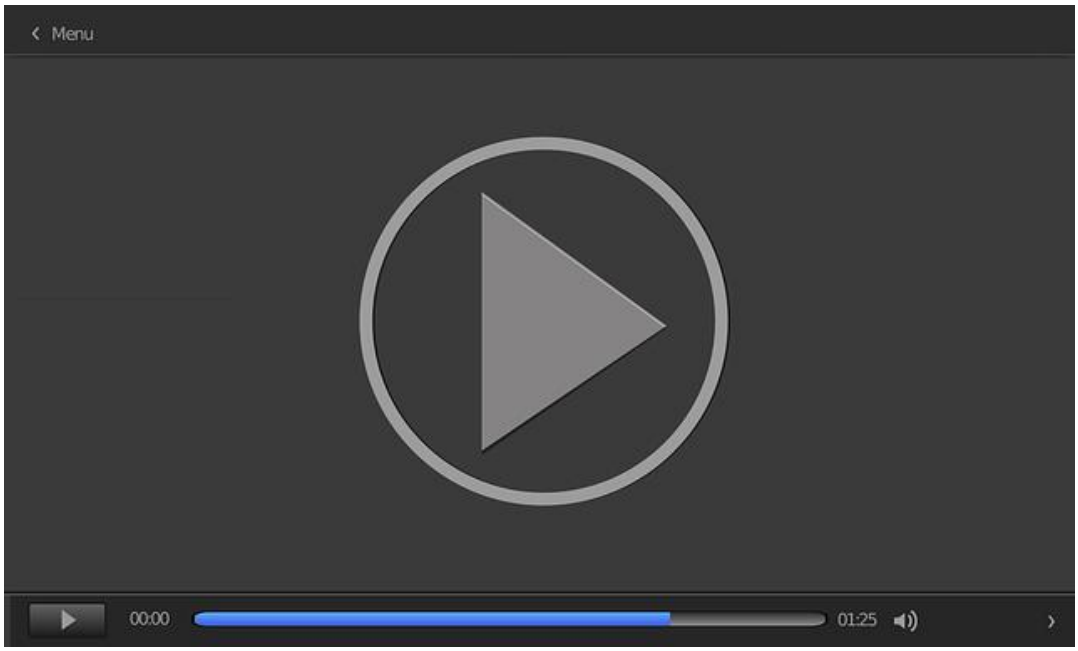
- **Author.** En esta sección, se proporcionan diferentes opciones para crear modelos: cuadernos Jupyter, AutoML o Designer, que es una interfaz gráfica para crear modelos a través de bloques integrados en un *workflow.*

- **Assets.** En esta sección, se agrupan varios elementos vistos en la Tabla 1: datos *(datasets),* experimentos, *pipelines,* registro de modelos y *endpoints* de predicción.

- **Manage.** En esta sección, se agrupan el resto de los elementos de la Tabla 1 y algunas opciones adicionales: infraestructura de computación, fuentes de datos, etiquetado (para clasificación de imágenes) y servicios enlazados (otros servicios de Azure para tratamiento de datos que pueden integrarse con el entorno de trabajo, como Azure Synapse).

# Ideas clave

En el siguiente vídeo, *Creación de workspaces e ingestión de datos en Azure Machine Learning Workspaces,* se muestra con un ejemplo real el proceso de puesta en marcha de un *workspace* en Azure Machine Learning.




Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=39649182-3751-4a52-88ac-b1480115a1e4

## 3.3. Ingestión y exploración de datos, ingeniería de variables y uso de pipelines

**Ingestión de datos**

Los datos en Azure Machine Learning se gestionan a través de los conceptos de ***dataset*** y ***datastore.*** Ambos abstraen al usuario de los detalles de ubicación, formato y permisos de acceso de los datos originales para que pueda poner el foco en la tarea de creación de modelos con estos datos.

U n *dataset* representa a los datos. Existen dos tipos: **tabulares** (CSV, Parquet, JSON, o resultados de consultas SQL) o **binarios** (imágenes, audio o vídeo). Los *datasets* definidos están disponibles a través de la sección correspondiente en Machine Learning Studio.

U n *datastore* se refiere al **sistema que almacena los datos.** Azure soporta los siguientes: contenedores de Blobs (ficheros de texto o binarios), ficheros en carpetas compartidas de Azure, *datalakes,* bases de datos SQL (SQL Server, MySQL, PostgreSQL) o Databricks. Los *datastores* se pueden crear o modificar desde la sección correspondiente en Machine Learning Studio. Una vez definido el *datastore,* es posible crear *datasets* a partir de los conjuntos de datos que almacena.

Los datos pueden ser ingestados en el *workspace* de Azure Machine Learning Studio de forma manual u automatizada. En cada caso, se utilizan diferentes herramientas.

Para **ingestión manual,** pueden usarse las siguientes:

- **Azure Storage Explorer:** cliente de escritorio que puede descargarse e instalarse en un PC. Permite subir datos a sistemas de almacenamiento en Azure o gestionar los datos existentes en ellos.

# Ideas clave

- **Azure CLI:** permite crear sistemas de almacenamiento o subir datos a estos sistemas a través del cliente de línea de comandos de Azure.
- **AzCopy:** utilidad de línea de comandos diseñada para subir datos a Azure.
- **Portal Azure:** el portal incluye una interfaz web para subir datos a los sistemas de almacenamiento de Azure.
- **Base de datos.** Es posible utilizar las herramientas nativas que proporcionan los sistemas de gestión de bases de datos soportados por Azure (SQL Server, MySQL, PostgreSQL) para crear tablas o *schemas,* o subir datos.
- **Azure Data Studio.** Es una aplicación de Microsoft para conectar de forma transparente a cualquier base de datos. De esta forma, el usuario evita tener que usar las herramientas nativas mencionadas en el punto anterior, que pueden ser más complejas.
- **Azure Machine Learning Designer.** Desde la interfaz gráfica de creación de modelos (Designer), es posible importar datos directamente a un *pipeline,* sin necesidad de crear un *datastore.*

Para **ingestión automatizada,** pueden usarse las siguientes:

- **Azure Data Factory.** Es la plataforma integrada de ingestión de datos de Azure. Permite conectar a cientos de fuentes de datos y definir un *workflow* para aplicar transformaciones a los datos ingestados. Estos datos transformados pueden a su vez almacenarse en otro sistema de almacenamiento. Por último, es posible explorar los datos a través de lenguaje DAX, el mismo que se utiliza en Excel o Power BI.
- **Azure Synapse Spark.** Es un sistema distribuido basado en Spark, que permite aplicar transformaciones a conjuntos de datos de gran volumen. Una vez tratados, quedan disponibles en Machine Learning Studio para llevar a cabo el proceso de creación de modelos.

**Exploración de datos**

La exploración de datos ingestados y convertidos en un *dataset* de Azure Machine Learning puede realizarse desde el propio cuaderno Jupyter.

En el siguiente ejemplo, se abre un dataset tabular (CSV) almacenado en un datastore, y se convierte en un dataframe de Pandas:

```
nombre_datastore = 'nombre_datastore'

datastore = Datastore.get(entorno_de_trabajo, nombre_datastore)

ruta_datastore = [(datastore, 'fichero.csv')]

dataset_tabular = Dataset.Tabular.from_delimited_files(path=
ruta_datastore)

df = dataset_tabular.to_pandas_dataframe()
```

A partir de este momento, pueden utilizarse todas las técnicas de análisis y visualización que proporcionan las librerías de Python: Pandas, Matplotlib y Seaborn.

**Ingeniería de variables**

Como en el caso anterior, la ingeniería de variables se realiza directamente en los **cuadernos Jupyter** utilizados para crear modelos. Sin embargo, Azure Machine Learning Studio proporciona una funcionalidad adicional para facilitar el proceso de etiquetado de imágenes y textos. Este proceso se emplea para construir modelos que detectan objetos en una imagen o interpretan contenido en textos. Se denomina Azure Machine Learning Labeling Service. Está disponible en la sección «Manage» del menú. Para empezar a utilizarlo, hay que crear un proyecto, haciendo clic en «Add Project».

Los **modos de etiquetado** soportados son los siguientes:

- Etiquetar con una **única clase** cada imagen.

# Ideas clave

- Asignar **varias etiquetas** a una misma imagen. Permite utilizar la imagen etiquetada en diferentes modelos de clasificación, con diferentes finalidades.
- Selección de **uno o varios objetos** en una imagen mediante rectángulos para facilitar el proceso de entrenamiento de un modelo que permita reconocer estos objetos.
- Dibujo de **polígonos** para delimitar de forma más precisa el objeto que queremos seleccionar en la imagen.
- Asignar una **única etiqueta** a un texto.
- Asignar **varias etiquetas** a un texto para que pueda ser utilizado por diferentes modelos de clasificación de textos.

La labor de etiquetado puede realizarla el propio usuario de forma manual, contratar a un equipo en el Azure Marketplace para que la realice por él o utilizar las facilidades de *machine learning* que ofrece Azure para realizar etiquetado automático.

**Uso de pipelines**

Los *pipelines* permiten dividir el proceso de creación de modelos en **pasos o bloques independientes** para facilitar la reutilización de estos. De esta forma, es posible crear un bloque que realiza, por ejemplo, una tarea de preparación de datos, y utilizarlo posteriormente en todos los ejercicios de modelado, independientemente del algoritmo empleado o el resto de las transformaciones que hagamos a los datos. Para ello, estos bloques pueden convertirse en **módulos parametrizables.**

En la Tabla 2, se muestran los **tipos de bloques** disponibles:

| Nombre | Descripción |
|---|---|
| `AutoMLStep` | Ejecuta una tarea con el sistema de AutoML de Azure Machine Learning, que permite construir modelos de forma automatizada. |
| `AzureBatchStep` | Ejecuta un *job* en el servicio de planificación de trabajos de Azure, que se utiliza para procesamiento *batch* a gran escala. |
| `DatabricksStep` | Ejecuta un cuaderno Databricks. Databricks es una plataforma de análisis de datos que soporta Python, Scala, R, Java y SQL. |
| `DataTransferStep` | Transfiere datos entre diferentes cuentas de almacenamiento de Azure. Una cuenta de almacenamiento proporciona espacio para subir blobs, ficheros, colas y tablas. |
| `HyperDriveStep` | El paquete HyperDrive permite identificar los hiperparámetros óptimos del modelo de forma automática. |
| `ModuleStep` | Ejecuta un módulo. Un módulo es una colección de *scripts* o ficheros binarios que permiten realizar una tarea. |
| `MpiStep` | Permite intercambiar mensajes entre nodos de una arquitectura con memoria distribuida. Se utiliza, entre otros casos de uso, para implementar deep learning en sistemas distribuidos. |
| `ParallelRunStep` | Ejecuta de forma asíncrona (no interactiva) una tarea que procesa un volumen elevado de datos. |
| `PythonScriptStep` | Ejecuta un *script* Python. |
| `RScriptStep` | Ejecuta un *script* en lenguaje de programación R. |
| `SynapseSparkStep` | Ejecuta un *script* Spark en el entorno Azure Synapse (*data warehouse*). |
| `CommandStep` | Ejecuta un comando en línea de comandos. |
| `KustoStep` | Ejecuta una consulta escrita en Kusto Query Language (KQL) para exploración de datos y descubrimiento de patrones. |

Tabla 2. Lista de bloques disponibles en los *pipelines.* Fuente: elaboración propia.

A continuación, se incluyen los pasos necesarios para crear y ejecutar un *pipeline* que incluye un único *step* de tipo PythonScriptStep, junto con el código fuente que implementa cada paso en Python:

- **1.** Cargar la configuración de nuestro workspace :

```
from azureml.core import Workspace
```

```
entorno = Workspace.from_config()
```

- **2.** Crear la infraestructura en la que se va a ejecutar el pipeline . En este caso, se utilizarán cuatro máquinas virtuales con 2 vCPUs, 7 GB RAM y 100 GB de espacio en disco duro (denominadas STANDARD_D2_V2 en Azure):

```
maquina_virtual ='STANDARD_D2_V2', max_nodes=4):

infraestructura = AmlCompute.provisioning_configuration(vm_size=
maquina_virtual, max_nodes=4)

amlcluster = ComputeTarget.create(entorno, 'cluster', infraestructura)

amlcluster.wait_for_completion(show_output=True)
```

- **3.** Definir el entorno de entrenamiento con las librerías necesarias:

```
paquetes = [ 'numpy', 'pandas', 'scikit-learn', 'tensorflow', 'azureml-
defaults' ]

configuracion = RunConfiguration()

configuracion.target = amlcluster

configuracion.environment.python.conda_dependencies =
CondaDependencies.create(pip_packages= paquetes)
```

- **4.** Definir el step , especificando el nombre del script , la ruta, el entorno de entrenamiento y la infraestructura de ejecución:

```
from azureml.pipeline.steps import PythonScriptStep

step = PythonScriptStep(name='Script Python',
script_name="script_python.py", source_directory="fuentes", runconfig=
configuracion, compute_target=amlcluster)
```

- **5.** Crear el pipeline, e incluir el step :

```
from azureml.pipeline.core import Pipeline

pipeline = Pipeline(entorno, steps=[step])
```

- **6.** Validar que el pipeline está correctamente definido:

```
pipeline.validate()
```

- **7.** Crear un nuevo experimento, encargado de ejecutar el pipeline :

```
from azureml.core import Experiment

experimento= Experiment(entorno, "azureml-pipeline")
```

- **8.** Ejecutar el experimento, es decir, el pipeline asociado, en la infraestructura de ejecución definida:

```
ejecucion = experimento.submit(pipeline)
```

## 3.4. Entrenamiento de modelos con Azure Machine Learning

El proceso de entrenamiento de modelos en Azure Machine Learning se realiza desde un cuaderno Jupyter, de forma similar al entorno de trabajo de un PC. Sin embargo, Azure proporciona, adicionalmente, un conjunto de **integraciones** que permiten explotar las funcionalidades propias del *workspace:* la selección de infraestructura de ejecución, los *datasets,* el registro de *logs* o el registro de experimentos/ejecuciones, modelos y entornos de entrenamiento.

En el siguiente ejemplo, se muestran los pasos necesarios para **entrenar un modelo de clasificación LGBM** en Azure Machine Learning, mostrando únicamente el código fuente asociado a dichas integraciones.

- **1.** Cargar la configuración de nuestro workspace , como en la sección anterior.
- **2.** Cargar el CSV en un dataframe de Pandas .
- **3.** Crear un dataset en el datastore por defecto del workspace , a partir del dataframe , y registrarlo en Azure Machine Learning para poder reutilizarlo en futuros experimentos:

```
datastore = entorno.get_default_datastore()

dataset = Dataset.Tabular.register_pandas_dataframe(dataframe, datastore,
name)
```

- **4.** Crear la infraestructura en la que se va a ejecutar el pipeline , como en la sección anterior.
- **5.** Definir el entorno de entrenamiento con las librerías necesarias, como en la sección anterior.

- **6.** Crear una nueva ejecución ( run ):

```
from azureml.core import Dataset, Run

run = Run.get_context()

entorno = run.experiment.workspace
```

- **7.** Cargar el dataframe a partir del dataset de Azure:

```
dataset = Dataset.get_by_id(entorno, id='id')

df = dataset.to_pandas_dataframe()
```

- **8.** Dividir los datos del dataframe en conjuntos de entrenamiento y pruebas.

- **9.** Guardar en las trazas ( logs ) de la ejecución los hiperparámetros del modelo almacenados en un diccionario Python params :

```
for k, v in params.items():

run.log(k, v)
```

- **10.** Lanzar el proceso de entrenamiento para el modelo, cuyos detalles dependen del algoritmo elegido. En este caso, se trata de un clasificador LGBM .

- **11.** Generar las métricas de rendimiento del clasificador, a partir del conjunto de datos de pruebas, y registrar los resultados en Azure Machine Learning:

```
y_pred = clf.predict(X_test)

run.log("accuracy (test)", accuracy_score(y_test, y_pred))

run.log("precision (test)", precision_score(y_test, y_pred))

run.log("recall (test)", recall_score(y_test, y_pred))

run.log("f1 (test)", f1_score(y_test, y_pred))
```

- **12.** Calcular el gráfico de importancia de variables —asumiendo que la implementación del algoritmo disponga de esta funcionalidad a través de la función plot_importance() , como el caso del clasificador LGBM — y registrarlo en Azure Machine Learning:

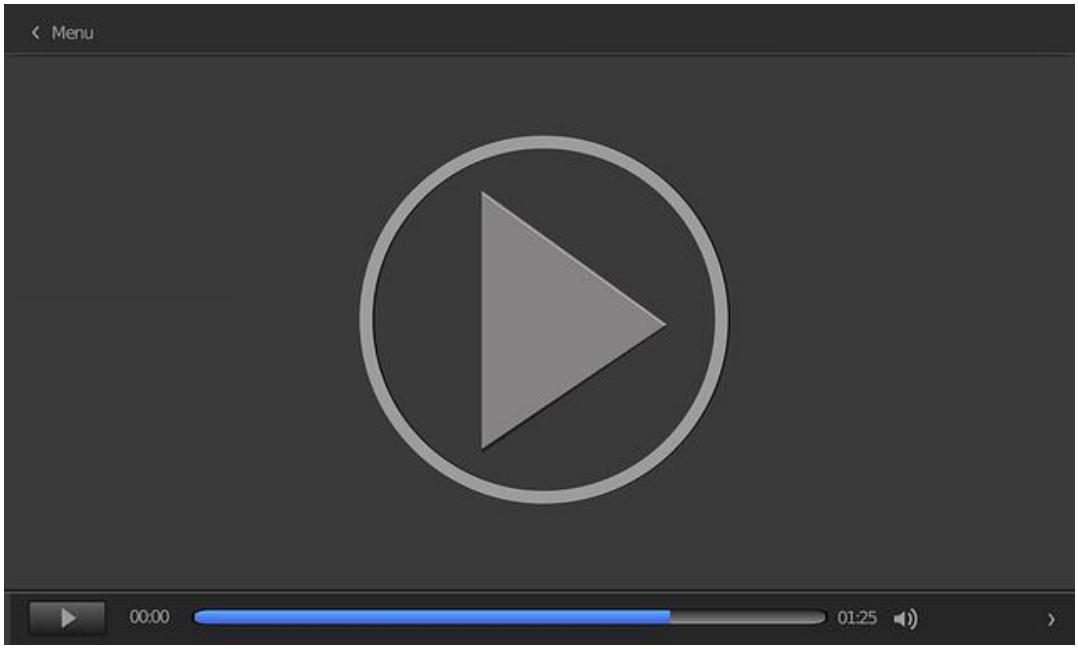
```
fig = plt.figure()

ax = plt.subplot(111)

modelo.plot_importance(clasificador, ax=ax)

run.log_image("feature importance", plot=fig)
```

- **13.** Registrar el modelo creado para poder desplegarlo posteriormente en un endpoint desde Azure Machine Learning Studio:

```
import joblib

joblib.dump(modelo, 'outputs/lgbm.pkl')

run.upload_file('lgbm.pkl', 'outputs/lgbm.pkl')

run.register_model(model_name='lgbm', model_path='lgbm.pkl')
```

# Ideas clave

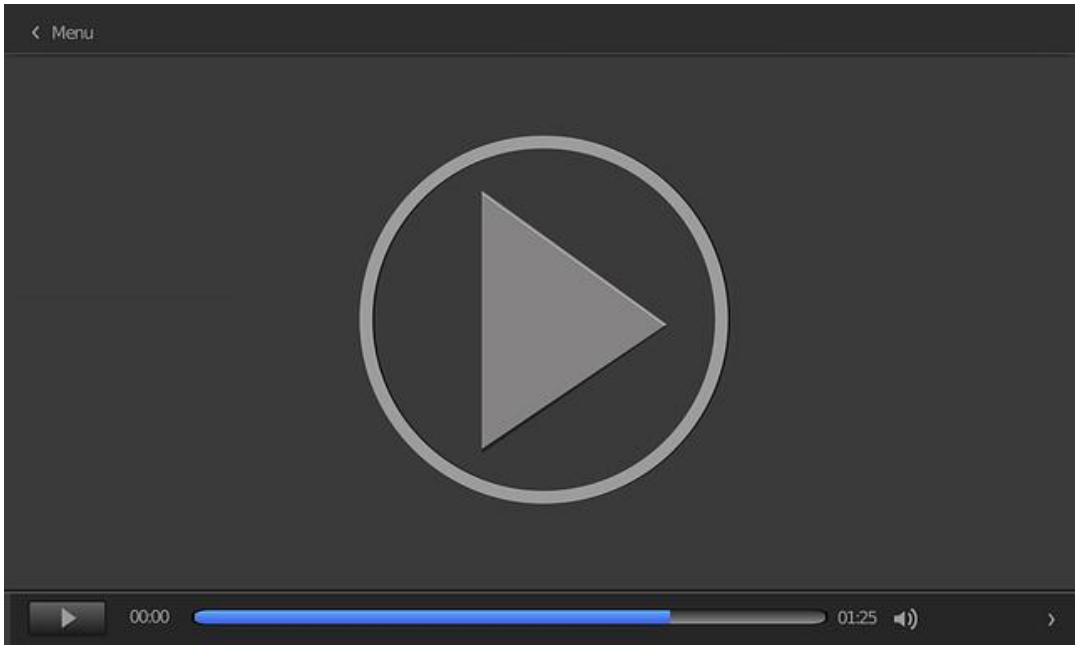
En el siguiente vídeo, *Elección y provisión de recursos de computación en Azure Machine Learning Workspaces,* se muestra un ejemplo real de provisión de infraestructura de computación y entrenamiento de modelos en Azure Machine Learning.




Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=17e1d08c-5faf-4036-88cf-b1480120af51

Adicionalmente, en el siguiente vídeo, *Entrenamiento y selección con modelos en Azure Machine Learning Designer,* se muestra un ejemplo real de entrenamiento y selección de modelos en Azure Machine Learning; en este caso, utilizando la herramienta Designer.




Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=fec15916-ed72-488a-9d07-b14900afd5d6

## 3.5. Entrenamiento de redes neuronales con aprendizaje profundo

En Azure Machine Learning es posible utilizar cuadernos Jupyter con *frameworks* de *deep learning* (aprendizaje profundo), como Keras (TensorFlow) o PyTorch, de forma similar a como lo haríamos en un entorno de PC convencional. Sin embargo, una de las ventajas de trabajar desde Azure Machine Learning es la posibilidad de ejecutar el proceso de entrenamiento en un ***cluster* de GPUs,** es decir, un entorno de *hardware* especializado en redes neuronales, que proporciona una mayor velocidad de ejecución.

En el siguiente ejemplo, se muestra cómo crear una **infraestructura de ejecución basada en GPUs,** en lugar de las máquinas virtuales (VMs), con CPU utilizadas en la sección «Uso de *pipelines* de este documento».

```
from azureml.core.workspace import Workspace

entorno = Workspace.from_config()

from azureml.core.compute import ComputeTarget, AmlCompute



nombre_cluster = "cluster-gpu"

tipo_vm = "STANDARD_NC6"

max_nodes = 3



infraestructura = AmlCompute.provisioning_configuration(vm_size=tipo_vm,
max_nodes=max_nodes)

amlcluster = ComputeTarget.create(entorno, nombre_cluster, infraestructura)
```

```
amlcluster.wait_for_completion(show_output=True)
```

La VM STANDARD_NC6 tiene una única GPU (basada en Nvidia Tesla K80), con 12 GB de memoria. Es la configuración más económica que ofrece Azure. En Tabla 3, se muestra el resto de las configuraciones para poder estimar la potencial mejora de rendimiento obtenida en el entrenamiento de modelos de aprendizaje profundo con cada una de ellas:

| Size | vCPU | Memory: GiB | Temp storage (SSD) GiB | GPU | GPU memory: GiB | Max data disks | Max NICs |
|---|---|---|---|---|---|---|---|
| Standard_NC6 | 6 | 56 | 340 | 1 | 12 | 24 | 1 |
| Standard_NC12 | 12 | 112 | 680 | 2 | 24 | 48 | 2 |
| Standard_NC24 | 24 | 224 | 1440 | 4 | 48 | 64 | 4 |
| Standard_NC24r* | 24 | 224 | 1440 | 4 | 48 | 64 | 4 |

1 GPU = one-half K80 card.

*RDMA capable

Tabla 3. Comparativa de VMs con GPUs en Azure. Fuente: Vikancha-Msft., 2023.

## 3.6. Ajuste de hiperparámetros con HyperDrive

El ajuste de hiperparámetros de un modelo en Azure Machine Learning se realiza con el paquete HyperDrive. Este paquete **automatiza** el proceso de ajuste, reduciendo considerablemente la complejidad asociada a esta tarea. Puede utilizarse desde Python para implementar las estrategias más comunes de búsqueda de hiperparámetros óptimos: *grid search, random search* y optimización bayesiana. — En las siguientes secciones se muestran ejemplos de implementación—.

**Implementación basada en grid search**

En la estrategia de *grid search* se prueban todas las combinaciones posibles de los valores de hiperparámetros proporcionados.

La implementación en Azure Machine Learning consta de los siguientes **pasos:**

- **1.** Creación de la configuración del grid . En este ejemplo, se proporcionan valores para el número de neuronas de las dos primeras capas de una red neuronal y también para su tamaño de batch :

```
from azureml.train.hyperdrive import GridParameterSampling

from azureml.train.hyperdrive.parameter_expressions import *

grid_sampling = GridParameterSampling({

    "--neuronas-capa-1": choice(16, 32, 64, 128),

    "--neuronas-capa-2": choice(16, 32, 64, 128),

    "--batch-size": choice(16, 32)

})
```

- **2.** Definir la métrica objetivo para determinar qué combinación de hiperparámetros es óptima. En este ejemplo, se utiliza la precisión ( accuracy ):

```
from azureml.train.hyperdrive import PrimaryMetricGoal

primary_metric_name = "accuracy"

primary_metric_goal = PrimaryMetricGoal.MAXIMIZE
```

- **3.** Especificar el script de entrenamiento, el entorno de entrenamiento y la infraestructura de ejecución:

```
src = ScriptRunConfig(source_directory="train", script="train.py",
compute_target=amlcluster, environment=entorno_de_entrenamiento)
```

- **4.** Inicializar la configuración de HyperDrive :

```
from azureml.train.hyperdrive import HyperDriveConfig

configuracion_hyperdrive = HyperDriveConfig(run_config=src,
hyperparameter_sampling=grid_sampling,
primary_metric_name=primary_metric_name,
primary_metric_goal=primary_metric_goal, max_total_runs=32,
max_concurrent_runs=4)
```

- **5.** Ejecutar la configuración de HyperDrive a través de un experimento:

```
from azureml.core.experiment import Experiment

experimento = Experiment(entorno, nombre_experimento)

ejecucion = experimento.submit(configuracion_hyperdrive)

print(ejecucion.get_portal_url())
```

La ejecución de HyperDrive retorna una URL en el portal de Azure que muestra los resultados obtenidos. En la Figura 2, se muestra un ejemplo de resultados con los diferentes valores de precisión.

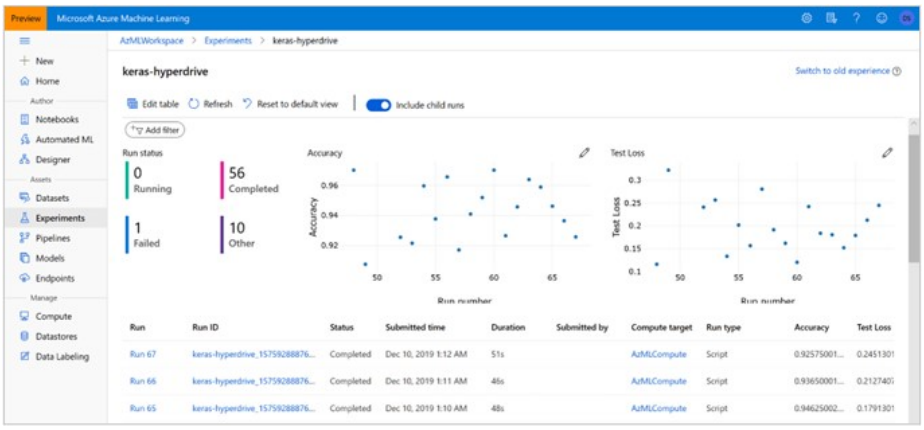



Figura 2. Resultados de la ejecución de HyperDrive para el ajuste de parámetros. Fuente: Soshnikov, 2020.

**Implementación basada en random search**

En la estrategia de *random search,* se prueba un número determinado de combinaciones aleatorias de los valores de hiperparámetros proporcionados.

La implementación en Azure Machine Learning solo se diferencia de la basada en *random search* en el primer paso (creación de la configuración).

A continuación, se muestra el código fuente actualizado:

```
from azureml.train.hyperdrive import RandomParameterSampling

from azureml.train.hyperdrive.parameter_expressions import *

random_sampling = RandomParameterSampling({

    "--neuronas-capa-1": choice(16, 32, 64, 128),

    "--neuronas-capa-2": choice(16, 32, 64, 128),

    "--batch-size": choice(16, 32)

})
```

```
max_total_runs = 25

max_duration_minutes = 60
```

En esta ocasión, se reduce el número total de combinaciones (32) a 25. Estas 25 combinaciones se eligen aleatoriamente.

**Implementación de finalización temprana**

Azure Machine Learning proporciona un **mecanismo para reducir el tiempo y los recursos invertidos** en el proceso de ajuste. Este mecanismo comprueba, en función de los criterios marcados por la política elegida, si se ha alcanzado un punto en el que no merece la pena seguir explorando combinaciones, porque las combinaciones restantes no lograrán superar los mejores resultados obtenidos hasta el momento.

Independientemente de la política de finalización temprana elegida, el mecanismo tiene asociados los siguientes **dos parámetros:**

- evaluation_interval **:** indica cada cuántas iteraciones del proceso de entrenamiento (por ejemplo, una *epoch* en una red neuronal) se va a comprobar si se cumplen los criterios de finalización temprana.

- delay_evaluation **:** indica cuántas iteraciones hay que esperar (desde el inicio del proceso de entrenamiento) para empezar a evaluar los criterios de finalización temprana.

# Ideas clave

Las **políticas de finalización temprana** pertenecen a una de las siguientes categorías.

- MedianStoppingPolicy **:** se cancela una ejecución ( run ) si la media móvil de los valores de rendimiento (según la métrica elegida) obtenidos hasta el intervalo actual está por debajo de la mediana de las medias móviles del resto de ejecuciones ( runs ).

- TruncationSelectionPolicy **:** se cancelan las ejecuciones ( runs ) cuyos valores de rendimiento (según la métrica elegida) se encuentren entre los peores de todas las ejecuciones, hasta un porcentaje definido como parámetro. Por ejemplo: si el valor es 10 %, se cancelan el 10 % de los runs , concretamente el 10 % que están obteniendo peor rendimiento.

- BanditPolicy **:** se cancelan las ejecuciones ( runs ) que tienen peor rendimiento que el *run* con mejor rendimiento hasta el momento. Se define un factor de desviación *(slack factor)* para determinar en qué medida debe ser peor el rendimiento para que en efecto un run sea cancelado. Con la introducción de este factor es posible mantener aquellos runs cuyo rendimiento es solo ligeramente peor que el del mejor run .

**Implementación basada en optimización bayesiana**

En la **estrategia de optimización** bayesiana se utiliza el resultado de las combinaciones probadas para determinar cuáles son las mejores candidatas que deben probarse a continuación. Por tanto, se trata de una **búsqueda dirigida.**

La implementación en Azure Machine Learning solo se diferencia de la basada en las anteriores en el primer paso (creación de la configuración).

A continuación, se muestra el **código fuente actualizado:**

```
from azureml.train.hyperdrive import BayesianParameterSampling

from azureml.train.hyperdrive.parameter_expressions import *

bayesian_sampling = BayesianParameterSampling({

    "--neuronas-capa-1": choice(16, 32, 64, 128),

    "--neuronas-capa-2": choice(16, 32, 64, 128),

    "--batch-size": choice(16, 32)

})
```

Al utilizar esta estrategia, es importante reducir el número máximo de ejecuciones concurrentes. Si existe demasiada concurrencia, la estrategia no dispone de suficiente información para dirigir la búsqueda eficazmente, dado que solo puede tener en cuenta los resultados de cada hilo, de forma independiente al resto.

## 3.7. AutoML: machine learning automatizado

En líneas generales, se denomina AutoML al **proceso automático de creación de modelos,** en el que no es necesario utilizar un lenguaje de programación ni disponer de conocimientos avanzados en ciencia de datos.

Azure Machine Learning dispone de capacidades de AutoML, que permiten automatizar cada una de las fases del proceso de creación de modelos. Sin embargo, el usuario puede elegir si desea configurar el proceso desde una **interfaz gráfica** (el portal de Azure) o desde un **lenguaje de programación** (Python). En ambos casos, se obtiene un modelo que permite predecir una variable a partir de los datos de entrada, pero sin necesidad de especificar las transformaciones de datos, elegir un algoritmo, especificar hiperparámetros o llevar a cabo la selección del modelo óptimo.

En el siguiente ejemplo, se proporcionan los pasos que se deben seguir para utilizar las capacidades de AutoML. Se ha elegido la opción de Python porque presenta ventajas de trazabilidad y reutilización respecto a la alternativa del *wizard* en el portal de Azure.

- **1.** En primer lugar, se fijan **valores** para los parámetros de configuración del proceso de AutoML. Son los siguientes: tiempo máximo de ejecución, número de *folds* en el proceso de *cross-validation,* métrica de rendimiento, estrategia de ingeniería de variables (manual o automática), aplicación automática de técnicas de transformación de datos (sí o no) y el nivel de detalle aportado en las trazas (logs) del proceso.

```
parametros = { "experiment_timeout_minutes": 15, "n_cross_validations": 3,
"pri-mary_metric": 'accuracy', "featurization": 'auto', "preprocess": True,
"verbosity": logging.INFO,

}
```

- **2.** Se cargan los datos en un dataframe de Pandas y se generan los conjuntos de datos de entrenamiento y pruebas.

- **3.** Se inicializa el proceso de AutoML con los parámetros definidos en el paso 1 y se añaden los siguientes: tipo de modelo (clasificación, regresión), fichero en el que se almacenan trazas ( logs ) para depuración, infraestructura de ejecución, dataframe con los datos de entrenamiento y variable que debe predecirse.

```
from azureml.train.automl import AutoMLConfig

configuracion = AutoMLConfig(task='classification', debug_log='debug.log',
compute_target=amlcluster, training_data=df_datos_entrenamiento,
label_column_name=variable_objetivo, **automl_settings)
```

- **4.** Se ejecuta el proceso de AutoML a través de un run :

```
from azureml.widgets import RunDetails

ejecucion = experiment.submit(configuracion, show_output=False)

RunDetails(ejecucion).show()
```

- **5.** Se selecciona el modelo óptimo:

```
mejor_ejecucion, mejor_modelo = ejecucion.get_output()
```

- **6.** Por último, se utiliza el modelo obtenido para generar predicciones, como lo haríamos con cualquier otro modelo en Python.

```
from sklearn.metrics import accuracy_score

y_test = df_datos_prueba[variable_objetivo]

X_test = df_datos_prueba.drop(variable_objetivo, axis=1)

y_pred = mejor_modelo.predict(X_test)

accuracy_score(y_test, y_pred)
```

## 3.8. Deep learning distribuido en Azure

**Introducción**

El **paralelismo de datos** permite reducir el tiempo necesario para completar el proceso de entrenamiento de un modelo, utilizando para ello varios nodos (sistemas) de forma simultánea, en lugar de depender de un único nodo.

El paralelismo de datos se **implementa** de la siguiente forma:

- Se dividen los datos en bloques y se envía cada bloque a uno de los nodos.
- Se utiliza el *distributed gradient descent* (DGD) para optimizar el modelo. DGD es un optimizador similar a *gradient descent,* pero diseñado para funcionar en entornos distribuidos (con múltiples nodos que participan en el proceso de entrenamiento).

El *framework* **Horovod,** desarrollado por Uber, es una de las alternativas para implementar paralelismo de datos en redes neuronales. Está disponible en Azure Machine Learning, y puede aplicarse a TensorFlow (Keras), PyTorch o Apache Spark.

**Uso de Horovod**

En el siguiente ejemplo, se describen los pasos a seguir para utilizar Horovod en el proceso de entrenamiento de una red neuronal con Keras.

En primer lugar, se genera un fichero `train.py`, que contiene el código fuente de entrenamiento del modelo con Horovod.

- **1.** Inicializar Horovod:

```
import horovod.keras as hvd

hvd.init()
```

- **2.** Configurar TensorFlow para que cada proceso esté asignado a una GPU en la arquitectura distribuida:

```
from tensorflow.keras import backend as K

import tensorflow as tf

config = tf.ConfigProto()

config.gpu_options.allow_growth = True

config.gpu_options.visible_device_list = str(hvd.local_rank())

K.set_session(tf.Session(config=config))
```

- **3.** Generar conjuntos de datos de entrenamiento y pruebas, y definir los hiperparámetros de la red neuronal.

- **4.** Reemplazar el optimizador por defecto (Adadelta) por el de Horovod, que funciona en múltiples nodos de forma paralela. En este paso, también se ajusta el *learning rate* en función del número de nodos disponibles.

```
from tensorflow.keras.optimizers import Adadelta

optimizador = Adadelta(1.0 * hvd.size())

optimizador = hvd.DistributedOptimizer(opt)
```

- **5.** Compilar el modelo. A continuación, ejecutar el proceso de entrenamiento, especificando un callback que inicializa los gradientes de todos los nodos en la arquitectura distribuida:

```
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=optimizador, metrics=['accuracy'])

callbacks = [ hvd.callbacks.BroadcastGlobalVariablesCallback(0) ]

model.fit(x_entrenamiento, y_entrenamiento, batch_size=batch_size,
callbacks=callbacks, epochs=epochs, verbose=1 if hvd.rank() == 0 else 0,
validation_data=(x_pruebas, y_pruebas))
```

- **6.** Evaluar el rendimiento del modelo con el conjunto de datos de prueba.

```
score = model.evaluate(x_pruebas, y_pruebas)
```

A continuación, se genera una configuración de ejecución con MPI a partir del fichero train.py del paso anterior. MPI es un estándar de intercambio de mensajes entre nodos de una arquitectura distribuida; y Horovod lo utiliza para implementar el proceso de entrenamiento con paralelismo de datos.

```
from azureml.core import ScriptRunConfig

from azureml.core.runconfig import MpiConfiguration

mpi = MpiConfiguration(process_count_per_node=1, node_count=2)

src = ScriptRunConfig(source_directory='fuentes', script='train.py',
run_config=configuracion, arguments=parametros, distributed_job_config=mpi)
```

Por último, se ejecuta la configuración definida de la misma manera que hemos visto en ejemplos de secciones previas.

```
ejecucion = experimento.submit(src)
```

## 3.9. Despliegue, operación de modelos y acceso vía API

**Despliegue de modelos**

De forma general, para desplegar un modelo en Azure Machine Learning se necesitan los siguientes **componentes:**

- Un modelo entrenado.
- Una imagen Docker que define el **entorno de ejecución** del servicio de predicciones (inferencias).
- Un `script` para invocar la función `predict()` del modelo, es decir, para generar una predicción a partir de los datos de entrada.
- Un entorno de *software* para el `script` anterior, similar al entorno de entrenamiento basado en Conda, descrito en ejemplos previos. Por ejemplo, un entorno Python con todas las librerías necesarias.
- La infraestructura sobre la que correrá el entorno de ejecución, es decir, el contenedor asociado a la imagen Docker.

Sin embargo, podemos simplificar el proceso de despliegue, utilizando la funcionalidad de **autodespliegue** que proporciona Azure Machine Learning para escenarios en los que el grado de personalización del despliegue es bajo. En este caso, solo es preciso especificar el modelo que queremos desplegar, los recursos *hardware* necesarios y el *framework* de Machine Learning que necesita el modelo para ejecutarse: Azure se encarga de automatizar el resto de los pasos.

Vamos a ver un ejemplo de autodespliegue con un modelo basado en ***scikit-learn:***

El primer paso es especificar los parámetros del entorno de ejecución. En este caso, se utilizará una única CPU y 2 GB de RAM.

```
from azureml.core.resource_configuration import ResourceConfiguration

configuracion = ResourceConfiguration(cpu=1, memory_in_gb=2.0, gpu=0)
```

El siguiente paso es indicar qué modelo queremos desplegar y el *framework* que requiere para ejecutarse. El modelo debe serializarse y registrarse en el registro de modelos *(model registry)* de Azure Machine Learning. El **proceso de serialización** transforma el modelo en un fichero binario que puede cargarse posteriormente en otro entorno, el que utilizaremos para realizar inferencias. Para serializar un modelo, se utiliza la función `dump()` de Joblib.

```
from azureml.core import Model

import joblib

joblib.dump(clf, 'outputs/modelo1.pkl')

modelo1 = run.register_model(model_name='modelo1',
model_path='outputs/modelo1.pkl',
model_framework=Model.Framework.SCIKITLEARN,
model_framework_version='0.24.2', resource_configuration= configuracion)
```

Por último, se realiza el despliegue del modelo en un servicio a partir de la configuración anterior. Este servicio está disponible a través de una URL.

```
nombre_servicio = 'servicio1'

servicio = Model.deploy(entorno_de_trabajo, nombre_servicio, [modelo1])

servicio.wait_for_deployment(show_output=True)

print("Scoring URL: " + service.scoring_uri)
```

Ahora es posible utilizar el servicio de inferencias para obtener las predicciones del modelo a partir de los datos de entrada especificados.

**Operación de modelos**

Después del despliegue, es importante llevar a cabo una **serie de actividades** en el ámbito de operaciones para garantizar que el modelo continúa prestando el servicio eficazmente a lo largo del tiempo. Azure Machine Learning proporciona **funcionalidades *ad-hoc*** para cada una de estas actividades.

**Perfilado del modelo**

Permite identificar los **recursos *hardware*** recomendados para que el modelo tenga una latencia aceptable, es decir, que proporciona una rápida respuesta a las solicitudes de predicciones.

En el siguiente ejemplo, se muestra cómo se puede obtener esta información:

```
profile = Model.profile(entorno_de_trabajo, servicio, [modelo1],
configuracion_inferencia, datos_prueba)

profile.wait_for_profiling(True)

print(profile.get_results())
```

Los **parámetros de la función** `profile()` son: el *workspace* de Azure Machine Learning, el servicio de predicciones, el modelo desplegado, la configuración de inferencia (que incluye el *script* `score.py` y el entorno de *software* asociado) y los datos de prueba, que se utilizan para solicitar predicciones y medir el tiempo de respuesta.

**Generación de trazas de infraestructura y aplicación**

El servicio **Application Insights** de Azure permite recoger trazas (logs) generadas por diferentes fuentes. En nuestro escenario, nos interesa monitorizar las trazas de la infraestructura en la que corre el modelo y las generadas por el propio modelo. De esta forma, es posible identificar errores de forma temprana y analizar cómo se está utilizando el servicio de predicciones.

En el siguiente ejemplo se configuran las trazas de la GPU sobre la que corre el modelo. Se utiliza nvidia_smi para capturar las métricas de la GPU y track_metric() para enviarlas a Application Insights.

```
from applicationinsights import TelemetryClient

import nvidia_smi

nvidia_smi.nvmlInit()

dev_handle = nvidia_smi.nvmlDeviceGetHandleByIndex(0)

res = nvidia_smi.nvmlDeviceGetUtilizationRates(dev_handle)

tc = TelemetryClient("<insert appinsights key")

tc.track_metric("gpu", res.gpu)

tc.track_metric("gpu-gpu-mem", res.memory)
```

En la Figura 3 se muestra Application Insights en el portal de Azure:

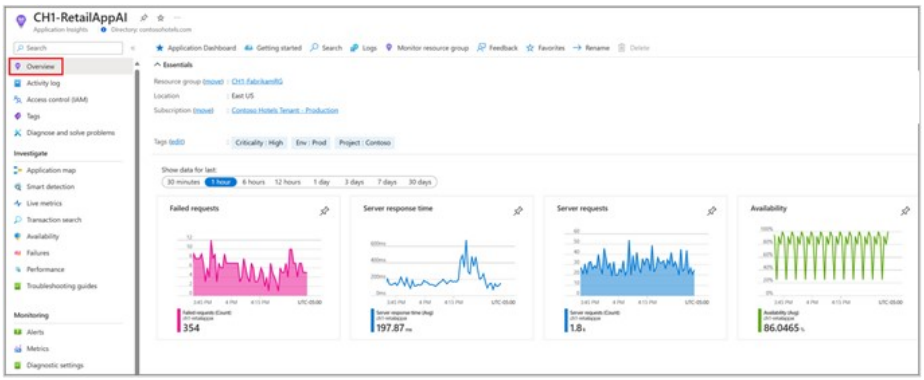



Figura 3. Interfaz web de Application Insights con las métricas recolectadas. Fuente: AaronMaxwell et al., 2023.

### Detección de *concept drift*

El *concept drift* se refiere a los **cambios** que se producen a lo largo del tiempo en las relaciones entre los datos utilizados para generar predicciones. Estos cambios implican que el modelo actual deja de ser eficaz y es necesario reentrenarlo con nuevos datos que reflejen estos cambios.

Azure Machine Learning proporciona una funcionalidad para detectar estos cambios de forma sencilla: el *data drift detector.* Esta funcionalidad puede utilizarse a través de la clase `azureml.datadrift.DataDriftDetector` en Python.

En la Figura 4, se muestran los resultados generados para medir el *concept drift* de un modelo en el portal de Azure:

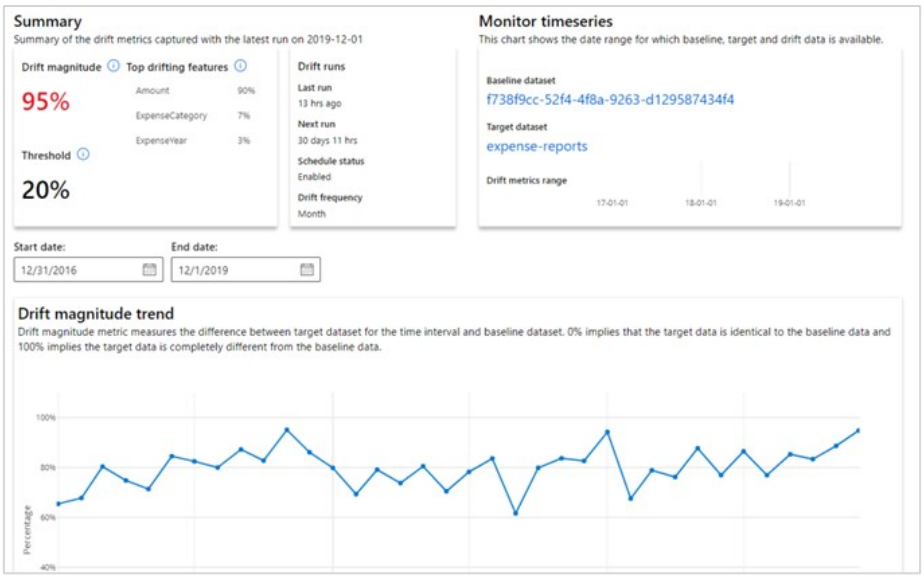



Figura 4. Resultados del análisis de *concept drift* en Azure Machine Learning. Fuente: Feasel, 2021.

## 3.10. Cuaderno de ejercicios

**1.** Utilizando la clase `dataset` de Azure Machine Learning, escribe el código fuente en lenguaje Python que realiza las siguientes tareas:

- Crear un `dataset` de tipo tabular a partir de un fichero `CSV` publicado en https://servidor.com/fichero.csv, con el nombre `nombre_dataset` y la descripción `descripcion_dataset` .
- Convertir el `dataset` en un `dataframe` de Pandas.

```
from azureml.core.dataset import Dataset

url_datos = 'https://servidor.com/fichero.csv'

dataset = Dataset.Tabular.from_delimited_files(url_datos)

dataset = dataset.register(workspace=entorno_de_trabajo,
name=nombre_dataset, description=descripcion_dataset)

df = dataset.to_pandas_dataframe()
```

**2.** Utilizando las clases `AmlCompute` y `ComputeTarget` de Azure Compute, escribe el código fuente en lenguaje Python, que realiza las siguientes tareas en el `workspace` `entorno_de_trabajo` *:*

- Obtiene la infraestructura para Azure Machine Learning previamente creada (con el nombre `"cluster"` ), y si no existe:
  - Crea esta infraestructura a partir de una VM de tipo `STANDARD_DS12_V2` , con un máximo de 4 nodos.
  - Espera un máximo de 10 minutos a que esté creado al menos 1 de los 4 nodos, y proporciona una salida detallada del resultado.

```
from azureml.core.compute import AmlCompute

from azureml.core.compute import ComputeTarget

from azureml.core.compute_target import ComputeTargetException



amlcluster_name = "cluster"



try:

    infraestructura = ComputeTarget(workspace=entorno_de_trabajo,
name=amlcluster_name)

except ComputeTargetException:

    configuracion =
AmlCompute.provisioning_configuration(vm_size='STANDARD_DS12_V2',
max_nodes=4)

    infraestructura = ComputeTarget.create(entorno_de_trabajo,
amlcluster_name, configuracion)

compute_target.wait_for_completion(show_output=True, min_node_count = 1,
timeout_in_minutes = 10)
```

**3.** Utilizando el dataset y la infraestructura creada en los ejercicios previos, escribe el código fuente en lenguaje Python que permite generar la configuración de AutoML con las siguientes especificaciones:

- Tiempo máximo de ejecución del experimento: 20 minutos .
- Número máximo de iteraciones concurrentes: 4 .
- Métrica de rendimiento: AUC con pesos.
- Tipo de modelo: clasificador .
- Nombre de la variable a predecir: variable_objetivo .
- Ruta al proyecto: ./project .
- Finalización temprana: activada .
- Generación automática de variables: activada .
- Fichero de trazas de depuración: automl_errors.log .

```
parametros_automl = {

    "experiment_timeout_minutes": 20,

    "max_concurrent_iterations": 4,

    "primary_metric" : 'AUC_weighted'

}

configuracion_automl = AutoMLConfig(compute_target=infraestructura, task =
"classification", training_data=dataset,
label_column_name="variable_objetivo", path = './project',
enable_early_stopping= True, featurization= 'auto', debug_log =
"automl_errors.log", ** parametros_automl)
```

**4.** Utilizando las clases ScriptRunConfig y Environment de Azure Machine Learning, escribir el código fuente en lenguaje Python que realiza las siguientes tareas en el workspace definido en el fichero de configuración estándar:

- Carga el workspace .
- Crea el entorno de *software* para R con el nombre entorno_r a partir de un fichero Dockerfile almacenado en la carpeta recursos.
- Obtiene el dataset almacenado en el datastore por defecto, con la ruta datos.
- Crea una configuración de entrenamiento con los siguientes parámetros:
  - Carpeta de código fuente: recursos .
  - Nombre del *script* en R: training.R .
  - Carpeta de datos: obtenida a partir del dataset .
  - Carpeta de resultados: salida .
  - Infraestructura: infraestructura .
  - Entorno de *software:* el mismo que se ha creado en el paso 2.

```
from azureml.core import Environment

from azureml.core import Dataset

from azureml.core import ScriptRunConfig

import os



carpeta = 'recursos'

entorno_de_trabajo = Workspace.from_config()
```

```
entorno_dockerfile = Environment.from_dockerfile(name='entorno_r',
dockerfile=os.path.join(src_dir, 'Dockerfile'))
```

```
datastore = entorno_de_trabajo.get_default_datastore()

dataset = Dataset.File.from_files(datastore.path('datos'))
```

```
configuracion_entrenamiento = ScriptRunConfig(source_directory=carpeta,
command=['Rscript training.R --data_folder', dataset.as_mount(), '--
output_folder salida'], compute_target=infraestructura,
environment=entorno_dockerfile)
```

**5.** Utilizando las clases de HyperDrive RandomParameterSampling y BanditPolicy , escribe el código fuente en Python para configurar un *random search* con las siguientes características:

- Hiperparámetros objetivo:
  - batch_size *:* valores 25, 50, 100.
  - neuronas-capa-1 : valores 10, 50, 200, 300, 500.
  - neuronas-capa-2 : valores 10, 50, 200, 500.
  - learning-rate *:* distribución *loguniform* con intervalo [-6, -1].
- Política de finalización temprana de tipo bandit , con un intervalo de evaluación de 2 y un slack factor de 0.1.
- Configuración de entrenamiento del ejercicio anterior.
- Objetivo: maximizar *accuracy* del conjunto de datos de validación.
- Prueba de 4 combinaciones.

- Máximo 4 ejecuciones concurrentes.

```
from azureml.train.hyperdrive import RandomParameterSampling, BanditPolicy,
HyperDriveConfig, PrimaryMetricGoal

from azureml.train.hyperdrive import choice, loguniform



random_search = RandomParameterSampling(

    {

        '--batch-size': choice(25, 50, 100),

        '--neuronas-capa-1': choice(10, 50, 200, 300, 500),

        '--neuronas-capa-2': choice(10, 50, 200, 500),

        '--learning-rate': loguniform(-6, -1)

    }

)



finalizacion_temprana = BanditPolicy(evaluation_interval=2,
slack_factor=0.1)



configuracion_hiperdrive =
HyperDriveConfig(run_config=configuracion_entrenamiento,
hyperparameter_sampling= random_search, policy=finalizacion_temprana,
primary_metric_name='validation_acc',
primary_metric_goal=PrimaryMetricGoal.MAXIMIZE, max_total_runs=4,
max_concurrent_runs=4)
```

## 3.11. Referencias bibliográficas

AaronMaxwell, et al. (2023, octubre 25). Panel de Información general de Application Insights - Azure Monitor. Microsoft Learn. https://learn.microsoft.com/es-es/azure/azure-monitor/app/overview-dashboard

Feasel, K. (2021, marzo 22). Dataset Drift Monitoring with Azure ML. *36 Chambers - The Legendary Journeys: Execution To The Max!* https://36chambers.wordpress.com/2021/03/23/dataset-drift-monitoring-with-azure-ml/

Marius, H. (2022, enero 5). A brief introduction to Azure Machine Learning Studio. *Medium.* https://towardsdatascience.com/a-brief-introduction-to-azure-machine-learning-studio-9bbf41800a60

Soshnikov, D. (2020, marzo 18). Using Azure Machine Learning for Hyperparameter Optimization. *Microsoft Blog.* https://techcommunity.microsoft.com/t5/educator-developer-blog/using-azure-machine-learning-for-hyperparameter-optimization/ba-p/1236469

Vikancha-Msft. (2023, julio 24). NC-Series - Azure Virtual Machines. Microsoft Learn. https://learn.microsoft.com/en-us/azure/virtual-machines/nc-series

## Documentación de Azure Machine Learning

Sdgilley (s. f.). *Documentación de Azure Machine Learning.* Microsoft Learn. https://learn.microsoft.com/es-es/azure/machine-learning/?view=azureml-api-2

La documentación oficial de Microsoft es un excelente recurso para comprender el detalle de cada una de las funcionalidades vistas en este tema. Adicionalmente, esta fuente es la que alberga todas las actualizaciones que se realizan al servicio, a diferencia de otras referencias bibliográficas que pueden quedar rápidamente obsoletas.

## Azure Machine Learning Service

Jainani, P. (2021, diciembre 14). Azure Machine Learning Service: Part 1 — An introduction. *Medium.* https://towardsdatascience.com/azure-machine-learning-service-part-1-an-introduction-739620d1127b

En esta serie de cinco publicaciones, se describen las principales características de Azure Machine Learning a través de ejemplos prácticos. Es posible seguir las explicaciones a través de los diferentes *notebooks* que el autor ha creado para cada una de las temáticas.

# Test

**1.** ¿Qué es un entorno *(environment)* de entrenamiento en Azure Machine Learning?

A. Es una máquina virtual en la que se ejecuta el proceso de entrenamiento.

B. Es un contenedor Docker con el *software* necesario para las tareas de desarrollo en el proceso de entrenamiento.

C. Es un sinónimo de experimento *(experiment).*

D. Es un contenedor Docker especializado en el ajuste de hiperparámetros.

**2.** ¿Para qué sirve la utilidad AzCopy de Azure?

A. Es el cliente de línea de comandos de Azure, que sirve para crear, modificar o eliminar un recurso en Azure.

B. Es una utilidad de línea de comandos diseñada para subir datos a Azure.

C. Es una herramienta disponible en el portal de Azure para subir datos a un *workspace* de Azure Machine Learning.

D. Es una herramienta disponible en el portal de Azure para copiar datos entre cuentas de almacenamiento.

**3.** ¿Cuál es la diferencia entre un *dataset* y un *datastore* en Azure Machine Learning?

A. Un *dataset* es una fuente de datos, mientras que un *datastore* es una abstracción de los datos reales que están almacenados en un *dataset.*

B. Un *datastore* es una fuente de datos, mientras que un *dataset* es una abstracción de los datos reales que están almacenados en un *datastore.*

C. Ambos son conceptos equivalentes en Azure Machine Learning.

D. Un *dataset* es una abstracción de los datos reales, mientras que el *datastore* son los datos reales.

**4.** ¿Qué función cumple un HyperDriveStep en un *pipeline* de Azure Machine Learning?

A. Permite transferir datos de un *datastore* a un *workspace* de Azure Machine Learning.

B. Permite automatizar el ajuste de hiperparámetros.

C. Permite paralelizar la ejecución de procesos de entrenamiento.

D. Sirve para configurar la infraestructura en la que se ejecutarán los procesos de entrenamiento.

**5.** ¿Cuál es la principal ventaja en el uso de *pipelines* en Azure Machine Learning?

A. Visualmente, queda mucho más claro qué finalidad tiene el modelo que se genera como resultado de la ejecución del *pipeline.*

B. Permiten reutilizar código fuente asociado a determinadas tareas repetitivas entre experimentos.

C. No presenta ninguna ventaja; es simplemente el enfoque adoptado en Azure Machine Learning para abordar la creación de modelos. Otros proveedores de nube han adoptado enfoques diferentes igualmente válidos.

D. El hecho de que cada bloque del *pipeline* se realiza de forma automática, a diferencia de los *notebooks,* en los que el trabajo es 100 % manual.

**6.** ¿Qué es Horovod y para qué se utiliza en Azure Machine Learning?

A. Es un lenguaje de programación orientado a redes neuronales.

B. El *framework* Horovod sirve para implementar paralelismo de datos en redes neuronales.

C. Horovod es un lenguaje de programación que sirve para implementar paralelismo de datos en redes neuronales.

D. Es un *framework* diseñado para entrenar redes neuronales con GPUs.

**7.** ¿Para qué sirve el parámetro **evaluation_interval** en HyperDrive?

A. Indica en qué posiciones del *dataset* se encuentran los datos de validación.

B. Indica cada cuántas ejecuciones debe evaluarse la política de finalización temprana.

C. Indica cada cuántos minutos debe evaluarse la política de finalización temprana.

D. Indica en qué ejecuciones específicas debe evaluarse la política de finalización temprana.

**8.** ¿En qué se diferencia *gradient descent* de *distributed gradient descent*?

A. En el caso de Azure Machine Learning, son términos intercambiables.

B. DGD es un optimizador similar a *gradient descent,* pero diseñado para funcionar en entornos distribuidos (con múltiples nodos que participan en el proceso de entrenamiento).

C . *Gradient descent* está optimizado para entornos distribuidos en general, mientras que DGD se especializa en entornos distribuidos con GPUs.

D. DGD es una variante optimizada de *gradient descent* que se ejecuta más rápidamente, y por tanto permite completar el proceso de entrenamiento en un tiempo inferior.

**9.** ¿Para qué sirve la clase **azureml.datadrift.DataDriftDetector** del SDK?

A. Sirve para corregir el *concept drift* detectado en los datos.

B. Sirve para detectar *concept drift* en los datos.

C. Sirve para detectar y corregir el *concept drift* en los datos.

D. Sirve para detectar datos obsoletos en un *dataset.*

**10.** ¿Cómo funciona la *median stopping policy,* en el contexto de la finalización temprana de HyperDrive?

A. Se cancela una ejecución *(run)* si la mediana de los valores de rendimiento (según la métrica elegida) obtenidos hasta el intervalo actual está por debajo de la mediana de las medias móviles del resto de ejecuciones *(runs)*

B. Se cancela una ejecución *(run)* si la media móvil de los valores de rendimiento (según la métrica elegida) obtenidos hasta el intervalo actual está por debajo de la mediana de las medias móviles del resto de ejecuciones *(runs)*

C. Se cancela una ejecución *(run)* si la mediana de los valores de rendimiento (según la métrica elegida) obtenidos hasta el intervalo actual está por debajo del promedio de las medias móviles del resto de ejecuciones *(runs).*

D. Se cancela una ejecución *(run)* si la mediana de los valores de rendimiento (según la métrica elegida) obtenidos hasta el intervalo actual está por encima del promedio de las medias móviles del resto de ejecuciones *(runs).*