Data in Azure Machine Learning

Article • 08/19/2022 • 8 minutes to read

Select the version of Azure Machine Learning developer platform you are using:

Azure Machine Learning lets you bring data from a local machine or an existing cloud-based storage. In this article you will learn the main data concepts in Azure Machine Learning, including:

- ✓ URIs - A **U**niform **R**esource **I**dentifier that is a reference to a storage location on your local computer or in the cloud that makes it very easy to access data in your jobs.
- ✓ Data asset - Create data assets in your workspace to share with team members, version, and track data lineage.
- ✓ Datastore - Azure Machine Learning Datastores securely keep the connection information to your data storage on Azure, so you don't have to code it in your scripts.
- ✓ MLTable - a method to abstract the schema definition for tabular data so that it is easier for consumers of the data to materialize the table into a Pandas/Dask/Spark dataframe.

URIs

A URI (uniform resource identifier) represents a storage location on your local computer, an attached Datastore, blob/ADLS storage, or a publicly available http(s) location. In addition to local paths (for example: `./path_to_my_data/`), several different protocols are supported for cloud storage locations:

- `http(s)` - Private/Public Azure Blob Storage Locations, or publicly available http(s) location
- `abfs(s)` - Azure Data Lake Storage Gen2 storage location
- `azureml` - An Azure Machine Learning Datastore location

Azure Machine Learning distinguishes two types of URIs:

Data type	Description	Examples

Data type	Description	Examples

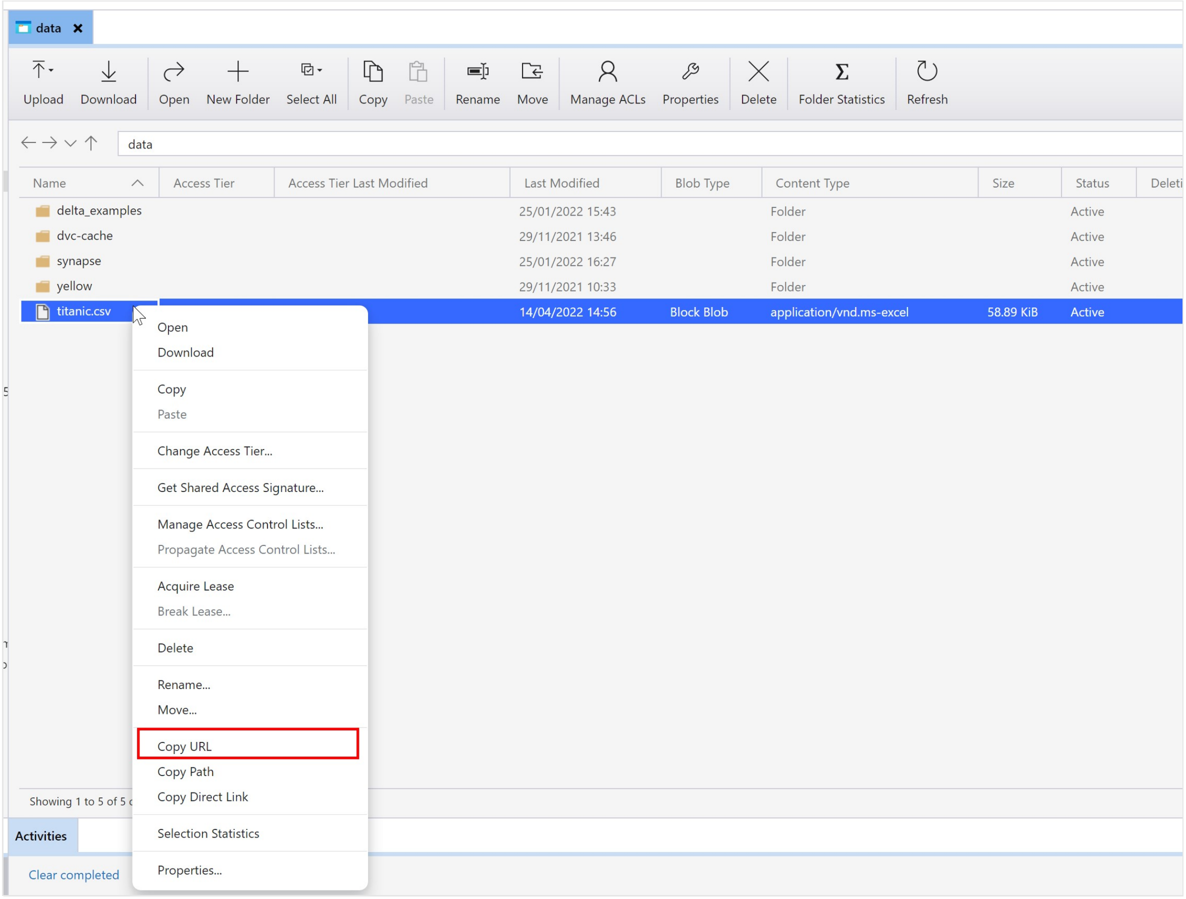
`uri_file`	Refers to a specific **file** location	`https://<account_name>.blob.core.windows.net/<container_name>/<folder>/<file>`
		`azureml://datastores/<datastore_name>/paths/<folder>/<file>`
		`abfss://<file_system>@<account_name>.dfs.core.windows.net/<folder>/<file>`

`uri_folder`	Refers to a specific **folder** location	`https://<account_name>.blob.core.windows.net/<container_name>/<folder>`
		`azureml://datastores/<datastore_name>/paths/<folder>`
		`abfss://<file_system>@<account_name>.dfs.core.windows.net/<folder>/`

URIs are mapped to the filesystem on the compute target, hence using URIs is like using files or folders in the command that consumes/produces them. URIs leverage **identity-based authentication** to connect to storage services with either your Azure Active Directory ID (default) or Managed Identity.

Tip

For data located in an Azure storage account we recommend using the Azure Storage Explorer . You can browse data and obtain the URI for any file/folder by right-selecting **Copy URL**:



Examples

uri_file

Below is an example of a job specification that shows how to access a file from a public blob store. In this example, the job executes the Linux `ls` command.

yml

```yml
# hello-data-uri-file.yml
$schema: https://azuremlschemas.azureedge.net/latest
/commandJob.schema.json
command: |
  ls ${{inputs.my_csv_file}}

inputs:
  my_csv_file:
    type: uri_file
    path: https://azuremlexamples.blob.core.windows.net/datasets/ti-
tanic.csv
environment: azureml:AzureML-sklearn-1.0-ubuntu20.04-py38-cpu@latest
```

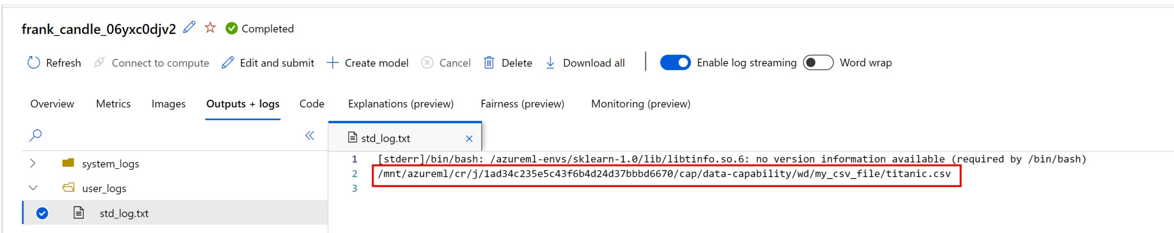
```
compute: azureml:cpu-cluster
```

Create the job using the CLI:

Azure CLI

```
az ml job create --file hello-data-uri-file.yml
```

When the job has completed the user logs will show the standard output of the Linux command `ls ${{inputs.my_csv_file}}`:



Notice that the file has been mapped to the filesystem on the compute target and `${{inputs.my_csv_file}}` resolves to that location.

Data asset

Azure Machine Learning allows you to create and version data assets in a workspace so that other members of your team can easily consume the data asset by using a name/version.

Example usage

Create data asset

To create a data asset, firstly define a data specification in a YAML file that provides a name, type and path for the data:

yml

```
# data-example.yml
$schema: https://azuremlschemas.azureedge.net/latest/data.schema.json
name: <name>
description: <description>
```

```
type: <type> # uri_file, uri_folder, mltable
path: https://<storage_name>.blob.core.windows.net/<container_name>/path
```

Then in the CLI, create the data asset:

Azure CLI

```
az ml data create --file data-example.yml --version 1
```

Datastore

An Azure Machine Learning datastore is a *reference* to an *existing* storage account on Azure. The benefits of creating and using a datastore are:

1. A common and easy-to-use API to interact with different storage types (Blob/Files /ADLS).
2. Easier to discover useful datastores when working as a team.
3. When using credential-based access (service principal/SAS/key), the connection information is secured so you don't have to code it in your scripts.

When you create a datastore with an existing storage account on Azure, you have the choice between two different authentication methods:

- **Credential-based** - authenticate access to the data using a service principal, shared access signature (SAS) token or account key. These credentials can be accessed by users who have *Reader* access to the workspace.
- **Identity-based** - authenticate access to the data using your Azure Active Directory identity or managed identity.

The table below summarizes which cloud-based storage services in Azure can be created as an Azure Machine Learning datastore and what authentication type can be used to access them.

Supported storage service	Credential-based authentication	Identity-based authentication
Azure Blob Container	✓	✓
Azure File Share	✓	

Supported storage service	Credential-based authentication	Identity-based authentication
Azure Data Lake Gen1	✓	✓
Azure Data Lake Gen2	✓	✓

> ⓘ **Note**
>
> The URI format to refer to a file/folder/mltable on a datastore is: `azureml://datastores/<name>/paths/<path>`

MLTable

`mltable` is a way to abstract the schema definition for tabular data so that it is easier for consumers of the data to materialize the table into a Pandas/Dask/Spark dataframe.

> **Tip**
>
> The ideal scenarios to use `mltable` are:
>
> - The schema of your data is complex and/or changes frequently.
> - You only need a subset of data (for example: a sample of rows or files, specific columns, etc).
> - AutoML jobs requiring tabular data.
>
> If your scenario does not fit the above then it is likely that URIs are a more suitable type.

A motivating example

Imagine a scenario where you have many text files in a folder:

text

```
├── my_data
│   ├── file1.txt
│   ├── file1_use_this.txt
```

```
│   ├── file2.txt
│   ├── file2_use_this.txt
.
.
.
│   ├── file1000.txt
│   ├── file1000_use_this.txt
```

Each text file has the following structure:

```text
store_location date zip_code amount x y z noise_col1 noise_col2
Seattle 20/04/2022 12324 123.4 true false true blah blah
.
.
.
London 20/04/2022 XX358YY 156 true true true blah blah
```

Some important features of this data are:

- The data of interest is only in files that have the following suffix: `_use_this.txt` and other file names that don't match should be ignored.
- The date should be represented as a date and not a string.
- The x, y, z columns are booleans, not strings.
- The store location is an index that is useful for generating subsets of data.
- The file is encoded in `ascii` format.
- Every file in the folder contains the same header.
- The first million records for zip_code are numeric but later on you can see they're alphanumeric.
- There are some dummy (noisy) columns in the data that aren't useful for machine learning.

You could materialize the above text files into a dataframe using Pandas and a URI:

```Python
import glob
import datetime
import os
import argparse
import pandas as pd
```

```
parser = argparse.ArgumentParser()
parser.add_argument("--input_folder", type=str)
args = parser.parse_args()

path = os.path.join(args.input_folder, "*_use_this.txt")
files = glob.glob(path)

# create empty list
dfl = []

# dict of column types
col_types = {
    "zip": str,
    "date": datetime.date,
    "x": bool,
    "y": bool,
    "z": bool
}

# enumerate files into a list of dfs
for f in files:
    csv = pd.read_table(
        path=f,
        delimiter=" ",
        header=0,
        usecols=["store_location", "zip_code", "date", "amount", "x", "y",
"z"],
        dtype=col_types,
        encoding='ascii'
    )
    dfl.append(csv)

# concatenate the list of dataframes
df = pd.concat(dfl)
# set the index column
df.index_columns("store_location")
```

However, it will be the responsibility of the *consumer* of the data asset to parse the schema into a dataframe. In the scenario defined above, that means the consumers will need to independently ascertain the Python code to materialize the data into a dataframe.

Passing responsibility to the consumer of the data asset will cause problems when:

- **The schema changes (for example, a column name changes):** All consumers of the data must update their Python code independently. Other examples can be type changes, columns being added/removed, encoding change, etc.

- **The data size increases** - If the data gets too large for Pandas to process, then all the consumers of the data need to switch to a more scalable library (PySpark/Dask).

Under the above two conditions, `mltable` can help because it enables the creator of the data asset to define the schema in a single file and the consumers can materialize the data into a dataframe easily without needing to write Python code to parse the schema. For the above example, the creator of the data asset defines an MLTable file **in the same directory** as the data:

text

```text
├── my_data
│   ├── MLTable
│   ├── file1.txt
│   ├── file1_use_this.txt
.
.
.
```

The MLTable file has the following definition that specifies how the data should be processed into a dataframe:

YAML

```
type: mltable

paths:
    - pattern: ./*_use_this.txt

traits:
    - index_columns: store_location

transformations:
    - read_delimited:
        encoding: ascii
        header: all_files_same_headers
        delimiter: " "
    - keep_columns: ["store_location", "zip_code", "date", "amount", "x",
"y", "z"]
    - convert_column_types:
        - columns: ["x", "y", "z"]
          to_type: boolean
        - columns: "date"
          to_type: datetime
```

The consumers can read the data into dataframe using three lines of Python code:

```python
import mltable

tbl = mltable.load("./my_data")
df = tbl.to_pandas_dataframe()
```

If the schema of the data changes, then it can be updated in a single place (the MLTable file) rather than having to make code changes in multiple places.

Just like `uri_file` and `uri_folder`, you can create a data asset with `mltable` types.

Next steps

- Install and set up the CLI (v2)
- Create datastores
- Create data assets
- Read and write data in a job
- Data administration