Azure Data Scientist Associate Certification Notes

Notes and Procedures

Bruttocao, Marco (858067)*

Contents

Build and operate machine learning solutions with Azure Machine Learning **2**

Introduction to the Azure Machine Learning SDK 2

Installing the Azure Machine Learning SDK for Python 3

The Azure Machine Learning CLI Extension 4

Compute Instances 4

Azure Machine Learning experiments 4

Train a machine learning model with Azure Machine Learning 5

Work with Data in Azure Machine Learning 7

*858067@stud.unive.it

Build and operate machine learning solutions with Azure Machine Learning

Introduction to the Azure Machine Learning SDK

A workspace is a context for the experiments, data, compute targets, and other assets associated with a machine learning workload (Workspaces are Azure resources, and as such they are defined within a resource group in an Azure subscription). 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

The Azure resources created alongside a workspace include:

- A storage account
- 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

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
- Use the Azure Command Line Interface (CLI) with the Azure Machine Learning CLI extension
- Create an Azure Resource Manager template

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:

- Designer, a drag and drop interface for "no code" machine learning model development.
- Automated Machine Learning, 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.

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:

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

Installing the Azure Machine Learning SDK for Python

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. 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
```

SDK Documentation

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"
}
```

> 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.

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.

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.

To install the Azure Machine Learning CLI extension, you must first install the Azure CLI.

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.

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 compute instance image that provides the compute specification you need, from small CPU-only VMs to large GPU-enabled workstations. Because compute instances are hosted in Azure, you only pay for the compute resources when they are running; so you can create a compute instance 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 compute instance.

Azure Machine Learning experiments

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. 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. 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. Experiments are most useful when they produce metrics and outputs that can be tracked across runs. 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. You can upload local files to the run's outputs folder by using the Run object's upload_file method in your experiment code.

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 as scale. 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.

```
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()
```

Train a machine learning model with Azure Machine Learning

You can use a ScriptRunConfig to run a script-based experiment that trains a machine learning model. When using an experiment to train a model, your script should save the trained model in the outputs folder.

```
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 = diabetes[['Feature1','Feature2','Feature3']].values, diabetes['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.

```
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()
```

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:

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

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.

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.

Work with Data in Azure Machine Learning