What is the Azure Machine Learning SDK for Python?

Article • 11/23/2021 • 17 minutes to read

Data scientists and AI developers use the Azure Machine Learning SDK for Python to build and run machine learning workflows with the Azure Machine Learning service. You can interact with the service in any Python environment, including Jupyter Notebooks, Visual Studio Code, or your favorite Python IDE.

Key areas of the SDK include:

- Explore, prepare and manage the lifecycle of your datasets used in machine learning experiments.
- Manage cloud resources for monitoring, logging, and organizing your machine learning experiments.
- Train models either locally or by using cloud resources, including GPU-accelerated model training.
- Use automated machine learning, which accepts configuration parameters and training data. It automatically iterates through algorithms and hyperparameter settings to find the best model for running predictions.
- Deploy web services to convert your trained models into RESTful services that can be consumed in any application.

For a step-by-step walkthrough of how to get started, try the tutorial.

The following sections are overviews of some of the most important classes in the SDK, and common design patterns for using them. To get the SDK, see the installation guide.

Stable vs experimental

The Azure Machine Learning SDK for Python provides both stable and experimental features in the same SDK.

Feature/capability status	Description
Stable features	**Production ready**

Feature/capability status	Description
	These features are recommended for most use cases and production environments. They are updated less frequently then experimental features.
Experimental features	**Developmental** These features are newly developed capabilities & updates that may not be ready or fully tested for production usage. While the features are typically functional, they can include some breaking changes. Experimental features are used to iron out SDK breaking bugs, and will only receive updates for the duration of the testing period. Experimental features are also referred to as features that are in **preview**. As the name indicates, the experimental (preview) features are for experimenting and is **not considered bug free or stable**. For this reason, we only recommend experimental features to advanced users who wish to try out early versions of capabilities and updates, and intend to participate in the reporting of bugs and glitches.

Experimental features are labelled by a note section in the SDK reference and denoted by text such as, **(preview)** throughout Azure Machine Learning documentation.

Workspace

Namespace: `azureml.core.workspace.Workspace`

The `Workspace` class is a foundational resource in the cloud that you use to experiment, train, and deploy machine learning models. It ties your Azure subscription and resource group to an easily consumed object.

View all parameters of the create Workspace method to reuse existing instances (**Storage, Key Vault, App-Insights, and Azure Container Registry-ACR**) as well as modify additional settings such as private endpoint configuration and compute target.

Import the class and create a new workspace by using the following code. Set `create_resource_group` to `False` if you have a previously existing Azure resource group that you want to use for the workspace. Some functions might prompt for Azure authentication credentials.

Python

```python
from azureml.core import Workspace
ws = Workspace.create(name='myworkspace',
                      subscription_id='<azure-subscription-id>',
                      resource_group='myresourcegroup',
                      create_resource_group=True,
                      location='eastus2'
                     )
```

Use the same workspace in multiple environments by first writing it to a configuration JSON file. This saves your subscription, resource, and workspace name data.

Python

```python
ws.write_config(path="./file-path", file_name="ws_config.json")
```

Load your workspace by reading the configuration file.

Python

```python
from azureml.core import Workspace
ws_other_environment = Workspace.from_config(path="./file-
path/ws_config.json")
```

Alternatively, use the static `get()` method to load an existing workspace without using configuration files.

Python

```python
from azureml.core import Workspace
ws = Workspace.get(name="myworkspace", subscription_id='<azure-subscription-
id>', resource_group='myresourcegroup')
```

The variable `ws` represents a `Workspace` object in the following code examples.

Experiment

Namespace: `azureml.core.experiment.Experiment`

The `Experiment` class is another foundational cloud resource that represents a collection of trials (individual model runs). The following code fetches an `Experiment` object from within `Workspace` by name, or it creates a new `Experiment` object if the name doesn't exist.

Python

```python
from azureml.core.experiment import Experiment
experiment = Experiment(workspace=ws, name='test-experiment')
```

Run the following code to get a list of all `Experiment` objects contained in `Workspace`.

Python

```python
list_experiments = Experiment.list(ws)
```

Use the `get_runs` function to retrieve a list of `Run` objects (trials) from `Experiment`. The following code retrieves the runs and prints each run ID.

Python

```python
list_runs = experiment.get_runs()
for run in list_runs:
    print(run.id)
```

There are two ways to execute an experiment trial. If you're interactively experimenting in a Jupyter notebook, use the `start_logging` function. If you're submitting an experiment from a standard Python environment, use the `submit` function. Both functions return a `Run` object. The `experiment` variable represents an `Experiment` object in the following code examples.

Run

Namespace: `azureml.core.run.Run`

A run represents a single trial of an experiment. `Run` is the object that you use to monitor the asynchronous execution of a trial, store the output of the trial, analyze results, and access generated artifacts. You use `Run` inside your experimentation code to log metrics and artifacts to the Run History service. Functionality includes:

- Storing and retrieving metrics and data.
- Using tags and the child hierarchy for easy lookup of past runs.
- Registering stored model files for deployment.
- Storing, modifying, and retrieving properties of a run.

Create a `Run` object by submitting an `Experiment` object with a run configuration object. Use the `tags` parameter to attach custom categories and labels to your runs. You can easily find and retrieve them later from `Experiment`.

Python

```python
tags = {"prod": "phase-1-model-tests"}
run = experiment.submit(config=your_config_object, tags=tags)
```

Use the static `list` function to get a list of all `Run` objects from `Experiment`. Specify the `tags` parameter to filter by your previously created tag.

Python

```python
from azureml.core.run import Run
filtered_list_runs = Run.list(experiment, tags=tags)
```

Use the `get_details` function to retrieve the detailed output for the run.

Python

```python
run_details = run.get_details()
```

Output for this function is a dictionary that includes:

- Run ID
- Status
- Start and end time
- Compute target (local versus cloud)
- Dependencies and versions used in the run
- Training-specific data (differs depending on model type)

For more examples of how to configure and monitor runs, see the how-to.

Model

Namespace: `azureml.core.model.Model`

The `Model` class is used for working with cloud representations of machine learning models. Methods help you transfer models between local development environments and the `Workspace` object in the cloud.

You can use model registration to store and version your models in the Azure cloud, in your workspace. Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. Azure Machine Learning supports any model that can be loaded through Python 3, not just Azure Machine Learning models.

The following example shows how to build a simple local classification model with `scikit-learn`, register the model in `Workspace`, and download the model from the cloud.

Create a simple classifier, `clf`, to predict customer churn based on their age. Then dump the model to a `.pkl` file in the same directory.

Python

```python
from sklearn import svm
import joblib
import numpy as np

# customer ages
X_train = np.array([50, 17, 35, 23, 28, 40, 31, 29, 19, 62])
X_train = X_train.reshape(-1, 1)
# churn y/n
y_train = ["yes", "no", "no", "no", "yes", "yes", "yes", "no", "no", "yes"]

clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(X_train, y_train)

joblib.dump(value=clf, filename="churn-model.pkl")
```

Use the `register` function to register the model in your workspace. Specify the local model path and the model name. Registering the same name more than once will create a new version.

Python

```python
from azureml.core.model import Model

model = Model.register(workspace=ws, model_path="churn-model.pkl",
model_name="churn-model-test")
```

Now that the model is registered in your workspace, it's easy to manage, download, and organize your models. To retrieve a model (for example, in another environment) object from `Workspace`, use the class constructor and specify the model name and any optional parameters. Then, use the `download` function to download the model, including the cloud folder structure.

Python

```python
from azureml.core.model import Model
import os

model = Model(workspace=ws, name="churn-model-test")
model.download(target_dir=os.getcwd())
```

Use the `delete` function to remove the model from `Workspace`.

Python

```python
model.delete()
```

After you have a registered model, deploying it as a web service is a straightforward process. First you create and register an image. This step configures the Python environment and its dependencies, along with a script to define the web service request and response formats. After you create an image, you build a deploy configuration that sets the CPU cores and memory parameters for the compute target. You then attach your image.

ComputeTarget, RunConfiguration, and ScriptRunConfig

Namespace: `azureml.core.compute.ComputeTarget`

Namespace: `azureml.core.runconfig.RunConfiguration`

Namespace: `azureml.core.script_run_config.ScriptRunConfig`

The `ComputeTarget` class is the abstract parent class for creating and managing compute targets. A compute target represents a variety of resources where you can train your machine learning models. A compute target can be either a local machine or a cloud resource, such as Azure Machine Learning Compute, Azure HDInsight, or a remote virtual machine.

Use compute targets to take advantage of powerful virtual machines for model training, and set up either persistent compute targets or temporary runtime-invoked targets. For a comprehensive guide on setting up and managing compute targets, see the how-to.

The following code shows a simple example of setting up an `AmlCompute` (child class of `ComputeTarget`) target. This target creates a runtime remote compute resource in your `Workspace` object. The resource scales automatically when a job is submitted. It's deleted automatically when the run finishes.

Reuse the simple `scikit-learn` churn model and build it into its own file, `train.py`, in the current directory. At the end of the file, create a new directory called `outputs`. This step creates a directory in the cloud (your workspace) to store your trained model that `joblib.dump()` serialized.

Python

```python
# train.py

from sklearn import svm
import numpy as np
import joblib
import os

# customer ages
X_train = np.array([50, 17, 35, 23, 28, 40, 31, 29, 19, 62])
X_train = X_train.reshape(-1, 1)
# churn y/n
y_train = ["yes", "no", "no", "no", "yes", "yes", "yes", "no", "no", "yes"]

clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(X_train, y_train)

os.makedirs("outputs", exist_ok=True)
joblib.dump(value=clf, filename="outputs/churn-model.pkl")
```

Next you create the compute target by instantiating a `RunConfiguration` object and setting the type and size. This example uses the smallest resource size (1 CPU core, 3.5 GB of memory). The `list_vms` variable contains a list of supported virtual machines and their sizes.

Python

```python
from azureml.core.runconfig import RunConfiguration
from azureml.core.compute import AmlCompute
list_vms = AmlCompute.supported_vmsizes(workspace=ws)

compute_config = RunConfiguration()
compute_config.target = "amlcompute"
compute_config.amlcompute.vm_size = "STANDARD_D1_V2"
```

Create dependencies for the remote compute resource's Python environment by using the `CondaDependencies` class. The `train.py` file is using `scikit-learn` and `numpy`, which need to be installed in the environment. You can also specify versions of dependencies. Use the `dependencies` object to set the environment in `compute_config`.

Python

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

dependencies = CondaDependencies()
dependencies.add_pip_package("scikit-learn")
dependencies.add_pip_package("numpy==1.15.4")
compute_config.environment.python.conda_dependencies = dependencies
```

Now you're ready to submit the experiment. Use the `ScriptRunConfig` class to attach the compute target configuration, and to specify the path/file to the training script `train.py`. Submit the experiment by specifying the `config` parameter of the `submit()` function. Call `wait_for_completion` on the resulting run to see asynchronous run output as the environment is initialized and the model is trained.

⚠ **Warning**

The following are limitations around specific characters when used in `ScriptRunConfig` parameters:

- The `"`, `$`, `;`, and `\` characters are escaped by the back end, as they are considered reserved characters for separating bash commands.

- The `(`, `)`, `%`, `!`, `^`, `<`, `>`, `&`, and `|` characters are escaped for local runs on Windows.

Python

```python
from azureml.core.experiment import Experiment
from azureml.core import ScriptRunConfig

script_run_config = ScriptRunConfig(source_directory=os.getcwd(),
script="train.py", run_config=compute_config)
experiment = Experiment(workspace=ws, name="compute_target_test")
run = experiment.submit(config=script_run_config)
run.wait_for_completion(show_output=True)
```

After the run finishes, the trained model file `churn-model.pkl` is available in your workspace.

Environment

Namespace: `azureml.core.environment`

Azure Machine Learning environments specify the Python packages, environment variables, and software settings around your training and scoring scripts. In addition to Python, you can also configure PySpark, Docker and R for environments. Internally, environments result in Docker images that are used to run the training and scoring processes on the compute target. The environments are managed and versioned entities within your Machine Learning workspace that enable reproducible, auditable, and portable machine learning workflows across a variety of compute targets and compute types.

You can use an `Environment` object to:

- Develop your training script.
- Reuse the same environment on Azure Machine Learning Compute for model training at scale.
- Deploy your model with that same environment without being tied to a specific compute type.

The following code imports the `Environment` class from the SDK and to instantiates an environment object.

Python

```python
from azureml.core.environment import Environment
Environment(name="myenv")
```

Add packages to an environment by using Conda, pip, or private wheel files. Specify each package dependency by using the CondaDependency class to add it to the environment's `PythonSection`.

The following example adds to the environment. It adds version 1.17.0 of `numpy`. It also adds the `pillow` package to the environment, `myenv`. The example uses the add_conda_package() method and the add_pip_package() method, respectively.

Python

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

myenv = Environment(name="myenv")
conda_dep = CondaDependencies()

# Installs numpy version 1.17.0 conda package
conda_dep.add_conda_package("numpy==1.17.0")

# Installs pillow package
conda_dep.add_pip_package("pillow")

# Adds dependencies to PythonSection of myenv
myenv.python.conda_dependencies=conda_dep
```

To submit a training run, you need to combine your environment, compute target, and your training Python script into a run configuration. This configuration is a wrapper object that's used for submitting runs.

When you submit a training run, the building of a new environment can take several minutes. The duration depends on the size of the required dependencies. The environments are cached by the service. So as long as the environment definition remains unchanged, you incur the full setup time only once.

The following example shows where you would use ScriptRunConfig as your wrapper object.

Python

```python
from azureml.core import ScriptRunConfig, Experiment
from azureml.core.environment import Environment

exp = Experiment(name="myexp", workspace = ws)
# Instantiate environment
myenv = Environment(name="myenv")

# Add training script to run config
runconfig = ScriptRunConfig(source_directory=".", script="train.py")

# Attach compute target to run config
runconfig.run_config.target = "local"

# Attach environment to run config
runconfig.run_config.environment = myenv

# Submit run
run = exp.submit(runconfig)
```

If you don't specify an environment in your run configuration before you submit the run, then a default environment is created for you.

See the Model deploy section to use environments to deploy a web service.

Pipeline, PythonScriptStep

Namespace: `azureml.pipeline.core.pipeline.Pipeline`

Namespace: `azureml.pipeline.steps.python_script_step.PythonScriptStep`

An Azure Machine Learning pipeline is an automated workflow of a complete machine learning task. Subtasks are encapsulated as a series of steps within the pipeline. An Azure Machine Learning pipeline can be as simple as one step that calls a Python script. Pipelines include functionality for:

- Data preparation including importing, validating and cleaning, munging and transformation, normalization, and staging
- Training configuration including parameterizing arguments, filepaths, and logging

/ reporting configurations
- Training and validating efficiently and repeatably, which might include specifying specific data subsets, different hardware compute resources, distributed processing, and progress monitoring
- Deployment, including versioning, scaling, provisioning, and access control
- Publishing a pipeline to a REST endpoint to rerun from any HTTP library

A `PythonScriptStep` is a basic, built-in step to run a Python Script on a compute target. It takes a script name and other optional parameters like arguments for the script, compute target, inputs and outputs. The following code is a simple example of a `PythonScriptStep`. For an example of a `train.py` script, see the tutorial sub-section.

Python

```python
from azureml.pipeline.steps import PythonScriptStep

train_step = PythonScriptStep(
    script_name="train.py",
    arguments=["--input", blob_input_data, "--output", output_data1],
    inputs=[blob_input_data],
    outputs=[output_data1],
    compute_target=compute_target,
    source_directory=project_folder
)
```

After at least one step has been created, steps can be linked together and published as a simple automated pipeline.

Python

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

pipeline = Pipeline(workspace=ws, steps=[train_step])
pipeline_run = experiment.submit(pipeline)
```

For a comprehensive example of building a pipeline workflow, follow the advanced tutorial.

Pattern for creating and using pipelines

An Azure Machine Learning pipeline is associated with an Azure Machine Learning workspace and a pipeline step is associated with a compute target available within that workspace. For more information, see this article about workspaces or this explanation of compute targets.

A common pattern for pipeline steps is:

1. Specify workspace, compute, and storage
2. Configure your input and output data using
   a. Dataset which makes available an existing Azure datastore
   b. PipelineDataset which encapsulates typed tabular data
   c. PipelineData which is used for intermediate file or directory data written by one step and intended to be consumed by another
3. Define one or more pipeline steps
4. Instantiate a pipeline using your workspace and steps
5. Create an experiment to which you submit the pipeline
6. Monitor the experiment results

This notebook is a good example of this pattern. job

For more information about Azure Machine Learning Pipelines, and in particular how they are different from other types of pipelines, see this article.

AutoMLConfig

Namespace: `azureml.train.automl.automlconfig.AutoMLConfig`

Use the `AutoMLConfig` class to configure parameters for automated machine learning training. Automated machine learning iterates over many combinations of machine learning algorithms and hyperparameter settings. It then finds the best-fit model based on your chosen accuracy metric. Configuration allows for specifying:

- Task type (classification, regression, forecasting)
- Number of algorithm iterations and maximum time per iteration
- Accuracy metric to optimize
- Algorithms to blocklist/allowlist
- Number of cross-validations
- Compute targets
- Training data

ⓘ **Note**

Use the `automl` extra in your installation to use automated machine learning.

For detailed guides and examples of setting up automated machine learning experiments, see the tutorial and how-to.

The following code illustrates building an automated machine learning configuration object for a classification model, and using it when you're submitting an experiment.

Python

```python
from azureml.train.automl import AutoMLConfig

automl_config = AutoMLConfig(task="classification",
                             X=your_training_features,
                             y=your_training_labels,
                             iterations=30,
                             iteration_timeout_minutes=5,
                             primary_metric="AUC_weighted",
                             n_cross_validations=5
                            )
```

Use the `automl_config` object to submit an experiment.

Python

```python
from azureml.core.experiment import Experiment

experiment = Experiment(ws, "automl_test_experiment")
run = experiment.submit(config=automl_config, show_output=True)
```

After you submit the experiment, output shows the training accuracy for each iteration as it finishes. After the run is finished, an `AutoMLRun` object (which extends the `Run` class) is returned. Get the best-fit model by using the `get_output()` function to return a `Model` object.

Python

```python
best_model = run.get_output()
y_predict = best_model.predict(X_test)
```

Model deploy

Namespace: `azureml.core.model.InferenceConfig`

Namespace: `azureml.core.webservice.webservice.Webservice`

The `InferenceConfig` class is for configuration settings that describe the environment needed to host the model and web service.

`Webservice` is the abstract parent class for creating and deploying web services for your models. For a detailed guide on preparing for model deployment and deploying web services, see this how-to.

You can use environments when you deploy your model as a web service. Environments enable a reproducible, connected workflow where you can deploy your model using the same libraries in both your training compute and your inference compute. Internally, environments are implemented as Docker images. You can use either images provided by Microsoft, or use your own custom Docker images. If you were previously using the `ContainerImage` class for your deployment, see the DockerSection class for accomplishing a similar workflow with environments.

To deploy a web service, combine the environment, inference compute, scoring script, and registered model in your deployment object, deploy().

The following example, assumes you already completed a training run using environment, `myenv`, and want to deploy that model to Azure Container Instances.

Python

```python
from azureml.core.model import InferenceConfig, Model
from azureml.core.webservice import AciWebservice, Webservice

# Register the model to deploy
model = run.register_model(model_name = "mymodel", model_path = "out-
puts/model.pkl")

# Combine scoring script & environment in Inference configuration
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=myenv)

# Set deployment configuration
deployment_config = AciWebservice.deploy_configuration(cpu_cores = 1,
                                                       memory_gb = 1)
```

```python
# Define the model, inference, & deployment configuration and web service
name and location to deploy
service = Model.deploy(workspace = ws,
                       name = "my_web_service",
                       models = [model],
                       inference_config = inference_config,
                       deployment_config = deployment_config)
```

This example creates an Azure Container Instances web service, which is best for small-scale testing and quick deployments. To deploy your model as a production-scale web service, use Azure Kubernetes Service (AKS). For more information, see AksCompute class.

Dataset

Namespace: `azureml.core.dataset.Dataset`

Namespace: `azureml.data.file_dataset.FileDataset`

Namespace: `azureml.data.tabular_dataset.TabularDataset`

The `Dataset` class is a foundational resource for exploring and managing data within Azure Machine Learning. You can explore your data with summary statistics, and save the Dataset to your AML workspace to get versioning and reproducibility capabilities. Datasets are easily consumed by models during training. For detailed usage examples, see the how-to guide.

- `TabularDataset` represents data in a tabular format created by parsing a file or list of files.
- `FileDataset` references single or multiple files in datastores or from public URLs.

The following example shows how to create a TabularDataset pointing to a single path in a datastore.

```python
from azureml.core import Dataset

dataset = Dataset.Tabular.from_delimited_files(path = [(datastore, 'train-
dataset/tabular/iris.csv')])
dataset.take(3).to_pandas_dataframe()
```

The following example shows how to create a `FileDataset` referencing multiple file URLs.

Python

```python
from azureml.core.dataset import Dataset

url_paths = [
            'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
            'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
            ]
dataset = Dataset.File.from_files(path=url_paths)
```

Next steps

Try these next steps to learn how to use the Azure Machine Learning SDK for Python:

- Follow the tutorial to learn how to build, train, and deploy a model in Python.
- Look up classes and modules in the reference documentation on this site by using the table of contents on the left.