Create a responsible AI dashboard to evaluate your models

When you compare and evaluate your machine learning models, you'll want to review more than just their performance metric. Azure Machine Learning allows you to create responsible AI dashboard to explore how the model performs on different cohorts of the data.

Prepare the data

To create the responsible AI dashboard, you'll need a training and test dataset, stored as Parquet files and registered as data assets. MLTable data assets referencing the parquet files.

The data is currently stored as CSV files. Let's convert them to Parquet files.

In [ ]:

```
import pandas as pd

# read training and test dataset
df_training = pd.read_csv("train-data/diabetes.csv")
df_test = pd.read_csv("test-data/diabetes-test.csv")

# display the first few rows of the training dataset
df_training.head()
```

--- Training and test dataset,
- stored as Parquet files and
- registered as MLtable data assets
- referencing the parquet file

--- Pipeline with prebuilt components that are registered by default in your workspace.
- A component at start- to start the dashboard
- A component at end - to gather all insights

In [ ]:

```
import pyarrow as pa
import pyarrow.parquet as pq

# convert data to table
table_training = pa.Table.from_pandas(df_training)
table_test = pa.Table.from_pandas(df_test)

# write tables out to parquet
pq.write_table(table_training, "train-data/diabetes-training.parquet", version="1.0")
pq.write_table(table_test, "test-data/diabetes-test.parquet", version="1.0")
```

Convert Data to parquet

Before you continue

You'll need the latest version of the **azureml-ai-ml** package to run the code in this notebook. Run the cell below to verify that it is installed.

> **Note:** If the **azure-ai-ml** package is not installed, run `pip install azure-ai-ml` to install it.

In [ ]:

```
pip show azure-ai-ml
```

Connect to your workspace

With the required SDK packages installed, now you're ready to connect to your workspace.

To connect to a workspace, we need identifier parameters - a subscription ID, resource group name, and workspace name. Since you're working with a compute instance, managed by Azure Machine Learning, you can use the default values to connect to the workspace.

In [ ]:

```
from azure.identity import DefaultAzureCredential, InteractiveBrowserCredential
from azure.ai.ml import MLClient

try:
    credential = DefaultAzureCredential()
    # Check if given credential can get token successfully.
    credential.get_token("https://management.azure.com/.default")
except Exception as ex:
    # Fall back to InteractiveBrowserCredential in case DefaultAzureCredential not work
    credential = InteractiveBrowserCredential()
```

In [ ]:

```
# Get a handle to workspace
ml_client = MLClient.from_config(credential=credential)
```

Create the data assets

To create the responsible AI dashboard, you need to register the training and testing datasets as MLtable data assets. The MLtable data assets reference the Parquet files you created earlier.

In [ ]:

```
train_data_path = "train-data/"
test_data_path = "test-data/"
data_version = "1"
```

In [ ]:

```
from azure.ai.ml.entities import Data
from azure.ai.ml.constants import AssetTypes

input_train_data = "diabetes_train_mltable"
input_test_data = "diabetes_test_mltable"

try:
    # Try getting data already registered in workspace
    train_data = ml_client.data.get(
        name=input_train_data,
        version=data_version,
    )
    test_data = ml_client.data.get(
        name=input_test_data,
        version=data_version,
    )
except Exception as e:
    train_data = Data(
        path=train_data_path,
        type=AssetTypes.MLTABLE,
        description="RAI diabetes training data",
        name=input_train_data,
        version=data_version,
    )
    ml_client.data.create_or_update(train_data)

    test_data = Data(
        path=test_data_path,
        type=AssetTypes.MLTABLE,
        description="RAI diabetes test data",
        name=input_test_data,
        version=data_version,
    )
    ml_client.data.create_or_update(test_data)
```

Defining the data as MLTable

Build the pipeline to create the responsible AI dashboard

To create the dashboard, you'll build a pipeline with prebuilt components that are registered by default in the

Azure Machine Learning workspace.

In [ ]:

```
# Get handle to azureml registry for the RAI built in components
registry_name = "azureml"
ml_client_registry = MLClient(
    credential=credential,
    subscription_id=subscription_id,
    resource_group_name=resource_group,
    registry_name=registry_name,
)
print(ml_client_registry)
```

Handle to registry for RAI builder

Register the model

A machine learning model has already been trained for you. The model predicts whether a patient has diabetes. All model files are stored in the `model` folder.

Register the model by pointing to the `model` folder and its contents.

In [ ]:

```
from azure.ai.ml.entities import Model
from azure.ai.ml.constants import AssetTypes

file_model = Model(
    path="model",
    type=AssetTypes.MLFLOW_MODEL,
    name="local-mlflow-diabetes",
    description="Model created from local file.",
)
model = ml_client.models.create_or_update(file_model)
```

Register the model

Build the pipeline

To create the responsible AI dashboard, you'll create a pipeline using the prebuilt components. You can choose which components to use, and which features you want to include in your dashboard. You'll create a dashboard that includes error analysis and model interpretability.

Next to the responsible AI features, a pipeline to build a dashboard needs to include a component at the start to construct the dashboard, and a component at the end to gather all generated insights.

In [ ]:

```
model_name = model.name
expected_model_id = f"{model_name}:1"
azureml_model_id = f"azureml:{expected_model_id}"
```

In [ ]:

Retrieve RAI components from mlclient_registry

```
label = "latest"

rai_constructor_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_insight_constructor", label=label
)

# we get latest version and use the same version for all components
version = rai_constructor_component.version
print("The current version of RAI built-in components is: " + version)

rai_erroranalysis_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_erroranalysis", version=version
)

rai_explanation_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_explanation", version=version
)

rai_gather_component = ml_client_registry.components.get(
    name="microsoft_azureml_rai_tabular_insight_gather", version=version
)
```

When you've retrieved all components you want to use, you can build the pipeline and connect the components in the appropriate order:

1. **Construct the dashboard.**
2. **Add error analysis.**
3. **Add explanations.**
4. **Gather all insights and visualize them in the dashboard.**

In [ ]:

Define the rai_decision pipeline

```
from azure.ai.ml import Input, dsl
from azure.ai.ml.constants import AssetTypes

compute_name = "aml-cluster"

@dsl.pipeline(
    compute=compute_name,
    description="RAI insights on diabetes data",
    experiment_name=f"RAI_insights_{model_name}",
)
def rai_decision_pipeline(
    target_column_name, train_data, test_data
):
    # Initiate the RAIInsights
    create_rai_job = rai_constructor_component(
        title="RAI dashboard diabetes",
        task_type="classification",
        model_info=expected_model_id,
        model_input=Input(type=AssetTypes.MLFLOW_MODEL, path=azureml_model_id),
        train_dataset=train_data,
        test_dataset=test_data,
        target_column_name=target_column_name,
    )
    create_rai_job.set_limits(timeout=30)

    # Add error analysis
    error_job = rai_erroranalysis_component(
        rai_insights_dashboard=create_rai_job.outputs.rai_insights_dashboard,
    )
    error_job.set_limits(timeout=10)

    # Add explanations
    explanation_job = rai_explanation_component(
        rai_insights_dashboard=create_rai_job.outputs.rai_insights_dashboard,
        comment="add explanation",
    )
    explanation_job.set_limits(timeout=10)

    # Combine everything
    rai_gather_job = rai_gather_component(
        constructor=create_rai_job.outputs.rai_insights_dashboard,
        insight_3=error_job.outputs.error_analysis,
        insight_4=explanation_job.outputs.explanation,
    )
    rai_gather_job.set_limits(timeout=10)

    rai_gather_job.outputs.dashboard.mode = "upload"

    return {
        "dashboard": rai_gather_job.outputs.dashboard,
    }
```

Now the pipeline has been built, you need to define the two necessary inputs: the training and test dataset.

In [ ]:

```
from azure.ai.ml import Input
target_feature = "Diabetic"

diabetes_train_pq = Input(
    type="mltable",
    path=f"azureml:{input_train_data}:{data_version}",
    mode="download",
)
diabetes_test_pq = Input(
    type="mltable",
    path=f"azureml:{input_test_data}:{data_version}",
    mode="download",
)
```

Define the inputs for pipeline

Finally, we'll put everything together: assign the inputs to the pipeline and set the target column (the predicted label).

In [ ]:

Assign inputs to pipeline and get insights_pipeline

```
import uuid
from azure.ai.ml import Output

# Pipeline to construct the RAI Insights
insights_pipeline_job = rai_decision_pipeline(
    target_column_name="Diabetic",
    train_data=diabetes_train_pq,
    test_data=diabetes_test_pq,
)

# Workaround to enable the download
rand_path = str(uuid.uuid4())
insights_pipeline_job.outputs.dashboard = Output(
    path=f"azureml://datastores/workspaceblobstore/paths/{rand_path}/dashboard/",
    mode="upload",
    type="uri_folder",
)
```

Run the pipeline

When you've successfully built the pipeline, you can submit it. The following code will submit the pipeline and check the status of the pipeline. You can also view the pipeline's status in the studio.

In [ ]:

```
from azure.ai.ml.entities import PipelineJob
from IPython.core.display import HTML
from IPython.display import display
import time

def submit_and_wait(ml_client, pipeline_job) -> PipelineJob:
    created_job = ml_client.jobs.create_or_update(pipeline_job)
    assert created_job is not None

    print("Pipeline job can be accessed in the following URL:")
    display(HTML('<a href="{0}">{0}</a>'.format(created_job.studio_url)))

    while created_job.status not in [
        "Completed",
        "Failed",
        "Canceled",
        "NotResponding",
    ]:
        time.sleep(30)
        created_job = ml_client.jobs.get(created_job.name)
        print("Latest status : {0}".format(created_job.status))
```

Submit and Run pipeline

```
    assert created_job.status == "Completed"
    return created_job


# This is the actual submission
insights_job = submit_and_wait(ml_client, insights_pipeline_job)
```

When the pipeline is completed, you can review the dashboard in the studio.