In this workshop you’ll be learning the end-to-end machine learning process that involves data preparation, training a model, registering the model and debug them. First you will go through the process of training the model locally. Then you will learn how to make your machine learning model training process more dynamic, managable and scalable in the cloud using Azure Machine Learning service. For the workshop we’ll be using the [UCI hospital diabetes dataset](https://archive.ics.uci.edu/ml/machine-learning-databases/00296/) to train a classification model using the Scikit-Learn framework. The model will predict whether or not a diabetic patient will be readmitted back to a hospital with 30 days of being discharged. To prep the data to be used for training our model we are going to review our dataset to cleanse any redundant data, missing values, reformat data or delete any irrelevant data that will be used for our use case. After the data has been clean, we’ll train.

Task #1: Data Prep

For this module, you’ll load the “1-dataprep-and-training-local.ipynb” jupyter notebook.

The first thing we need is to read the hospital diabetes data into a dataframe that will allow us to visualize and perform data manipulation. To do this will be using the read\_csv function from the “pandas” library.

df = pd.read\_csv('diabetic\_data.csv')

Next, take a look at the size of the data to verify that it will be sufficient for our training. As you can see 101,766 is a very large data set.

Let’s display the dataframe to see what kind of diabetes is in the hospital data. As you can see it have the patient race, gender, age, weight, their prior hospitalization inform, prescribed medication and whether or not they were readmitted back to the hospital.

Checking for special characters

One of the invalid data that stands out from the Weight column is the “?” special characters that are used as fillers in our dataset. We’ll loop through the dataset to identify columns that have this special character.

print(df.loc[: , (df == '?').any()])

The output shows that that race, weight, payer\_code, medical\_specialty, diag\_1, diag\_2, and diag\_3 are the column with missing values that have the special character. To fix this we’ll change the character to null by using the numpy NAN function.

df = df.replace('?', np.NaN)

Checking for null values

print(df.isna().any())

* When we run the “isna” function on the dataframe, we can see that there are 7 columns with null values in our dataset. Depending on the number of null values that a column has, we’ll keep or delete the column. As we count the columns with a high number of null values from the weight column. Although, Weight is liking to impact a diabetes patient’s readmission status, a majority of it’s values are null. Race is another column that has 12% of its values missing. This is not a signification number; we can drop the null values. The null values from payer\_code and medical\_specialty are significant either, so can drop the fields.

df = df.dropna()

Deleting irrelevant columns

The patient's form of payment does not have impact on the return to the hospital, so we dropped the Payer\_Code. However, we added the Medicare and Medicaid column to indicate whether or not the Payer\_Code for the hospital bill used a subsidize government medical assistance for low-income patients. This is to help us understand if there are any socioeconomic gaps in the diabetic patient demographic.

df.loc[:, "medicare"] = (df.payer\_code == "MC")

df.loc[:, "medicaid"] = (df.payer\_code == "MD")

There are 20+ columns of whether or not a patient took a certain diabetic medication (rosiglitazon, citoglipton, metformin etc.) that have no correlation on a patient's return to the hospital**.** As result, we’ll delete these medications.

df.drop(['encounter\_id', 'patient\_nbr', 'weight', 'payer\_code', 'medical\_specialty', 'admission\_type\_id',

         'repaglinide','nateglinide','chlorpropamide','glimepiride','acetohexamide','glipizide','glyburide','tolbutamide',

        'pioglitazone','rosiglitazone','acarbose','miglitol','troglitazone','tolazamide','examide','citoglipton', 'metformin',

        'glyburide-metformin','glipizide-metformin','glimepiride-pioglitazone','metformin-rosiglitazone','metformin-pioglitazone', 'diag\_2', 'diag\_3', 'change'], axis=1, inplace=True)

Lastly, unique patient numbers or hospital record identifications do not have value to model. Also, a patient’s first, second or third diagnosis while that were hospitalized may be useful , but are too random from patient to patient.

Formatting data

Part of working with data is formatting the columns or values into names that are more meaningful and easier to understand. Here we’ll change the readmitted outcome from NO or <30.

df['readmitted'] = df['readmitted'].replace({"NO":"not readmitted", "<30":"readmitted"})

Next, the age group are represented in brackets. For example, [0 - 10] denotes the age between 0 and 10 years old. In addition, since there’s a small sample size of patients in this age group will consolidate the age groups.

df.loc[:, "age"] = df["age"].replace( ["[0-10)", "[10-20)", "[20-30)"], "30 years or younger")

df.loc[:, "age"] = df["age"].replace(["[30-40)", "[40-50)", "[50-60)"], "30-60 years")

df.loc[:, "age"] = df["age"].replace(["[60-70)", "[70-80)", "[80-90)", "[90-100)"], "Over 60 years")

Task #1: Training data

Now that the data has been cleansed, it is ready to be used for training our model. To do this we will split out data into training and testing data sets with scikit-learns train\_test\_spilt function. This is optional, but we will store our data for later use. The file format we’ll use for our dataset is parquet.

Next, to normalize our data, we’ll create a helper function that identified all the indices for the non-numeric columns. Scikit-Learn’s OneHotEncoder will be used to encode all the string or object columns. Next, we’ll use the StandardScalar to encode the number column fields. The ColumnTransformer will be use to perform the data transformation. Where create a pipeline to put all of these prepressing tasks for the model to execute before training the model.

For our diabetes readmission classification, we use the LogisticRegression model on our dataset.

After training the model, we can review the accuracy score to evaluate how well our trained model performed. The score of 0.86 looks good.

Task 2: Train the model in the cloud

In the previous lab, we used the diabetes hospital readmission dataset to understand and wrangled data into a clean dataset to train a classification model to predication whether a diabetics patient would be readmitted back to a hospital within 30 days of being discharge. All of this done on the local machine. In this session, you will learn how to train a model in the cloud using the end-to-end streamline process to make your model reproduceable, easier to manage, and scalable.

Create a cloud client

Before you can use the Azure Machine Learning studio, you need to create a cloud client to authenticate and connect to the workspace. The authorization needs the subscription id, resource group, and name of the Azure ML workspace.

registry\_name = "azureml"

credential = DefaultAzureCredential()

ml\_client =  MLClient.from\_config(credential=credential)

ml\_client\_registry = MLClient(

    credential=credential,

    subscription\_id=ml\_client.subscription\_id,

    resource\_group\_name=ml\_client.resource\_group\_name,

    registry\_name=registry\_name

    )

Register the data

Data preparation is a long and tedious process the takes up a majority of the machine learning process. After you have cleanse into a good state, Azure Machine Learning provides datastores for you to register and store your datasets to. Your dataset is stored in one location, but it can be referenced as a point without having to move the data. The benefit of registering your datasets is, individuals on your team can reference them; and you can track all the experiments that used.

We are going to use the training and testing data that you used in Lab#1 and register it to the cloud. To register the files to Azure, we need to use the connect client. In the case, we need to specify the locally directory path. The AssetTypes.URI\_FILE specifies what time of data file we are using.

import os

from azure.ai.ml.entities import Data

from azure.ai.ml.constants import AssetTypes

training\_data = Data(

    name=training\_dataset\_filename,

    path='data/train\_dataset.parquet',

    type=AssetTypes.URI\_FILE,

    description="Patient train data",

)

tr\_data = ml\_client.data.create\_or\_update(training\_data)

testing\_data = Data(

    name=testing\_dataset\_filename,

    path='data/test\_dataset.parquet',

    type=AssetTypes.URI\_FILE,

    description="Patient test data",

)

te\_data = ml\_client.data.create\_or\_update(testing\_data)

Create a Compute

To run a job or machine learning computation, we need to create a compute instance. You can create none in notebook or use existing one. To create a compute, you need a name and the size of the instance. You can also specify how you and the instance to scare during runtime.

from azure.ai.ml.entities import AmlCompute

compute\_name = "trainingcompute"

all\_compute\_names = [x.name for x in ml\_client.compute.list()]

if compute\_name in all\_compute\_names:

    print(f"Found existing compute: {compute\_name}")

else:

    my\_compute = AmlCompute(

        name=compute\_name,

        size="Standard\_DS2\_v2",

        min\_instances=0,

        max\_instances=4,

        idle\_time\_before\_scale\_down=3600

    )

    ml\_client.compute.begin\_create\_or\_update(my\_compute)

    print("Initiated compute creation")

Create an environment

An environment specifies all the dependencies that a task or job that you need to run need. This can include python version, docker file, python libraries, etc. You have the option of creating a custom environment from scratch that suits your needs or use one of the Azure Machine Learning [curated environments](https://aka.ms/MBAzureMLEnvironments) available. In our case, we are using Responsible AI’s out of the box curated environment: AzureML-responsibleai-0.20-ubuntu20.04-py38-cpu since we’ll need needing it in the next lab. (**NOTE**:  Select the Environments tab in [Azure Machine Learn Studio](https://aka.ms/MBAzureMLStudio) to see the available environments)

Create training the model in the cloud

We’ll be using [Azure Machine Learning components](https://aka.ms/MBAzureMLComponent) to divide the experiment into different tasks. Components are reusable independent units of tasks that have inputs and output in machine learning (e.g., data cleaning, training a model, registering a model, deploying a model etc.). For our experiment, we will create a component for training a model. The component will consist of a python training code with inputs and outputs.

The first thing to define in our python code is the training script containing a function that declares an argument parser object that adds the names and types for the input and output parameters. For our Diabetes Hospital Readmission use case, we will be passing the path to our training dataset and the target column name as the classifier. Then the trained model will be the output for the script.

def parse\_args():

    # setup arg parser

    parser = argparse.ArgumentParser()

    # add arguments

    parser.add\_argument("--training\_data", type=str, help="Path to training data")

    parser.add\_argument("--target\_column\_name", type=str, help="Name of target column")

    parser.add\_argument("--model\_output", type=str, help="Path of output model")

    # parse args

    args = parser.parse\_args()

    # return args

    return args

In our python code, we’ll define a main class that the takes the input arguments to train the month. As you will see the code to change the model remain the same. The only change is create an experiment in Azure Machine Learning studio and using MLFlow to start the artifacts and metrics from your training model for tracking.

    current\_experiment = Run.get\_context().experiment

    tracking\_uri = current\_experiment.workspace.get\_mlflow\_tracking\_uri()

    print("tracking\_uri: {0}".format(tracking\_uri))

    mlflow.set\_tracking\_uri(tracking\_uri)

    mlflow.set\_experiment(current\_experiment.name)

After your model has been training. We’ll store the Scikit-Learn model output in a local directory then with MLFlow’s save\_model function for scikit-learn models to save the model output.

    model\_dir =  "./model\_output"

    with tempfile.TemporaryDirectory() as td:

        print("Saving model with MLFlow to temporary directory")

        tmp\_output\_dir = os.path.join(td, model\_dir)

        mlflow.sklearn.save\_model(sk\_model=model, path=tmp\_output\_dir)

        print("Copying MLFlow model to output path")

        for file\_name in os.listdir(tmp\_output\_dir):

            print("  Copying: ", file\_name)

            # As of Python 3.8, copytree will acquire dirs\_exist\_ok as

            # an option, removing the need for listdir

            shutil.copy2(src=os.path.join(tmp\_output\_dir, file\_name), dst=os.path.join(args.model\_output, file\_name))

Once you have defined you python code with the arguments and the main function to execute, we can create a compute to load the python to Azure Machine Learning components.

Create a job in the cloud

To define our training component, we’ll need to create a yaml file where we specified the component name, input, and output parameters; the location of the python code that trains the model; the command-line to execute the python code; and the environment to run the code. You can use Azure Machine Learning the Responsible AI’s out of the box curated environment: AzureML-responsibleai-0.20-ubuntu20.04-py38-cpu. Then use the client session to register the component definition in workspace components.

from azure.ai.ml import load\_component

yaml\_contents = f"""

$schema: http://azureml/sdk-2-0/CommandComponent.json

name: rai\_hospital\_training\_component

display\_name: hospital  classification training component for RAI example

version: {rai\_hospital\_classifier\_version\_string}

type: command

inputs:

  training\_data:

    type: path

  target\_column\_name:

    type: string

outputs:

  model\_output:

    type: path

code: ./component/

environment: azureml://registries/azureml/environments/AzureML-responsibleai-0.20-ubuntu20.04-py38-cpu/versions/4

""" + r"""

command: >-

  python hospital\_training.py

  --training\_data ${{{{inputs.training\_data}}}}

  --target\_column\_name ${{{{inputs.target\_column\_name}}}}

  --model\_output ${{{{outputs.model\_output}}}}

"""

yaml\_filename = "RAIhospitalClassificationTrainingComponent.yaml"

with open(yaml\_filename, 'w') as f:

    f.write(yaml\_contents.format(yaml\_contents))

train\_component\_definition = load\_component(

    source=yaml\_filename

)

ml\_client.components.create\_or\_update(train\_component\_definition)

Create a pipeline in the cloud

An [Azure Machine Learning pipeline](https://aka.ms/MBAzureMLPipeline) packages all the components and runs them sequentially during runtime with a specified compute instance, docker images or conda environments in the job process. After the training component defined above has been created, we need to define the pipeline that is going to package all the dependencies needed to run the training experiment. To do this, you will need the following:

* The experiment name and description
* Input object for the training dataset path
* Input object for the testing dataset path
* Get component name that trains the model
* Get component name that registers the model
* The compute instance for running the training job

All of this information is packaged in a pipeline job to run the experiment for training and registering the model.

from azure.ai.ml import dsl, Input

hospital\_train\_parquet = Input(

    type="uri\_file", path="data/train\_dataset.parquet", mode="download"

)

hospital\_test\_parquet = Input(

    type="uri\_file", path="data/test\_dataset.parquet", mode="download"

)

@dsl.pipeline(

    compute=compute\_name,

    description="Register Model for RAI hospital ",

    experiment\_name=f"RAI\_hospital\_Model\_Training\_{model\_name\_suffix}",

)

def my\_training\_pipeline(target\_column\_name, training\_data):

    trained\_model = train\_component\_definition(

        target\_column\_name=target\_column\_name,

        training\_data=training\_data

    )

    trained\_model.set\_limits(timeout=120)

    \_ = register\_component(

        model\_input\_path=trained\_model.outputs.model\_output,

        model\_base\_name=model\_base\_name,

        model\_name\_suffix=model\_name\_suffix,

    )

    return {}

model\_registration\_pipeline\_job = my\_training\_pipeline(target\_column, hospital\_train\_parquet)

Run the training job

Once you have defined and registered the pipeline to the workspace, you can submit the pipeline to run in a job. In our python code, we are checking the status of the job.

from azure.ai.ml.entities import PipelineJob

import webbrowser

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

    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))

    # open the pipeline in web browser

    webbrowser.open(created\_job.services["Studio"].endpoint)

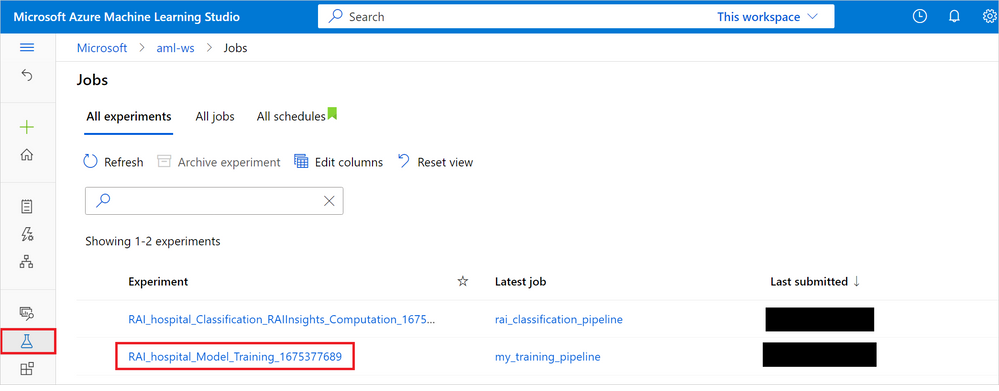
    #assert created\_job.status == 'Completed'

    return created\_job

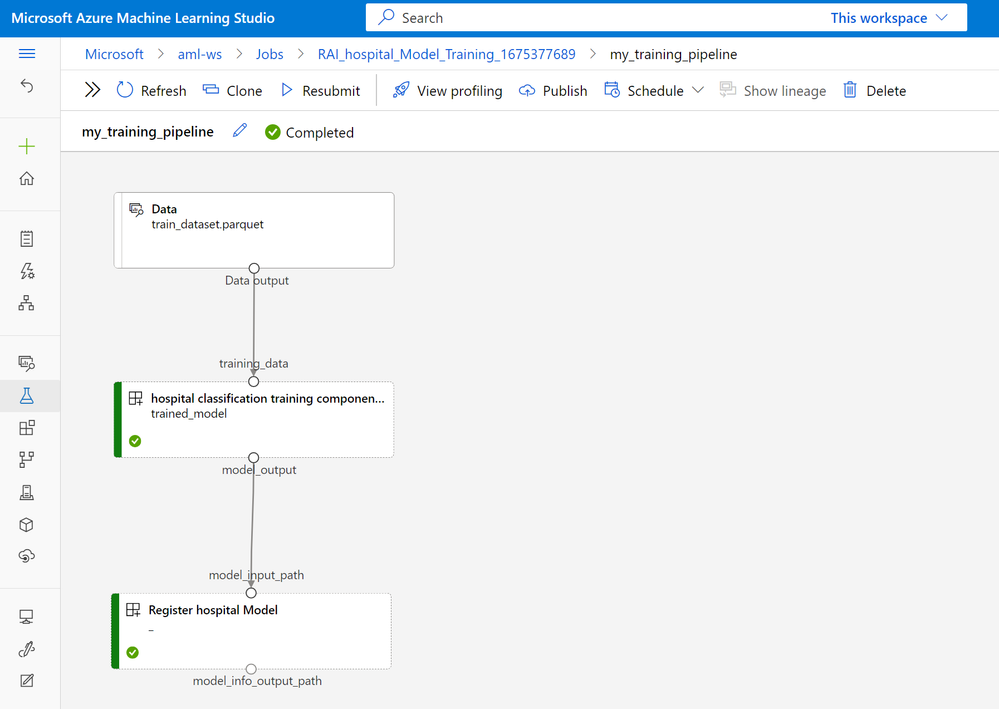
# This is the actual submission

training\_job = submit\_and\_wait(ml\_client, model\_registration\_pipeline\_job)

To monitor process of the job and status of all the components in the pipeline job is by clicking on the Jobs icon from the Azure Machine Learning Studio.



From the Jobs list, you can click on the job to view the jobs progress and can drill-down to pinpoint where an error occurred. You will see the Diabetes Hospital Readmission dataset feeding as an input into our training component and the output model feeding into the register model marked as complete. After the pipeline has successfully finished running, you will have a trained model that is registered to the Azure Machine Learning Studio.



Task 1: Train the model on your machine

1. Understand your data.
2. Understand our goal/scenario.
3. Clean your data.
4. Train the model locally.

Task 2: Train the model in the cloud

1. Register the data in the cloud.
2. Create a compute.
3. Create an environment.
4. Create a job and train the model in the cloud.

Task 3: Add Responsible AI

1. Create RAI component containing the output of the trained model, the validation and train data.
2. Create an RAI pipeline.
3. Run the pipeline.

Task 4: Analyzing the RAI dashboard

1. Open the dashboard and analyze various aspects of RAI.
2. Identify distribution errors in the model using error analyzes.
3. Discover model disparities across different data cohorts.
4. Explore data balance issues.
5. Find global and local explanations of your model.

Task 5: Deploy your model with an endpoint

1. Create an Azure ML deployment and endpoint.
2. Test the endpoint.