

Tune hyperparameters with a sweep job

There are many machine learning algorithms that require hyperparameters (parameter values that influence training, but can't be determined from the training data itself). For example, when training a logistic regression model, you can use a regularization rate hyperparameter to counteract bias in the model; or when training a convolutional neural network, you can use hyperparameters like learning rate and batch size to control how weights are adjusted and how many data items are processed in a mini-batch respectively. The choice of hyperparameter values can significantly affect the performance of a trained model, or the time taken to train it; and often you need to try multiple combinations to find the optimal solution.

Before you start

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 training script

Hyperparameter tuning is ideal when you want to train a machine learning models but vary the input parameters. You'll need to create a training script that expects an input parameter representing one of the algorithm's hyperparameters.

Run the following cells to create the `src` folder and the training script.

Note that the training script expects two input parameters:

- `--training_data` which expects a string. You'll specify the path to a registered data asset as the input

training data.

- `--reg_rate` which expects a number, but has a default value of `0.01`. You'll use this input parameter for hyperparameter tuning.

In []:

```
import os

# create a folder for the script files
script_folder = 'src'
os.makedirs(script_folder, exist_ok=True)
print(script_folder, 'folder created')
```

In []:

```
%%writefile $script_folder/train.py
# import libraries
import mlflow
import argparse
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
import matplotlib.pyplot as plt

def main(args):
    # read data
    df = get_data(args.training_data)

    # split data
    X_train, X_test, y_train, y_test = split_data(df)

    # train model
    model = train_model(args.reg_rate, X_train, X_test, y_train, y_test)

    # evaluate model
    eval_model(model, X_test, y_test)

# function that reads the data
def get_data(path):
    print("Reading data...")
    df = pd.read_csv(path)

    return df

# function that splits the data
def split_data(df):
    print("Splitting data...")
    X, y = df[['Pregnancies', 'PlasmaGlucose', 'DiastolicBloodPressure', 'TricepsThickness',
    'SerumInsulin', 'BMI', 'DiabetesPedigree', 'Age']].values, df['Diabetic'].values

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)

    return X_train, X_test, y_train, y_test

# function that trains the model
def train_model(reg_rate, X_train, X_test, y_train, y_test):
    mlflow.log_param("Regularization rate", reg_rate)
    print("Training model...")
    model = LogisticRegression(C=1/reg_rate, solver="liblinear").fit(X_train, y_train)

    return model

# function that evaluates the model
def eval_model(model, X_test, y_test):
    # calculate accuracy
```

```

y_hat = model.predict(X_test)
acc = np.average(y_hat == y_test)
print('Accuracy:', acc)
mlflow.log_metric("Accuracy", acc)

# calculate AUC
y_scores = model.predict_proba(X_test)
auc = roc_auc_score(y_test, y_scores[:,1])
print('AUC: ' + str(auc))
mlflow.log_metric("AUC", auc)

# plot ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores[:,1])
fig = plt.figure(figsize=(6, 4))
# Plot the diagonal 50% line
plt.plot([0, 1], [0, 1], 'k--')
# Plot the FPR and TPR achieved by our model
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.savefig("ROC-Curve.png")
mlflow.log_artifact("ROC-Curve.png")

def parse_args():
    # setup arg parser
    parser = argparse.ArgumentParser()

    # add arguments
    parser.add_argument("--training_data", dest='training_data',
                        type=str)
    parser.add_argument("--reg_rate", dest='reg_rate',
                        type=float, default=0.01)

    # parse args
    args = parser.parse_args()

    # return args
    return args

# run script
if __name__ == "__main__":
    # add space in logs
    print("\n\n")
    print("*" * 60)

    # parse args
    args = parse_args()

    # run main function
    main(args)

    # add space in logs
    print("*" * 60)
    print("\n\n")

```

Configure and run a command job

Run the cell below to train a classification model to predict diabetes. The model is trained by running the `train.py` script that can be found in the `src` folder. It uses the registered `diabetes-data` data asset as the training data.

- `code` : specifies the folder that includes the script to run.
- `command` : specifies what to run exactly.
- `environment` : specifies the necessary packages to be installed on the compute before running the command.
- `compute` : specifies the compute to use to run the command.
- `display_name` : the name of the individual job.
- `experiment_name` : the name of the experiment the job belongs to.

Note that the command job only runs the training script once, with a regularization rate of 0.1. Before you run a sweep job to tune hyperparameters, it's a best practice to test whether your script works as expected with a command job.

In []:

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

# configure job

job = command(
    code="./src",
    command="python train.py --training_data ${inputs.diabetes_data} --reg_rate ${inputs.reg_rate}",
    inputs={
        "diabetes_data": Input(
            type=AssetTypes.URI_FILE,
            path="azureml:diabetes-data:1"
        ),
        "reg_rate": 0.01,
    },
    environment="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu@latest",
    compute="aml-cluster",
    display_name="diabetes-train-mlflow",
    experiment_name="diabetes-training",
    tags={"model_type": "LogisticRegression"}
)

# submit job
returned_job = ml_client.create_or_update(job)
aml_url = returned_job.studio_url
print("Monitor your job at", aml_url)
```

Define the search space

When your command job has completed successfully, you can configure and run a sweep job.

First, you'll need to specify the search space for your hyperparameter. To train three models, each with a different regularization rate (0.01, 0.1, or 1), you can define the search space with a Choice hyperparameter.

In []:

```
from azure.ai.ml.sweep import Choice

command_job_for_sweep = job(
    reg_rate=Choice(values=[0.01, 0.1, 1]),
)
```

Configure and submit the sweep job

You'll use the sweep function to do hyperparameter tuning on your training script. To configure a sweep job, you'll need to configure the following:

- compute**: Name of the compute target to execute the job on.
- sampling_algorithm**: The hyperparameter sampling algorithm to use over the search space. Allowed values are random, grid and bayesian.
- primary_metric**: The name of the primary metric reported by each trial job. The metric must be logged in the user's training script using `mlflow.log_metric()` with the same corresponding metric name.
- goal**: The optimization goal of the primary_metric. The allowed values are maximize and minimize.
- limits**: Limits for the sweep job. For example, the maximum amount of trials or models you want to train.

Note that the command job is used as the base for the sweep job. The configuration for the command job will be reused by the sweep job.

In []:

```
# apply the sweep parameter to obtain the sweep job
sweep_job = command_job_for_sweep.sweep(
    compute="aml-cluster",
    sampling_algorithm="grid", (random, bayesian, grid)
    primary_metric="training_accuracy_score",
    goal="Maximize",
)

# set the name of the sweep job experiment
sweep_job.experiment_name="sweep-diabetes"

# define the limits for this sweep
sweep_job.set_limits(max_total_trials=4, max_concurrent_trials=2, timeout=7200)
```

Run the following cell to submit the sweep job.

In []:

```
returned_sweep_job = ml_client.create_or_update(sweep_job)
aml_url = returned_sweep_job.studio_url
print("Monitor your job at", aml_url)
```

When the job is completed, navigate to the job overview. The Trials tab will show all models that have been trained and how the Accuracy score differs for each regularization rate value you tried.