SageMaker Pipelines California Housing - Taking different steps based on model performance

This notebook illustrates how to take different actions based on model performance in a SageMaker Pipeline.

The steps in this pipeline include: * Preprocessing the California Housing dataset. * Train a TensorFlow2 Artificial Neural Network (ANN) Model. * Evaluate the model performance - mean square error (MSE). * If MSE is higher than threshold, use a Lambda step to send an E-Mail to the Data Science team. * If MSE is lower than threshold, register the model into the Model Registry, and use a Lambda step to deploy the model to SageMaker Endpoint.

Prerequisites

Add `AmazonSageMakerPipelinesIntegrations` policy

The notebook execution role should have policies which enable the notebook to create a Lambda function. The Amazon managed policy `AmazonSageMakerPipelinesIntegrations` can be added to the notebook execution role.

The policy description is:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:DeleteFunction",
                "lambda:InvokeFunction",
                "lambda:UpdateFunctionCode"
            ],
            "Resource": [
                "arn:aws:lambda:*:*:function:*sagemaker*",
                "arn:aws:lambda:*:*:function:*sageMaker*",
                "arn:aws:lambda:*:*:function:*SageMaker*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "sqs:CreateQueue",
                "sqs:SendMessage"
            ],
            "Resource": [
                "arn:aws:sqs:*:*:*sagemaker*",
                "arn:aws:sqs:*:*:*sageMaker*",
                "arn:aws:sqs:*:*:*SageMaker*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::*:role/*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "lambda.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

Add inline policy to enable creation of IAM role required for the Lambda Function

The notebook execution role should have an inline policy which enable the notebook to create the IAM role required for the Lambda function. An inline policy can be added to the notebook execution role.

The policy description is:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:CreateRole",
                "iam:AttachRolePolicy"
            ],
            "Resource": "*"
        }
    ]
}
```

[ ]:
```
import sys

!{sys.executable} -m pip install "sagemaker>=2.51.0"
```

[ ]:
```
import os
import time
import boto3
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import sagemaker
from sagemaker import get_execution_role
```

[ ]:
```
sess = boto3.Session()
sm = sess.client("sagemaker")
role = get_execution_role()
sagemaker_session = sagemaker.Session(boto_session=sess)
bucket = sagemaker_session.default_bucket()
region = boto3.Session().region_name
model_package_group_name = "TF2-California-Housing"  # Model name in model registry
prefix = "tf2-california-housing-pipelines"
pipeline_name = "TF2CaliforniaHousingPipeline"  # SageMaker Pipeline name
current_time = time.strftime("%m-%d-%H-%M-%S", time.localtime())
```

Download California Housing dataset and upload to Amazon S3

We use the California housing dataset.

More info on the dataset:

This dataset was obtained from the `StatLib` repository. http://lib.stat.cmu.edu/datasets/

The target variable is the median house value for California districts.

This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people).

[ ]:
```
data_dir = os.path.join(os.getcwd(), "data")
os.makedirs(data_dir, exist_ok=True)

raw_dir = os.path.join(os.getcwd(), "data/raw")
os.makedirs(raw_dir, exist_ok=True)
```

[ ]:
```
!aws s3 cp s3://sagemaker-sample-files/datasets/tabular/california_housing
/cal_housing.tgz .
```

[ ]:
```
!tar -zxf cal_housing.tgz
```

[ ]:
```
columns = [
    "longitude",
    "latitude",
    "housingMedianAge",
    "totalRooms",
    "totalBedrooms",
    "population",
    "households",
    "medianIncome",
    "medianHouseValue",
]
cal_housing_df = pd.read_csv("CaliforniaHousing/cal_housing.data", names=columns,
header=None)
```

[ ]:
```
cal_housing_df.head()
```

[ ]:
```
```

```
X = cal_housing_df[
    [
```

[ ]:
```
from sagemaker.workflow.parameters import ParameterInteger, ParameterString,
ParameterFloat

# raw input data
input_data = ParameterString(name="InputData", default_value=raw_s3)

# processing step parameters
processing_instance_type = ParameterString(
    name="ProcessingInstanceType", default_value="ml.m5.large"
)

# training step parameters
training_instance_type = ParameterString(name="TrainingInstanceType",
default_value="ml.m5.large")
training_epochs = ParameterString(name="TrainingEpochs", default_value="100")

# model performance step parameters
accuracy_mse_threshold = ParameterFloat(name="AccuracyMseThreshold",
default_value=0.75)

# Inference step parameters
endpoint_instance_type = ParameterString(name="EndpointInstanceType",
default_value="ml.m5.large")
```

Processing Step

The first step in the pipeline will preprocess the data to prepare it for training. We create a `SKLearnProcessor` object similar to the one above, but now parameterized, so we can separately track and change the job configuration as needed, for example to increase the instance type size and count to accommodate a growing dataset.

[ ]:

```
%%writefile preprocess.py

import glob
import numpy as np
import os
from sklearn.preprocessing import StandardScaler

if __name__ == "__main__":

    input_files = glob.glob("{}/*.npy".format("/opt/ml/processing/input"))
    print("\nINPUT FILE LIST: \n{}\n".format(input_files))
    scaler = StandardScaler()
    x_train = np.load(os.path.join("/opt/ml/processing/input", "x_train.npy"))
    scaler.fit(x_train)
    for file in input_files:
```

[ ]:
```
from sagemaker.sklearn.processing import SKLearnProcessor
from sagemaker.processing import ProcessingInput, ProcessingOutput
from sagemaker.workflow.steps import ProcessingStep

framework_version = "0.23-1"

# Create SKlearn processor object,
# The object contains information about what instance type to use, the IAM role to use
etc.
# A managed processor comes with a preconfigured container, so only specifying version
is required.
sklearn_processor = SKLearnProcessor(
    framework_version=framework_version,
    role=role,
    instance_type=processing_instance_type,
    instance_count=1,
    base_job_name="tf2-california-housing-processing-job",
)

# Use the sklearn_processor in a Sagemaker pipelines ProcessingStep
step_preprocess_data = ProcessingStep(
    name="Preprocess-California-Housing-Data",
    processor=sklearn_processor,
    inputs=[
        ProcessingInput(source=input_data, destination="/opt/ml/processing/input"),
    ],
    outputs=[
        ProcessingOutput(output_name="train", source="/opt/ml/processing/train"),
        ProcessingOutput(output_name="test", source="/opt/ml/processing/test"),
    ],
    code="preprocess.py",
)
```

Train model step

In the second step, the train and validation output from the precious processing step are used to train a model.

[ ]:

```
from sagemaker.tensorflow import TensorFlow
from sagemaker.inputs import TrainingInput
from sagemaker.workflow.steps import TrainingStep
from sagemaker.workflow.step_collections import RegisterModel
import time

# Where to store the trained model
model_path = f"s3://{bucket}/{prefix}/model/"

hyperparameters = {"epochs": training_epochs}
tensorflow_version = "2.4.1"
python_version = "py37"

tf2_estimator = TensorFlow(
    source_dir="code",
    entry_point="train.py",
    instance_type=training_instance_type,
    instance_count=1,
    framework_version=tensorflow_version,
    role=role,
    base_job_name="tf2-california-housing-train",
    output_path=model_path,
    hyperparameters=hyperparameters,
    py_version=python_version,
)

# Use the tf2_estimator in a Sagemaker pipelines ProcessingStep.
# NOTE how the input to the training job directly references the output of the
previous step.
step_train_model = TrainingStep(
    name="Train-California-Housing-Model",
    estimator=tf2_estimator,
    inputs={
        "train": TrainingInput(
            s3_data=step_preprocess_data.properties.ProcessingOutputConfig.Outputs[
                "train"
            ].S3Output.S3Uri,
            content_type="text/csv",
        ),
        "test": TrainingInput(
            s3_data=step_preprocess_data.properties.ProcessingOutputConfig.Outputs[
                "test"
            ].S3Output.S3Uri,
            content_type="text/csv",
        ),
    },
```

Evaluate model step

When a model is trained, it's common to evaluate the model on unseen data before registering it with the model registry. This ensures the model registry isn't cluttered with poorly performing model versions. To evaluate the model, create a ScriptProcessor object and use it in a ProcessingStep.

Note that a separate preprocessed test dataset is used to evaluate the model, and not the output of the processing step. This is only for demo purposes, to ensure the second run of the pipeline creates a model with better performance. In a real-world scenario, the test output of the processing step would be used.

[ ]:
```
%%writefile evaluate.py

import os
import json
import subprocess
import sys
import numpy as np
import pathlib
import tarfile


def install(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])


if __name__ == "__main__":

    install("tensorflow==2.4.1")
    model_path = f"/opt/ml/processing/model/model.tar.gz"
    with tarfile.open(model_path, "r:gz") as tar:
        tar.extractall("./model")
    import tensorflow as tf

    model = tf.keras.models.load_model("./model/1")
    test_path = "/opt/ml/processing/test/"
    x_test = np.load(os.path.join(test_path, "x_test.npy"))
    y_test = np.load(os.path.join(test_path, "y_test.npy"))
    scores = model.evaluate(x_test, y_test, verbose=2)
    print("\nTest MSE :", scores)

    # Available metrics to add to model: https://docs.aws.amazon.com/sagemaker/latest
/dg/model-monitor-model-quality-metrics.html
    report_dict = {
        "regression_metrics": {
            "mse": {"value": scores, "standard_deviation": "NaN"},
        },
    }

    output_dir = "/opt/ml/processing/evaluation"
    pathlib.Path(output_dir).mkdir(parents=True, exist_ok=True)

    evaluation_path = f"{output_dir}/evaluation.json"
    with open(evaluation_path, "w") as f:
        f.write(json.dumps(report_dict))
```

[ ]:
```
```

```python
from sagemaker.workflow.properties import PropertyFile

# Create SKLearnProcessor object.
# The object contains information about what container to use, what instance type etc.
evaluate_model_processor = SKLearnProcessor(
    framework_version=framework_version,
    instance_type=processing_instance_type,
    instance_count=1,
    base_job_name="tf2-california-housing-evaluate",
    role=role,
)

# Create a PropertyFile
# A PropertyFile is used to be able to reference outputs from a processing step, for
instance to use in a condition step.
# For more information, visit https://docs.aws.amazon.com/sagemaker/latest/dg/build-
and-manage-propertyfile.html
evaluation_report = PropertyFile(
    name="EvaluationReport", output_name="evaluation", path="evaluation.json"
)

# Use the evaluate_model_processor in a Sagemaker pipelines ProcessingStep.
step_evaluate_model = ProcessingStep(
    name="Evaluate-California-Housing-Model",
    processor=evaluate_model_processor,
    inputs=[
        ProcessingInput(
            source=step_train_model.properties.ModelArtifacts.S3ModelArtifacts,
            destination="/opt/ml/processing/model",
        ),
        ProcessingInput(
            source=step_preprocess_data.properties.ProcessingOutputConfig.Outputs[
                "test"
            ].S3Output.S3Uri,
            destination="/opt/ml/processing/test",
        ),
    ],
    outputs=[
        ProcessingOutput(output_name="evaluation", source="/opt/ml/processing
/evaluation"),
    ],
    code="evaluate.py",
    property_files=[evaluation_report],
```

Send E-Mail Lambda Step

When defining the `LambdaStep`, the SageMaker Lambda helper class provides helper functions for creating the Lambda function. Users can either use the `lambda_func` argument to provide the function ARN to an already deployed Lambda function OR use the `Lambda` class to create a Lambda function by providing a script, function name and role for the Lambda function.

When passing inputs to the Lambda, the `inputs` argument can be used and within the Lambda function's handler, the `event` argument can be used to retrieve the inputs.

The dictionary response from the Lambda function is parsed through the `LambdaOutput` objects provided to the `outputs` argument. The `output_name` in `LambdaOutput` corresponds to the dictionary key in the Lambda's return dictionary.

Define the Lambda function

Users can choose the leverage the Lambda helper class to create a Lambda function and provide that function object to the `LambdaStep`. Alternatively, users can use a pre-deployed Lambda function and provide the function ARN to the `Lambda` helper class in the lambda step.

Here, If the MSE is lower than threshold, an E-Mail will be sent to Data Science team.

Note that the E-Mail sending part is left for you to implement by the framework you choose.

[ ]:
```
%%writefile send_email_lambda.py

"""
This Lambda function sends an E-Mail to the Data Science team with the MSE from model
evaluation step.
The evaluation.json location in S3 is provided via the `event` argument
"""

import json
import boto3


s3_client = client = boto3.client("s3")


def lambda_handler(event, context):

    print(f"Received Event: {event}")

    evaluation_s3_uri = event["evaluation_s3_uri"]
    path_parts = evaluation_s3_uri.replace("s3://", "").split("/")
    bucket = path_parts.pop(0)
    key = "/".join(path_parts)

    content = s3_client.get_object(Bucket=bucket, Key=key)
    text = content["Body"].read().decode()
    evaluation_json = json.loads(text)
    mse = evaluation_json["regression_metrics"]["mse"]["value"]

    subject_line = "Please check high MSE ({}) detected on model
evaluation".format(mse)
    print(f"Sending E-Mail to Data Science Team with subject line: {subject_line}")

    # TODO - ADD YOUR CODE TO SEND EMAIL...

    return {"statusCode": 200, "body": json.dumps("E-Mail Sent Successfully")}
```

IAM Role

The Lambda function needs an IAM role that will allow it to read the `evaluation.json` from S3. The role ARN must be provided in the `LambdaStep`.

A helper function in `iam_helper.py` is available to create the Lambda function role. Please note that the role uses the Amazon managed policy - `AmazonS3ReadOnlyAccess`. This should be replaced with an IAM policy with the least privileges as per AWS IAM best practices.

[ ]:
```
from iam_helper import create_s3_lambda_role

lambda_role = create_s3_lambda_role("send-email-to-ds-team-lambda-role")
```

Create the Lambda Function step

[ ]:
```
from sagemaker.workflow.lambda_step import LambdaStep
from sagemaker.lambda_helper import Lambda

evaluation_s3_uri = "{}/evaluation.json".format(
    step_evaluate_model.arguments["ProcessingOutputConfig"]["Outputs"][0]["S3Output"]
["S3Uri"]
)

send_email_lambda_function_name = "sagemaker-send-email-to-ds-team-lambda-" +
current_time

send_email_lambda_function = Lambda(
    function_name=send_email_lambda_function_name,
    execution_role_arn=lambda_role,
    script="send_email_lambda.py",
    handler="send_email_lambda.lambda_handler",
)

step_higher_mse_send_email_lambda = LambdaStep(
    name="Send-Email-To-DS-Team",
    lambda_func=send_email_lambda_function,
    inputs={"evaluation_s3_uri": evaluation_s3_uri},
)
```

Register model step

If the trained model meets the model performance requirements a new model version is registered with the model registry for further analysis. To attach model metrics to the model version, create a `` `ModelMetrics `` <https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor-model-quality-metrics.html>`__ object using the evaluation report created in the evaluation step. Then, create the RegisterModel step.

[ ]:

```
from sagemaker.model_metrics import MetricsSource, ModelMetrics
from sagemaker.workflow.step_collections import RegisterModel

# Create ModelMetrics object using the evaluation report from the evaluation step
# A ModelMetrics object contains metrics captured from a model.
model_metrics = ModelMetrics(
    model_statistics=MetricsSource(
        s3_uri=evaluation_s3_uri,
        content_type="application/json",
    )
)

# Create a RegisterModel step, which registers the model with Sagemaker Model
Registry.
step_register_model = RegisterModel(
    name="Register-California-Housing-Model",
    estimator=tf2_estimator,
    model_data=step_train_model.properties.ModelArtifacts.S3ModelArtifacts,
    content_types=["text/csv"],
    response_types=["text/csv"],
    inference_instances=["ml.m5.large", "ml.m5.xlarge"],
    transform_instances=["ml.m5.xlarge"],
    model_package_group_name=model_package_group_name,
```

Create the model

The model is created and the name of the model is provided to the Lambda function for deployment. The `CreateModelStep` dynamically assigns a name to the model.

```
[ ]: from sagemaker.workflow.step_collections import CreateModelStep
from sagemaker.tensorflow.model import TensorFlowModel

model = TensorFlowModel(
    role=role,
    model_data=step_train_model.properties.ModelArtifacts.S3ModelArtifacts,
    framework_version=tensorflow_version,
    sagemaker_session=sagemaker_session,
)

step_create_model = CreateModelStep(
    name="Create-California-Housing-Model",
    model=model,
    inputs=sagemaker.inputs.CreateModelInput(instance_type=endpoint_instance_type),
)
```

Deploy model to SageMaker Endpoint Lambda Step

When defining the `LambdaStep`, the SageMaker Lambda helper class provides helper functions for creating the Lambda function. Users can either use the `lambda_func` argument to provide the function ARN to an already deployed Lambda function OR use the `Lambda` class to create a Lambda function by providing a script, function name and role for the Lambda function.

When passing inputs to the Lambda, the `inputs` argument can be used and within the Lambda function's handler, the `event` argument can be used to retrieve the inputs.

The dictionary response from the Lambda function is parsed through the `LambdaOutput` objects provided to the `outputs` argument. The `output_name` in `LambdaOutput` corresponds to the dictionary key in the Lambda's return dictionary.

Here, the Lambda Function will deploy the model to SageMaker Endpoint.

[ ]:

```
%%writefile deploy_model_lambda.py


"""
This Lambda function deploys the model to SageMaker Endpoint.
If Endpoint exists, then Endpoint will be updated with new Endpoint Config.
"""

import json
import boto3
import time
```

IAM Role

The Lambda function needs an IAM role that will allow it to deploy a SageMaker Endpoint. The role ARN must be provided in the `LambdaStep`.

A helper function in `iam_helper.py` is available to create the Lambda function role. Please note that the role uses the Amazon managed policy - `AmazonSageMakerFullAccess`. This should be replaced with an IAM policy with the least privileges as per AWS IAM best practices.

```
    print(f"Received Event: {event}")
    current_time = time.strftime("%m-%d-%H-%M-%S", time.localtime())
    endpoint_instance_type = event["endpoint_instance_type"]
```

[ ]:
```
from iam_helper import create_sagemaker_lambda_role

lambda_role = create_sagemaker_lambda_role("deploy-model-lambda-role")
```

[ ]:
```
from sagemaker.workflow.lambda_step import LambdaStep
from sagemaker.lambda_helper import Lambda

endpoint_config_name = "tf2-california-housing-endpoint-config"
endpoint_name = "tf2-california-housing-endpoint-" + current_time

deploy_model_lambda_function_name = "sagemaker-deploy-model-lambda-" + current_time

deploy_model_lambda_function = Lambda(
    function_name=deploy_model_lambda_function_name,
    execution_role_arn=lambda_role,
    script="deploy_model_lambda.py",
    handler="deploy_model_lambda.lambda_handler",
)

step_lower_mse_deploy_model_lambda = LambdaStep(
    name="Deploy-California-Housing-Model-To-Endpoint",
    lambda_func=deploy_model_lambda_function,
    inputs={
        "model_name": step_create_model.properties.ModelName,
        "endpoint_config_name": endpoint_config_name,
        "endpoint_name": endpoint_name,
        "endpoint_instance_type": endpoint_instance_type,
    },
)
```

```
        update_endpoint_response = sm_client.update_endpoint(
            EndpointName=endpoint_name, EndpointConfigName=endpoint_config_name
        print(f"update_endpoint_response: {update_endpoint_response}")
    else:
        print("Creating Endpoint")
        create_endpoint_response = sm_client.create_endpoint(
            EndpointName=endpoint_name, EndpointConfigName=endpoint_config_name
```

Accuracy condition step

Adding conditions to the pipeline is done with a ConditionStep. In this case, we only want to register the new model version with the model registry if the new model meets an accuracy condition.

```
)
print(f"create_endpoint_response: {create_endpoint_response}")
```

[ ]:
```
from sagemaker.workflow.conditions import ConditionLessThanOrEqualTo
from sagemaker.workflow.condition_step import ConditionStep
from sagemaker.workflow.functions import JsonGet

# Create accuracy condition to ensure the model meets performance requirements.
# Models with a test accuracy lower than the condition will not be registered with the
model registry.
cond_lte = ConditionLessThanOrEqualTo(
    left=JsonGet(
        step_name=step_evaluate_model.name,
        property_file=evaluation_report,
        json_path="regression_metrics.mse.value",
    ),
    right=accuracy_mse_threshold,
)

# Create a Sagemaker Pipelines ConditionStep, using the condition above.
# Enter the steps to perform if the condition returns True / False.
step_cond = ConditionStep(
    name="MSE-Lower-Than-Threshold-Condition",
    conditions=[cond_lte],
    if_steps=[step_register_model, step_create_model,
step_lower_mse_deploy_model_lambda],
    else_steps=[step_higher_mse_send_email_lambda],
)
```

Pipeline Creation: Orchestrate all steps

Now that all pipeline steps are created, a pipeline is created.

[ ]:
```
from sagemaker.workflow.pipeline import Pipeline

# Create a Sagemaker Pipeline.
# Each parameter for the pipeline must be set as a parameter explicitly when the
pipeline is created.
# Also pass in each of the steps created above.
# Note that the order of execution is determined from each step's dependencies on
other steps,
# not on the order they are passed in below.
pipeline = Pipeline(
    name=pipeline_name,
    parameters=[
        processing_instance_type,
        training_instance_type,
        input_data,
        training_epochs,
        accuracy_mse_threshold,
        endpoint_instance_type,
    ],
    steps=[step_preprocess_data, step_train_model, step_evaluate_model, step_cond],
)
```

Execute the Pipeline

[ ]:
```
import json

definition = json.loads(pipeline.definition())
definition
```

[ ]:
```
pipeline.upsert(role_arn=role)
```

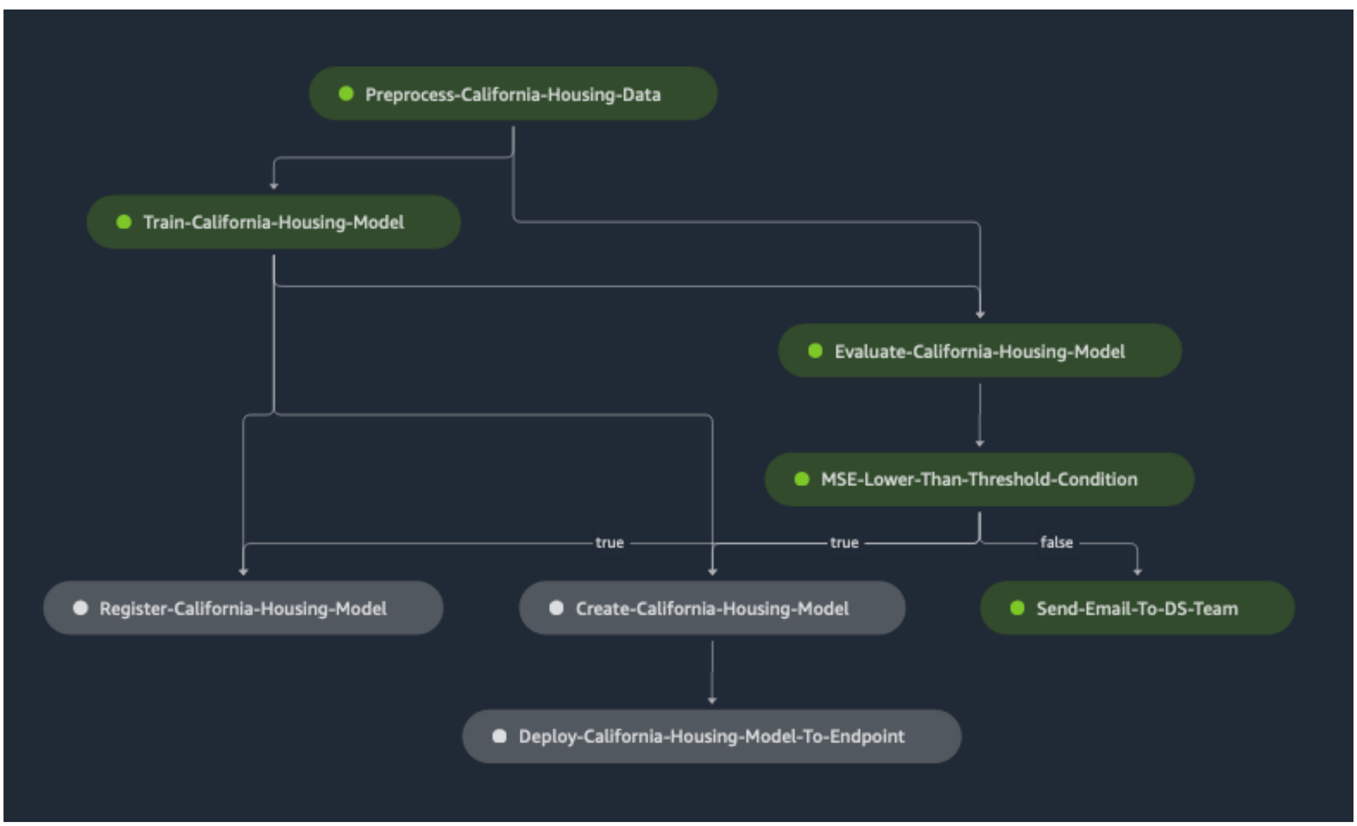
[ ]:
```
execution = pipeline.start()
```

[ ]:
```
execution.wait()
```

Visualize SageMaker Pipeline - MSE lower than the threshold

In SageMaker Studio, choose `SageMaker Components and registries` in the left pane and under `Pipelines`, click the pipeline that was created. Then all pipeline executions are shown, and the one just created should have a status of `Succeded`. Selecting that execution, the different pipeline steps can be tracked as they execute.

You can see that the `Register-California-Housing-Model` step was executed.



Start a pipeline with 2 epochs to trigger the `send-email-to-`

`ds-team-lambda` Lambda Function

Run the pipeline again, but this time, with only 2 epochs and a lower MSE Threshold of 0.2. This will result in a higher MSE value on model evaluation, and will cause the `send-email-to-ds-team-lambda` Lambda Function to be triggered.

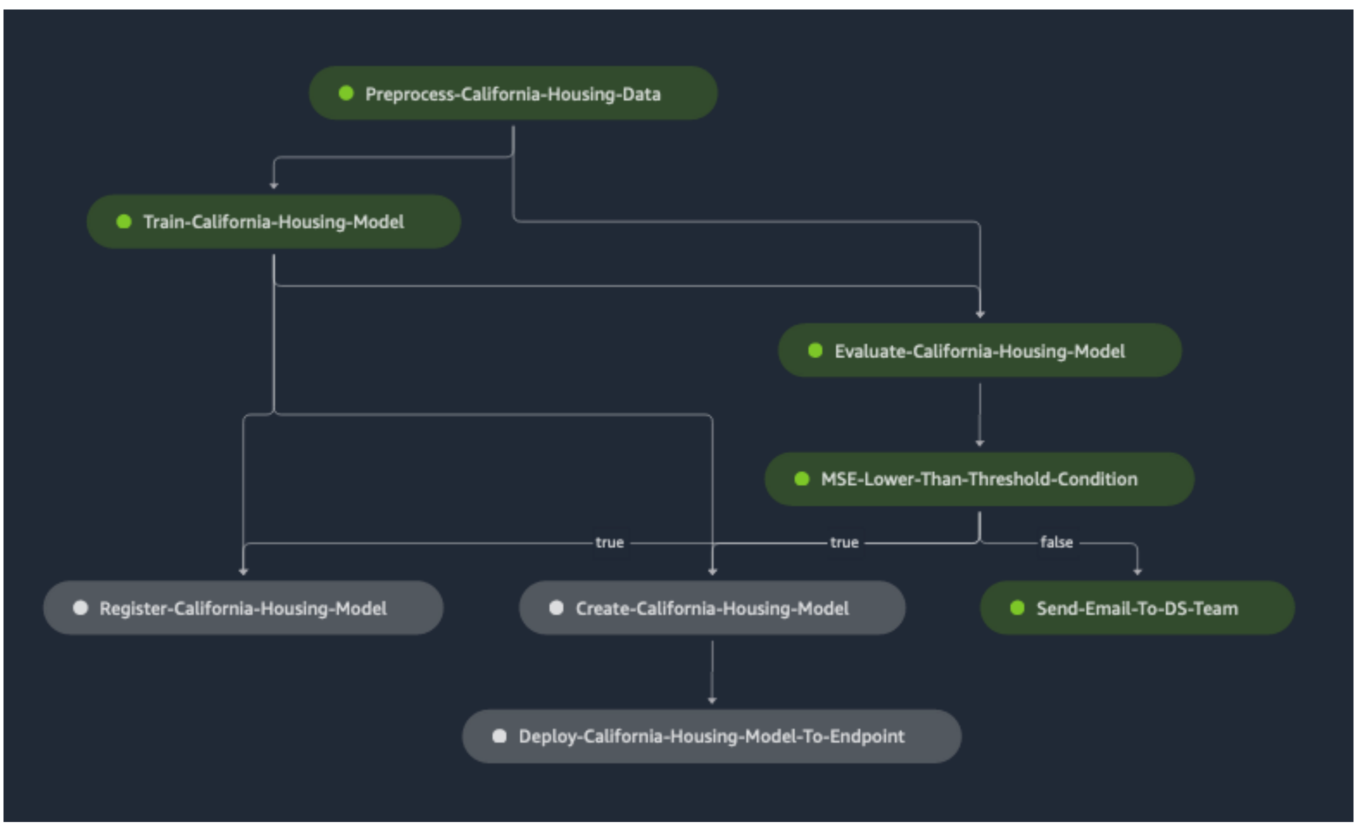
```
[ ]: # Execute pipeline with explicit parameters
     execution = pipeline.start(parameters=dict(TrainingEpochs=2,
     AccuracyMseThreshold=0.2))
```

```
[ ]: execution.wait()
```

Visualize SageMaker Pipeline - MSE higher than the threshold

In SageMaker Studio, choose `SageMaker Components and registries` in the left pane and under `Pipelines`, click the pipeline that was created. Then all pipeline executions are shown, and the one just created should have a status of `Succeded`. Selecting that execution, the different pipeline steps can be tracked as they execute.

You can see that the `Send-Email-To-DS-Team` step was executed.



Clean up (optional)

Stop / Close the Endpoint

You should delete the endpoint before you close the notebook if you don't need to keep the endpoint running for serving real-time predictions.

[ ]:
```
import boto3

client = boto3.client("sagemaker")
client.delete_endpoint(EndpointName=endpoint_name)
```

Delete the model registry and the pipeline to keep the studio environment tidy.

[ ]:
```
def delete_model_package_group(sm_client, package_group_name):
    try:
        model_versions =
sm_client.list_model_packages(ModelPackageGroupName=package_group_name)

    except Exception as e:
        print("{} \n".format(e))
        return

    for model_version in model_versions["ModelPackageSummaryList"]:
        try:

sm_client.delete_model_package(ModelPackageName=model_version["ModelPackageArn"])
        except Exception as e:
            print("{} \n".format(e))
        time.sleep(0.5)  # Ensure requests aren't throttled

    try:
        sm_client.delete_model_package_group(ModelPackageGroupName=package_group_name)
        print("{} model package group deleted".format(package_group_name))
    except Exception as e:
        print("{} \n".format(e))
    return


def delete_sagemaker_pipeline(sm_client, pipeline_name):
    try:
        sm_client.delete_pipeline(
            PipelineName=pipeline_name,
        )
        print("{} pipeline deleted".format(pipeline_name))
    except Exception as e:
        print("{} \n".format(e))
        return
```

[ ]:
```
delete_model_package_group(client, model_package_group_name)
delete_sagemaker_pipeline(client, pipeline_name)
```

Delete the Lambda functions.

[ ]:
```
send_email_lambda_function.delete()
deploy_model_lambda_function.delete()
```

[ ]:
```

```