# DIVISION OF COMPUTER SCIENCE AND ENGINEERING

**SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY**

## LABORATORY RECORD ODD SEMESTER 2023-2024

**Name**

**:**

**Reg.No**

**:**

**Course Code :**

**Course Name :**

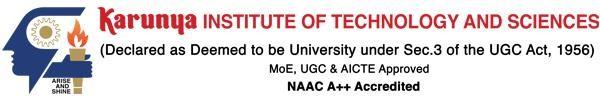
**JOSEPH FRANKLIN S**

**URK20AI1005**

**21CS2015**

**MLOps Lab**

**NOVEMBER 2023**



# DIVISION OF COMPUTER SCIENCE AND ENGINEERING

**SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY**

## LABORATORY RECORD

**Academic Year 2023-2024 Course Code**

# 21CS2015

## Course Name

# MLOps Lab

**Register No. URK20AI1005**

It is hereby certified that this is the bonafide record of work done

by Mr./Ms. **JOSEPH FRANKLIN S** during the odd

semester of the academic year 2023-2024 and submitted for the University Practical Examination held on .

## Faculty-in-charge Program Coordinator

**Name:**

## Examiner

**TABLE OF CONTENTS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **Date** | **Name of the Exercise** | **Page No.** | **Marks** | **Signature** |
| 1 | 18/07/2023 | Write a test case for a python function. |  |  |  |
| 2 | 25/07/2023 | Configuring continuous integration with Github actions. |  |  |  |
| 3 | 01/08/2023 | Create a container to run ML model. |  |  |  |
| 4 | 08/08/2023 | Permutation Feature Importance. |  |  |  |
| 5 | 12/08/2023 | Serving a trained model over HTTP. |  |  |  |
| 6 | 05/09/2023 | Continuous delivery for machine learning models. |  |  |  |
| 7 | 12/09/2023 | Creating a model using AutoML |  |  |  |
| 8 | 03/10/2023 | Monitor and logging. |  |  |  |
| 9 | 10/10/2023 | Machine learning interoperability. |  |  |  |
| 10 | 17/10/1023 | Machine Learning command line workflows. |  |  |  |

|  |  |
| --- | --- |
| **Exp Name** | **WRITE TEST CASE FOR PYTHON FUNCTION** |
| **Date** | **01-08-2023** |

**AIM:**

To write test cases for Python functions

**DESCRIPTION:**

In software engineering, a test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

It will lay out particular variables that QAs need to compare expected and actual results to conclude if the feature works. Test case components mention input, execution, and expected output/response. It tells engineers what to do, how to do it, and what results are acceptable.

The Objective of Writing Test Cases in Software Testing

* To validate specific features and functions of the software.
* To guide testers through their day-to-day hands-on activity.
* To record a catalog of steps undertaken, which can be revisited in the event of a bug popping up.
* To provide a blueprint for future projects and testers so they don’t have to start work from scratch.
* To help detect usability issues and design gaps early on.
* To help new testers and devs quickly pick up testing, even if they join in the middle of an ongoing project.

**ALGORITHM:**

Step 1: Start the program

Step 2: Create the function to be tested.

Step 3: Generate a few sample input-output pairs to test the function.

Step 4: Run the inputs against the function and verify if the function matches.

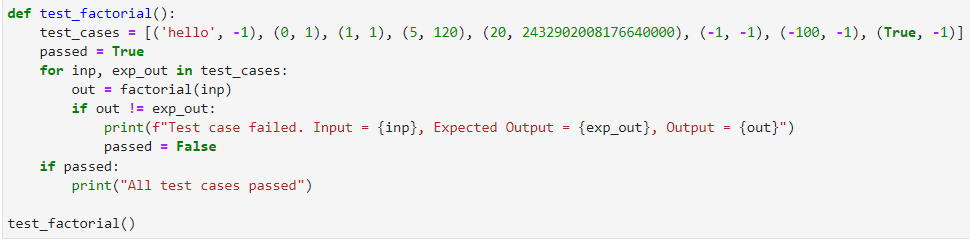
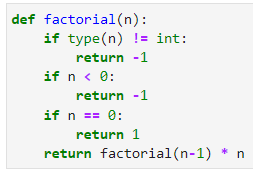
Step 5: If there is a mismatch, inform the developer.

Step 6: If everything matches, the case is complete.

Step 7: Stop the program

**SOURCE CODE:**

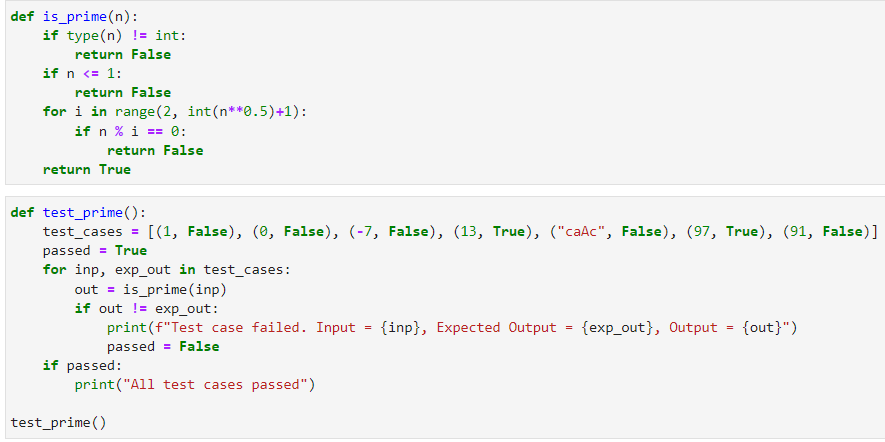
1. Write a test case for a Python function that calculates the factorial of a given number. Ensure that the test case covers both positive and negative input values, including zero.



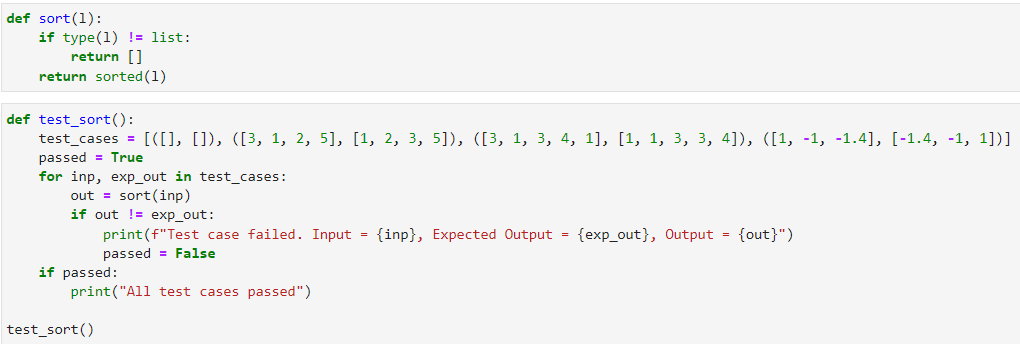
1. Write a test case for a Python function that checks whether a given string is a palindrome. Consider test cases for strings with even and odd lengths, as well as cases with different types of characters (letters, digits, special characters).



1. Write a test case for a Python function that determines whether a given number is prime. Create test cases for both prime and non-prime numbers, including negative numbers and zero.



1. Write a test case for a Python function that sorts a list of integers in ascending order. Consider test cases with different sizes of lists, including empty lists and lists with repeated elements.



**OUTPUT:**

1)



2)



3)



4)



**RESULT:**

The python program to test cases for python functions has been implemented and verified successfully.

|  |  |
| --- | --- |
| **Exp Name** | **CONFIGURING CONTINUOUS INTEGRATION WITH GITHUB ACTIONS** |
| **Date** | **08-08-2023** |

**AIM:**

To configure Continuous Integration with Github Actions

**DESCRIPTION:**

**Continuous Integration**

Continuous Integration is a software development practice that integrates code into a shared repository frequently. This is done by developers several times a day each time they update the codebase. Each of these integrations can then be tested automatically.

It is a process in devops where changes are merged into a central repository after which the code is automated and tested. The continuous integration process is a practice in software engineering used to merge developers' working copies several times a day into a shared mainline.

It refers to the process of automating the integration of code changes coming from several sources. The process comprises several automation tools that emphasize on the code’s correctness before Integration.

**Github**

* Github is a version control system.
* Github is a code hosting platform for version control and collaboration.
* It lets you and other work together on projects from anywhere
* Git means Global Information tracker
* Git allows multiple developers to work on a project simultaneously while ensuring that their changes do not interfere with one another
* Github is delivered through a software as a service(SaaS) business model which was started in 2008

**Github Commands**

* git config –global user.name “Andamma”02
* git config –global user.email “ashwathy@karunya.edu”
* git config –global user.password “123456”
* git init
* git pull copied link
* git add .
* git commit –m “changes made”
* git remote add origin copied link
* git remote –v
* git push –u origin master

**ALGORITHM:**

Step 1: Start the program

Step 2: Create a Github Repository with README.md.

Step 3: Clone the Github Repository to local storage.

Step 4: Update the repository.

Step 5: Commit the updated repository.

Step 6: Push the updated repository to the remote server.

Step 7: Stop the program

**SOURCE CODE:**

git config --global user.name “joseph\_12”

git config --global user.email “josephfranklin@karunya.edu.in”

mkdir mlops

git clone https://github.com/ joseph\_12/Locos.git

cd Locos

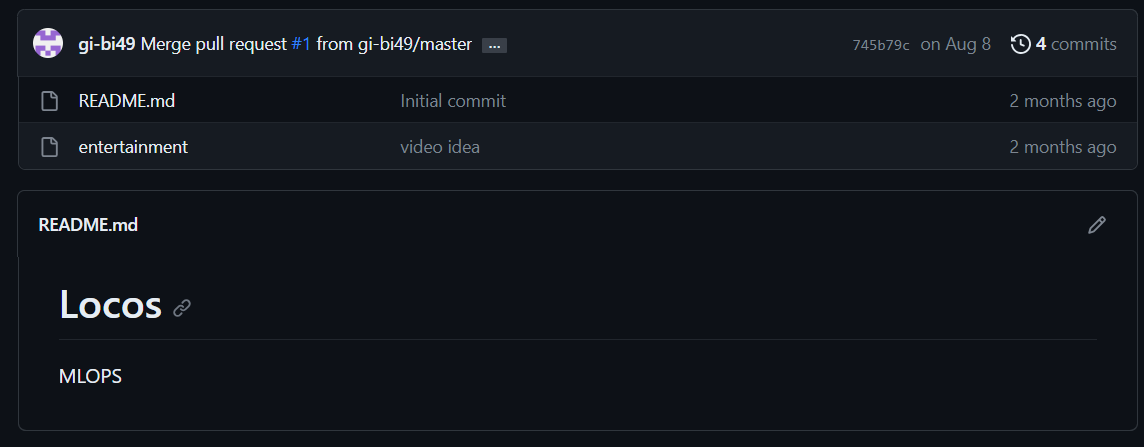
echo "MLOPS" > entertainment

git add .

git commit -m "Updated entertainment

git push origin main (main is the name of the branch)

**OUTPUT:**

****

**RESULT:**

We have successfully configured continuous integration with github actions.

|  |  |
| --- | --- |
| **Exp Name** | **CREATE A CONTAINER TO RUN ML MODEL** |
| **Date** | **12-08-2023** |

**AIM:**

To create a container to run ML model.

**DESCRIPTION:**

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers – images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging

Docker containers that run on Docker Engine:

* Standard: Docker created the industry standard for containers, so they could be portable anywhere
* Lightweight: Containers share the machine’s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
* Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

**ALGORITHM:**

Step 1: Define the ML model and its dependencies:

Step 2: Create a Dockerfile.

Step 3: Build the container image.

Step 4: Run the container.

Step 5: Stop the program

**SOURCE CODE:**

**DOCKERFILE**

FROM python:3.9

WORKDIR /app

COPY . .

RUN pip install scikit-learn pandas numpy joblib

RUN python train.py

CMD ["python", "inference.py"]

**train.py**

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import numpy as np

import os

from sklearn.linear\_model import LogisticRegression

import joblib

# Load the iris dataset

iris\_df = load\_iris()

X = iris\_df.data

y = iris\_df.target

# Perform train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1)

# Create a directory for data if it doesn't exist

data\_dir = "./data"

model\_dir = './model'

os.mkdirs(data\_dir, exist\_ok = True)

os.mkdirs(model\_dir, exist\_ok = True)

# Create and train the logistic regression model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Save the trained model

joblib.dump(model, os.path.join(model\_dir, "logistic\_model.joblib"))

print("Training complete")

**inference.py**

# Necessary Imports

import joblib

import pandas as pd

# Load the trained model

model = joblib.load("./model/logistic\_model.joblib")

# Get user inputs

sepalLength = float(input("Enter sepal Length: "))

sepalWidth = float(input("Enter sepal Width: "))

petalLength = float(input("Enter petal Length: "))

petalWidth = float(input("Enter petal width: "))

# Convert to data point (np.ndarray)

user\_input = [[sepalLength, sepalWidth, petalLength, petalWidth]]

# Make predictions on class

predictions = model.predict(user\_input)

classes = ["Iris-Setosa", "Iris-Versicolor", "Iris-Virginica"]

print("Predicted class is,", classes[predictions[0]])

print("Inference complete")

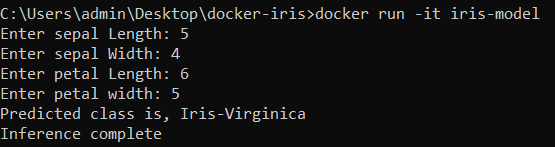
**#Terminal code**

**docker build -t my\_app .**

**docker run -it my\_app**

**OUTPUT:**

Run inference.py



**RESULT:**

The python program to run ML model on container has been implemented and verified successfully.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **URK20AI1005**   |  |  |  |  | | --- | --- | --- | --- | | **Exp Name** | | **Permutation Feature Importance Explainer** | | | **Exp No.** | **4** | **Date** | **17-10-2023** |     **AIM:** To implement Permutation Feature Importance(PFI) Explainer  **DESCRIPTION:**  **Model Explainability**  Model explainability refers to the concept of being able to understand the machine learning model. For example – If a healthcare model is predicting whether a patient is suffering from a particular disease or not. The medical practitioners need to know what parameters the model is taking into account or if the model contains any bias. So, it is necessary that once the model is deployed in the real world. Then, the model developers can explain the model.  **Permutation Feature Importance (PFI)**  The Permutation Feature Importance(PFI) is defined to be the decrease in a model score when a single feature value is randomly shuffled. It measures the increase in the prediction error of the model after we permuted the feature’s values, which breaks the relationship between the feature and the true outcome. This procedure breaks the relationship between the feature and the target, thus the drop in the model score is indicative of how much the model depends on the feature.  **ALGORITHM:**  Step 1: Start the Program  Step 2: Load the dataset and perform train-test split.  Step 3: Train the model with any model.  Step 4: Apply Permutation Feature Importance on each feature to identify the most important features.  Step 5: Display the plot of the most important features.  Step 6: Stop the program.  **SOURCE CODE:**  import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  from sklearn.ensemble import RandomForestClassifier  from sklearn.metrics import accuracy\_score  from sklearn.inspection import permutation\_importance  from sklearn.datasets import load\_iris  iris = load\_iris()  X =pd.DataFrame(iris.data,columns = iris.feature\_names)  Y=iris.target  X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)  model = RandomForestClassifier(n\_estimators=100,random\_state=42)  model.fit(X\_train,Y\_train)  y\_pred=model.predict(X\_test)  acc=accuracy\_score(Y\_test,y\_pred)  print("Accuracy before permutation",acc)  perm\_importance = permutation\_importance(model,X\_test,Y\_test,n\_repeats=30,random\_state=42)  feature\_importance = perm\_importance.importances\_mean  feature\_names = X.columns  for feature\_name,importance in zip(feature\_names,feature\_importance):      print(f"{feature\_name}:{importance:}")  import matplotlib.pyplot as plt  plt.barh(feature\_names,feature\_importance)  **Train the Model**  import numpy as np  from sklearn.model\_selection import train\_test\_split from sklearn.ensemble import RandomForestClassifier from sklearn.metrics import accuracy\_score  from sklearn.inspection import permutation\_importance from sklearn.datasets import load\_iris |

**URK20AI1005**

iris = load\_iris()

X = pd.DataFrame(iris.data, columns = iris.feature\_names)

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = R

andomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy before permutation: {accuracy \* 100:.2f}%")

perm\_importance = permutation\_importa

nce(model, X\_test, y\_test, n\_repeats=30,

random\_state=42)

feature\_importances = perm\_importance.importances\_mean

feature\_names = X.columns

for feature\_name, importance in zip(feature\_names, feature\_importances):

print(f"{feature\_name}: {importance:.4f}

")

import matplotlib.pyplot as plt

plt.barh(feature\_names, feature\_importances)

plt.xlabel('Permutation Importance')

plt.title('Permutation Feature Importance')

plt.show()

**OUTPUT:**

**RESULT:**

The python program to

implement

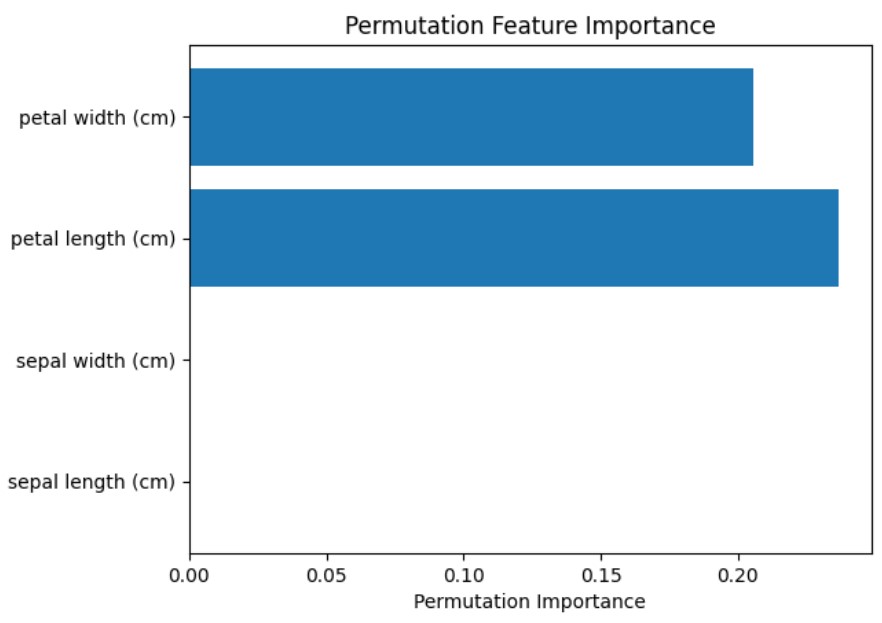
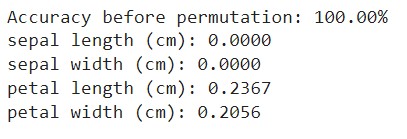
Permutation Feature

Importance (PFI)

Explainer

has been

implemented and verified successfully.



|  |  |
| --- | --- |
| **Exp Name** | **SERVING A TRAINED MODEL OVER HTTP** |
| **Date** | **12-09-2023** |

**AIM:**

To serve a trained model over http using python.

**DESCRIPTION:**

Serving a trained model over HTTP refers to the process of deploying a machine learning model so that it can be accessed and utilized by other software applications or clients through standard HTTP requests and responses.

This approach allows you to make your machine learning models accessible via APIs (Application Programming Interfaces) over the internet or within your organization's network, making it easier to integrate machine learning capabilities into various applications, including web and mobile applications.

It marks the transformation of complex algorithms and statistical models into user-friendly tools that can be accessed by web and mobile applications.

By offering machine learning as a service over the HTTP protocol, we empower organizations and individuals to leverage the potential of AI for a myriad of applications.

**ALGORITHM:**

Step 1: Define the ML model and its dependencies.

Step 2. Build a Flask app to expose the model using HTTP request.

Step 3: Create a Dockerfile.

Step 4: Build the container image and expose any available port.

Step 5: Run the container.

Step 6: Test the API endpoint using a HTTP request.

Step 7: Stop the program

**SOURCE CODE:**

**DOCKERFILE**

FROM python:3.7

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

RUN python train.py

EXPOSE 5000

CMD ["python", "app.py"]

**train.py**

import sklearn.datasets as datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

from sklearn.linear\_model import LinearRegression

import pandas as pd

import joblib

import os

df = pd.read\_csv('California\_Houses.csv')

df.head()

df.dropna(axis=0, inplace=True)

X = df.iloc[:, 1:7]

y = df.iloc[:, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1)

model = LinearRegression()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print("Training complete!")

joblib.dump(model, "model.joblib")

**app.py**

from flask import Flask, request, jsonify

import joblib

import numpy as np

app = Flask(\_\_name\_\_)

clf = joblib.load("model.joblib")

@app.route("/predict", methods = ["POST"])

def predict():

data = request.get\_json()

new\_sample = np.array(data["data"])

prediction = clf.predict(new\_sample)

return jsonify({"Expected Price" : prediction[0]})

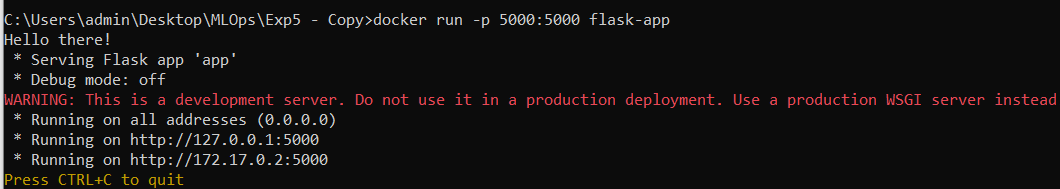
if \_\_name\_\_ == "\_\_main\_\_":

print("Hello there!")

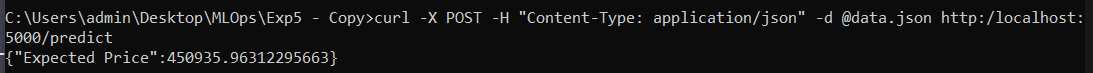
app.run(host='0.0.0.0', port=5000)

**OUTPUT:**

Run the server



HTTP Request



HTTP Request(on server’s side)



**RESULT:**

The python program to serve ML model over HTTP request has been implemented and verified successfully.

|  |  |
| --- | --- |
| **Exp Name** | **Continuous Integration (CI/CD) for ML pipelines using Github actions** |
| **Date** | **19-09-2023** |

**AIM:**

To implement Continuous Integration (CI/CD) for ML pipelines using Github actions

**DESCRIPTION:**

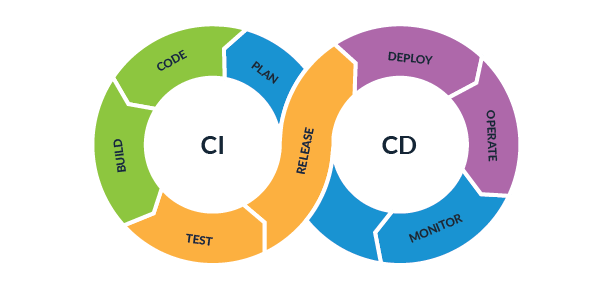
**Github**

* Github is a version control system
* Github is a code hosting platform for version control and collaboration.
* It lets you and other work together on projects from anywhere
* Git means Global Information tracker
* Git allows multiple developers to work on a project simultaneously while ensuring that their changes do not interfere with one another
* Github is delivered through a software as a service (SaaS) business model which was started in 2008

**CI/CD Pipeline**

A continuous integration and continuous deployment (CI/CD) pipeline is a series of steps that must be performed in order to deliver a new version of software. CI/CD pipelines are a practice focused on improving software delivery throughout the software development life cycle via automation.

By automating CI/CD throughout development, testing, production, and monitoring phases of the software development lifecycle, organizations are able to develop higher quality code, faster. Although it’s possible to manually execute each of the steps of a CI/CD pipeline, the true value of CI/CD pipelines is realized through automation.

  
 CICD Pipeline

**ALGORITHM:**

Step 1: Define the ML model and its dependencies and upload it to a Github Repository.

Step 2. Go to Settings -> Actions -> General -> Enable Read and Write Permissions.

Step 3: Create a cml.yaml file in .github/workspaces folder.

Step 4: Define steps to run the train.py script and send a report within the cml.yaml file.

Step 5: Github will automatically run the scripts and send report.

Step 6: Make any changes to your file and commit. You should receive a report to your mail.

**SOURCE CODE:**

**train.py**

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

from sklearn.neural\_network import MLPRegressor

import pandas as pd

import joblib

import os

df = pd.read\_csv('California\_Houses.csv')

df.head()

df.dropna(axis=0, inplace=True)

X = df.iloc[:, 1:7]

y = df.iloc[:, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1)

model = MLPRegressor()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print("Training complete!")

joblib.dump(model, "model.joblib")

with open("metrics.txt", 'w') as fw:

fw.write(f"Mean Squared Error of current model is: {mean\_squared\_error(y\_test, y\_pred)}")

**cml.yaml**

name: mlops-exp6

on: [push]

jobs:

run:

runs-on: ubuntu-latest

container: ghcr.io/iterative/cml:0-dvc2-base1

steps:

- uses: actions/checkout@v3

- uses: iterative/setup-cml@v1

- name: Train model

run: |

# Your ML workflow goes here

pip install -r requirements.txt

python train.py

- name: Write CML report

env:

REPO\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

run: |

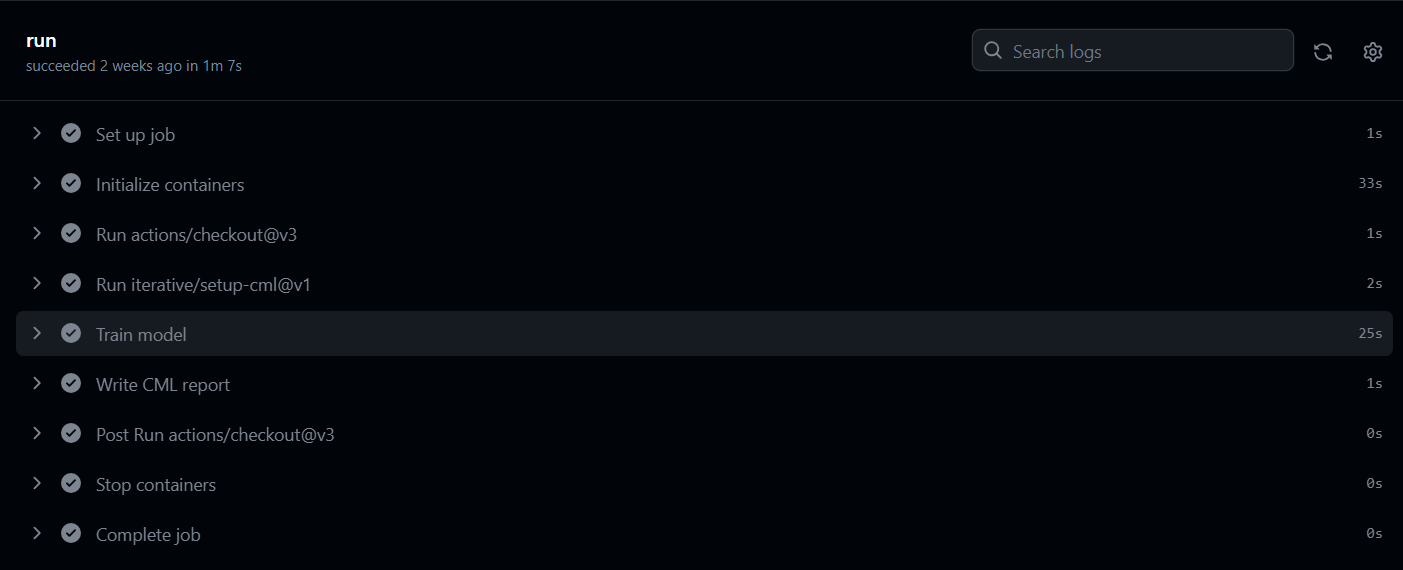
# Post reports as comments in GitHub PRs

cat metrics.txt >> report.md

cml comment create report.md

**OUTPUT:**

Run Workflow



**RESULT:**

The python program to implement CI/CD pipeline using Github actions has been implemented and verified successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp Name** | | **CREATING A MODEL USING AutoML** | |
| **Exp No.** | **7** | **Date** | **03-10-2023** |

**AIM:** To create a Machine Learning model using AutoML

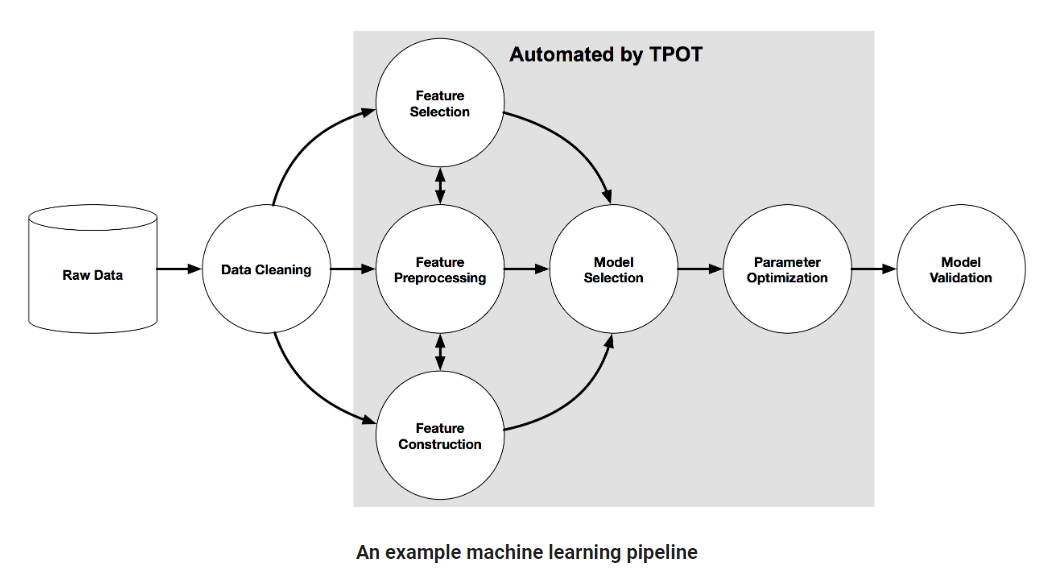
**DESCRIPTION:**

Automated Machine Learning provides methods and processes to make Machine Learning available for non-Machine Learning experts, to improve efficiency of Machine Learning and to accelerate research on Machine Learning.

AutoML automatically performs the following steps:

* Preprocess and clean the data.
* Select and construct appropriate features.
* Select an appropriate model family.
* Optimize model hyperparameters.
* Design the topology of neural networks (if deep learning is used).
* Postprocess machine learning models.
* Critically analyze the results obtained.

TPOT is a Python Automated Machine Learning tool that optimizes machine learning pipelines using genetic programming.



**ALGORITHM:**

Step 1: Start the Program

Step 2: Read the dataset and perform train-test split.

Step 3: Define the TPOT Classifier/Regressor with the requirement parameters.

Step 4: Fit the data to the classifier.

Step 5: Export the model to a file.

Step 6: Stop the program.

**SOURCE CODE:**

**train.py**

from tpot import TPOTClassifier

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

iris = load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)

tpot = TPOTClassifier(

generations=5,

population\_size=20,

verbosity=2,

random\_state=0,

config\_dict='TPOT sparse',

memory='auto',

n\_jobs = -1,

cv=5

)

tpot.fit(X\_train, y\_train)

acc = tpot.score(X\_test, y\_test)

print(f"Accuracy = {acc:.2f}")

tpot.export('best\_model\_pipeline.py')

**Dockerfile**

FROM python:3.8

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

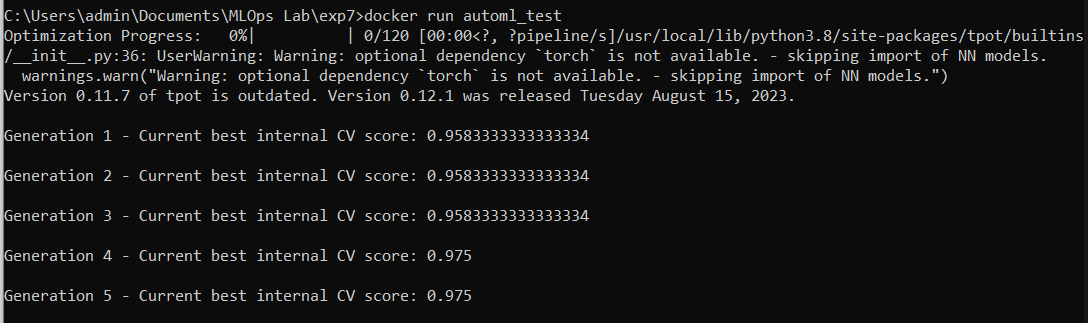
COPY . .

ENV NAME AutoML\_Iris

CMD ["python", "automl\_script.py"]

**OUTPUT:**

Run the container



**RESULT:**

The python program to implement AutoML has been implemented and verified successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp Name** | | **Monitor and Logging** | |
| **Exp No.** | **8** | **Date** | **03-10-2023** |

**AIM:** To implement Monitor and logging in Machine Learning with python

**DESCRIPTION:**

**Logging** is a method of tracking and storing data to ensure application availability and to assess the impact of state transformations on performance. The purpose of logging is to create an ongoing record of application events. Log files can be used to review any event within a system, including failures and state transformations. Consequently, log messages can provide valuable information to help pinpoint the cause of performance problems.  Log data can help DevOps teams troubleshoot issues by identifying which changes resulted in error reporting, but is only as valuable as the information it contains.

**Monitoring** is a diagnostic tool used for alerting DevOps to system-related issues by analyzing metrics. Monitoring metrics is like the security alarm that alerts you to a possible intrusion; log files act as the security camera footage that will provide clues to tell you what happened and how.

**ALGORITHM:**

Step 1: Start the Program

Step 2: Read the dataset and perform train-test split.

Step 3: Import mlflow and the other required libraries.

Step 4: Train the model and log the accuracy of the model at each epoch.

Step 5: Export the model to a file.

Step 6: Stop the program.

**SOURCE CODE:**

**train.py**

import mlflow

import mlflow.sklearn

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.neural\_network import MLPRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generate some sample data

np.random.seed(0)

X = np.random.rand(100, 1)

y = 2 \* X[:, 0] + 1 + 0.1 \* np.random.randn(100)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Start an MLflow nested run

with mlflow.start\_run(nested=True):

    # Define hyperparameters

    hidden\_layer\_sizes = (100, 50)  # Two hidden layers with 100 and 50 neurons

    learning\_rate\_init = 0.001

    max\_iter = 500

    # Create and train the MLPRegressor

    model = MLPRegressor(hidden\_layer\_sizes=hidden\_layer\_sizes, learning\_rate\_init=learning\_rate\_init, max\_iter=max\_iter)

    mse\_list = []  # List to store MSE values at each epoch

    mlflow.log\_params({

        "hidden\_layer\_sizes": hidden\_layer\_sizes,

        "learning\_rate\_init": learning\_rate\_init,

        "max\_iter": max\_iter

    })

    for epoch in range(max\_iter):

        model.partial\_fit(X\_train, y\_train)

        y\_pred = model.predict(X\_test)

        mse = mean\_squared\_error(y\_test, y\_pred)

        mse\_list.append(mse)

        # Log metrics at each epoch

        mlflow.log\_metric("epoch\_mse", mse, step=epoch)

    # Save the final model

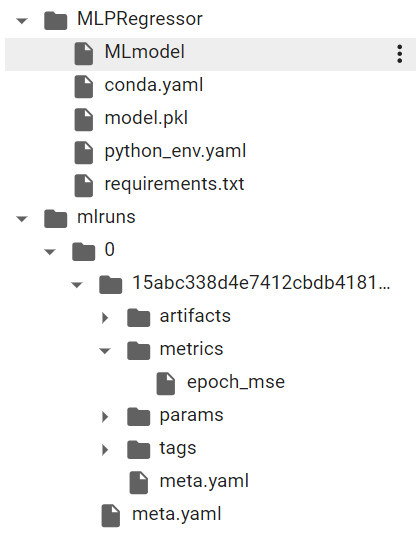
    mlflow.sklearn.save\_model(model, "MLPRegressor")

# End the nested MLflow run

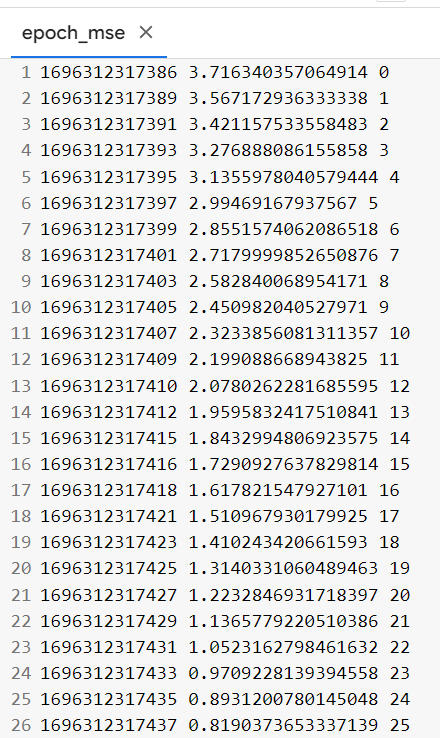
mlflow.end\_run()

**OUTPUT:**

Model and Metrics



Epochs MSE



**RESULT:**

The python program to implement Monitoring and Logging for ML model has been implemented and verified successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp Name** | | **Machine Learning Interoperability** | |
| **Exp No.** | **9** | **Date** | **10-10-2023** |

**AIM:** To implement Machine Learning Interoperability with ONNX

**DESCRIPTION:**

Machine learning interoperability refers to the ability of different machine learning systems, models, or components to work together seamlessly

Model Interoperability: Ensuring that machine learning models trained in one framework or platform can be used in another. Common model formats like ONNX (Open Neural Network Exchange) are designed to facilitate model interoperability.

**ONNX**

ONNX, which stands for Open Neural Network Exchange, is an open-source format and ecosystem designed to enable interoperability among various deep learning frameworks and machine learning tools.

It allows you to represent machine learning models, including deep neural networks, in a standardized format, making it easier to share, deploy, and run models across different platforms and frameworks.

ONNX is particularly valuable when you want to move models between different libraries or use them in production environments.

**ALGORITHM:**

Step 1: Start the Program

Step 2: Load the dataset and perform train-test split.

Step 3: Train the model with any model.

Step 4: Convert the model to ONNX format with any library.

Step 5: Save the ONNX file.

Step 6: Load the ONNX file using the Runtime library.

Step 7: Configure the Runtime library and test the model against custom input

Step 8: Stop the program.

**SOURCE CODE:**

**Train the Model**

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score

import onnx

from skl2onnx import convert\_sklearn

from skl2onnx.common.data\_types import FloatTensorType

iris = load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = MLPClassifier(hidden\_layer\_sizes=(64, 32), max\_iter=1000, random\_state=42)

model.fit(X\_train, y\_train)

initial\_types = [('input', FloatTensorType([None, X\_train.shape[1]]))]

onnx\_model = convert\_sklearn(model, initial\_types=initial\_types)

onnx.save\_model(onnx\_model, 'iris\_model.onnx')

**Test ONNX Model**

import onnx

import onnxruntime as ort

import numpy as np

# Create an ONNX Runtime session with the specified providers

providers = ['CPUExecutionProvider']

ort\_session = ort.InferenceSession("iris\_model.onnx", providers=providers)

input\_data = np.array([[5.1, 3.5, 1.4, 0.2],

                       [6.3, 2.8, 5.1, 1.5]], dtype=np.float32)

# Run inference using ONNX Runtime

predictions = ort\_session.run(None, {"input": input\_data})

print("Predictions:", predictions)

class\_labels = ["Iris-Setosa", "Iris-Versicolor", "Iris-Virginica"]

predicted\_labels = [class\_labels[max(prediction, key=lambda k : prediction[k])] for prediction in predictions[1]]

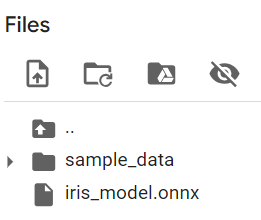
print("Predicted Class Labels:", predicted\_labels)

**OUTPUT:**

ONNX Model Inference



Model File



**RESULT:**

The python program to implement Machine Learning Interoperability with ONNX has been implemented and verified successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp Name** | | **Machine Learning Command Line Workflows** | |
| **Exp No.** | **10** | **Date** | **17-10-2023** |

**AIM:** To implement Machine Learning Command Line Workflows.

**DESCRIPTION:**

**Import Libraries**

The script begins by importing several Python libraries that are necessary for the tasks it performs. These libraries include Click for creating the command-line interface, pandas for working with data in a tabular format, scikit-learn for machine learning, and joblib for saving and loading machine learning models..

**Define CLI Commands:**

The script defines two CLI commands –train and predict. These commands can be executed from the command line

‘train’ command is used to train a machine learning model on the Iris dataset. It takes several optional parameters like

* ‘test\_size’ – for specifying the size of the test data.
* ‘n\_estimators’ – for the number of trees in a random forest
* ‘max\_depth’ – for the maximum depth of the trees
* ‘model\_output’ – for the name of the file to save the trained model

**ALGORITHM:**

Step 1: Start the Program Step 2: Import required libraries.

Step 3: Define CLI commands.

Step 4: Load the dataset and perform train test split.

Step 5: Train the model.

Step 6: Save the model and perform inference on the model.

Step 7: Use the command line interface to interact with the model.

Step 8: Stop the program.

**SOURCE CODE:**

import click import pandas as pd from sklearn.model\_selection import train\_test\_split from sklearn.ensemble import RandomForestClassifier import joblib

from sklearn.datasets import load\_iris

@click.group() def cli():

"""Machine Learning Command Line Workflows""" pass

@click.command()

@click.option('--test\_size', default=0.2, help='Test set size (default: 0.2)')

@click.option('--n\_estimators', default=100, help='Number of estimators in RandomForest (default: 100)')

@click.option('--max\_depth', default=None, help='Maximum depth of the tree(default:

None)')

@click.option('--model\_output', default='model.pkl', help='Output filename for the trained model (default: model.pkl)') def train(test\_size, n\_estimators, max\_depth, model\_output):

"""Train a machine learning model on the Iris dataset."""

# Ensure max\_depth is an integer or None

max\_depth = int(max\_depth) if max\_depth is not None else None

# Load the Iris dataset iris = load\_iris() X = iris.data y = iris.target

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_size, random\_state=42)

# Train a model (Random Forest)

model = RandomForestClassifier(n\_estimators=n\_estimators, max\_depth=max\_depth) model.fit(X\_train, y\_train) # Save the trained model joblib.dump(model, model\_output)

print(f"Model trained and saved to {model\_output}")

@click.command()

@click.option('--output\_file', default='output\_predictions.csv', help='Output filename for predictions (default: output\_predictions.csv)')

@click.option('--model\_path', default = 'model.pkl', help = 'Input filename for the model

(default: model.pkl)') def predict(output\_file, model\_path):

"""Make predictions using a trained model on the Iris dataset."""

# Load the trained model try:

model = joblib.load(model\_path) except FileNotFoundError: print(f"Error: Trained model '{model\_path}' not found. Please train a model first using the 'train' command.")

return

# Load the Iris dataset iris = load\_iris() X = iris.data # Make predictions

predictions = model.predict(X) # Save predictions to an output file

pd.DataFrame({'prediction': predictions}).to\_csv(output\_file, index=False) print(f"Predictions saved to {output\_file}")

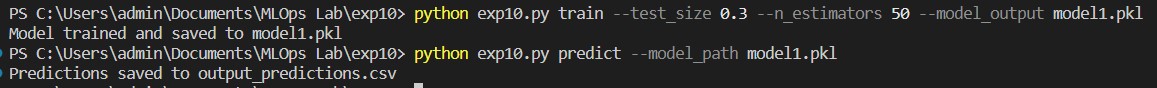
cli.add\_command(train)

cli.add\_command(predict)

if \_\_name\_\_ == "\_\_main\_\_":

cli()

**OUTPUT:**



**RESULT:**

The python program to implement Machine Learning Command Line Workflows has been implemented and verified successfully.