

**EX2: CONFIGURING CONTINUOUS INTEGRATION WITH GITHUB ACTIONS**

**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

**EX3: CREATE A CONTAINER TO RUN ML MODEL**

**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

**Ex4: Permutation Feature Importance Explainer**

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)

**ex5: SERVING A TRAINED MODEL OVER HTTP**

**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"])

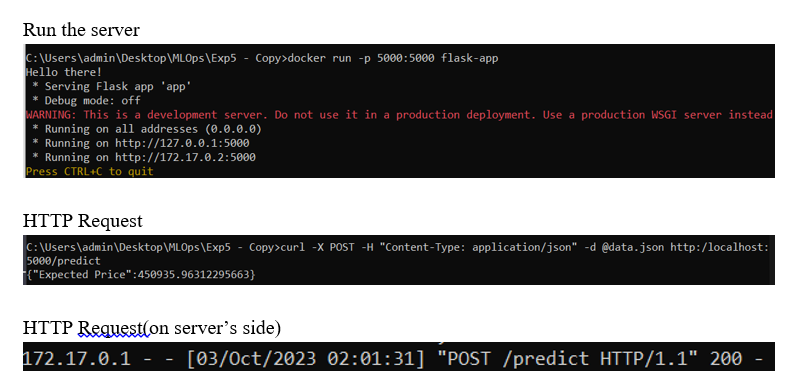
prediction = clf.predict(new\_sample)

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

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

print("Hello there!")

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

****

**Ex7: CREATING A MODEL USING AutoML**

**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"]

**Ex8: Monitor and Logging**

**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

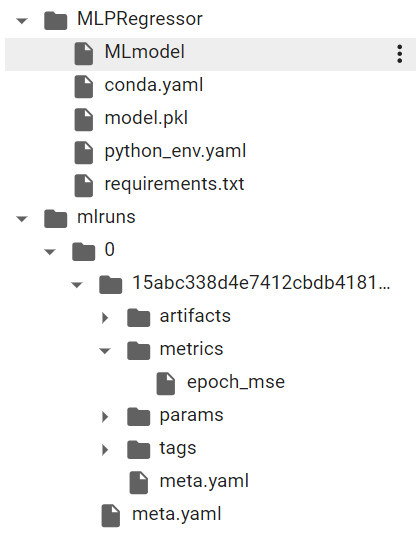
        mlflow.log\_metric("epoch\_mse", mse, step=epoch)

    # Save the final model

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

# End the nested MLflow run

mlflow.end\_run()



**Ex9: Machine Learning Interoperability**

**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)

**exp10: Machine Learning Command Line Workflows**

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()

