```
import numpy as np
import arviz as az
import pymc as pm
import math
import pickle

from learning_function_library import *
from plxscripting.easy import *
```

Definition of PLAXIS Functions

```
def validate_and_modify(material, g_i):
    validate_command = g_i.validate(material)

    while validate_command != 'The material definition is valid.':
        parts = validate_command.split()

        if 'Error: EOedRef = ' in validate_command:
            for i in range(len(parts)):
                if parts[i] == '>=':
                    material_property_type = parts[i-1].strip('.,:!')
                    suggested_value =
float(parts[i+1].strip('.,:!'))*1.01
                    if material_property_type != 'K0NC':
                        g_i.sps(material, material_property_type,
suggested_value)
                    else:
                        phi = math.degrees(np.arcsin(1-
suggested_value))
                        psi = np.max(phi - 30, 0)
                        g_i.sps(material, 'phi', phi, 'psi', psi)

                elif parts[i] == '<=':
                    material_property_type = parts[i-1].strip('.,:!')
                    suggested_value =
float(parts[i+1].strip('.,:!'))*0.99
                    if material_property_type != 'K0NC':
                        g_i.sps(material, material_property_type,
suggested_value)
                    else:
                        phi = math.degrees(np.arcsin(1-
suggested_value))
                        psi = np.max(phi - 30, 0)
                        g_i.sps(material, 'phi', phi, 'psi', psi)

        elif validate_command == 'Error: Use EOedRef larger than
0.1*E50Ref.':
            g_i.sps(material, 'EOedRef', 1.01*0.1*material.E50Ref)
```

```
        elif validate_command == 'Error: Use EOedRef smaller than
3*E50Ref.':
            g_i.sps(material, 'EOedRef', 0.99*3*material.E50Ref)

        elif validate_command == 'Use lambda greater than kappa.':
            g_i.sps(material, 'lambda',
np.random.normal(5,1)*material.kappa)

        validate_command = g_i.validate(material)

    return

def initialize_PLAXIS():
    s_i, g_i = new_server('localhost', 10000, password =
'Mypassword1')

    g_i.clear()

    g_i.SoilContour.initializerectangular(0, -50, 150, 10)

    clay_params = [('SoilModel', 'Modified Cam-Clay'),
                   ('Identification', 'Clay'),
                   ('gammaUnsat', 17),
                   ('gammaSat', 17),
                   ('nuUR', 0.25),
                   ('GroundwaterSoilClassStandard', 'Fine'),
                   ('CInter', 15)]

    sand_params = [('SoilModel', 'Hardening Soil'),
                   ('Identification', 'Sand'),
                   ('gammaUnsat', 18),
                   ('gammaSat', 20),
                   ('nuUR', 0.2),
                   ('PowerM', 0.5),
                   ('GroundwaterSoilClassStandard', 'Coarse')]

    embankment_params = [('SoilModel', 'Hardening Soil'),
                   ('Identification', 'Embankment'),
                   ('gammaUnsat', 18),
                   ('gammaSat', 20),
                   ('E50Ref', 75000),
                   ('nuUR', 0.2),
                   ('PowerM', 0.5),
                   ('cRef', 2),
                   ('phi', 35),
                   ('psi', 5),
                   ('GroundwaterSoilClassStandard', 'Coarse'),
                   ('PermHorizontalPrimary', 1e-3*86400),
                   ('PermVertical', 1e-3*86400)]
```

```
    concrete_params = [('MaterialType', 'Elastic'),
                ('Identification', 'Concrete'),
                ('Gamma', 24),
                ('LSpacing', 4),
                ('PredefinedCrossSectionType', 'Solid square beam'),
                ('Width', 0.4),
                ('E', 30000000),
                ('AxialSkinResistance', 'Layer dependent')]

    clay_MCC = g_i.soilmat(*clay_params)
    sand_Hardening = g_i.soilmat(*sand_params)
    embankment_Hardening = g_i.soilmat(*embankment_params)
    concrete = g_i.embeddedbeammat(*concrete_params)

    g_i.borehole(0)
    g_i.Boreholes[0].Head = -2

    g_i.soillayer(2)
    g_i.soillayer(13)

    g_i.Soillayers[0].Soil.Material = sand_Hardening
    g_i.Soillayers[1].Soil.Material = clay_MCC

    g_i.polygon((53, 0), (56, 1), (94, 1), (97, 0))
    g_i.polygon((56, 1), (59, 2), (91, 2), (94, 1))
    g_i.polygon((59, 2), (62, 3), (88, 3), (91, 2))
    g_i.polygon((62, 3), (65, 4), (85, 4), (88, 3))

    for polygon in g_i.Polygons:
        polygon.Soil.Material = embankment_Hardening

    for x in ([55, 59, 63, 67, 71, 75, 79, 83, 87, 91, 95]):
        g_i.embeddedbeam((x, 0), (x, -8))

    for line in g_i.Lines:
        line.EmbeddedBeam.Material = concrete

    g_i.gotomesh()
    g_i.mesh(0.01)

    g_i.gotostages()

    InitialPhase = g_i.Phases[0]

    PilePhase = g_i.phase(InitialPhase)
    g_i.sps(PilePhase, 'Identification', 'Pile', 'DeformCalcType',
'Plastic')
    g_i.activate(g_i.Lines, PilePhase)

    EmbankmentPhase1 = g_i.phase(PilePhase)
```

```
    g_i.sps(EmbankmentPhase1, 'Identification', 'Embankment1',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[3], EmbankmentPhase1)

    EmbankmentPhase2 = g_i.phase(EmbankmentPhase1)
    g_i.sps(EmbankmentPhase2, 'Identification', 'Embankment2',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[2], EmbankmentPhase2)

    EmbankmentPhase3 = g_i.phase(EmbankmentPhase2)
    g_i.sps(EmbankmentPhase3, 'Identification', 'Embankment3',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[1], EmbankmentPhase3)

    EmbankmentPhase4 = g_i.phase(EmbankmentPhase3)
    g_i.sps(EmbankmentPhase4, 'Identification', 'Embankment4',
'DeformCalcType', 'Consolidation', 'TimeInterval', 2)
    g_i.activate(g_i.Polygons[0], EmbankmentPhase4)

    EndOfConsolidationPhase = g_i.phase(EmbankmentPhase4)
    g_i.sps(EndOfConsolidationPhase, 'Identification',
'EndOfConsolidation', 'DeformCalcType', 'Consolidation',
'TimeInterval', 18250)

    localhostport_o = g_i.selectmeshpoints()
    s_o, g_o = new_server('localhost', localhostport_o, password =
'Mypassword1')
    g_o.addcurvepoint('Node', 75, 4)
    g_o.update()

    return g_i

def true_function_PLAXIS(X, g_i):

    phi = X[0]
    c = X[1]
    E = X[2]
    M_MCC = X[3]
    lambda_MCC = X[4]
    kappa_MCC = X[5]
    k_clay = X[6]
    k_sand = X[7]
    OCR = X[8]

    # Calculate other paramaters based on given
    psi = np.max(phi - 30, 0)

    clay_params = [ ('lambda', lambda_MCC),
                    ('kappa', kappa_MCC),
                    ('M', M_MCC),
```

```
                    ('PermHorizontalPrimary', k_clay*86400),
                    ('PermVertical', k_clay*86400),
                    ('OCR', OCR)]

    sand_params = [ ('E50Ref', E),
                    ('cRef', c),
                    ('phi', phi),
                    ('psi', psi),
                    ('PermHorizontalPrimary', k_sand*86400),
                    ('PermVertical', k_sand*86400)]


    clay_MCC = g_i.Clay
    sand_Hardening = g_i.Sand

    EndOfConsolidationPhase = g_i.Phases[-1]

    g_i.sps(clay_MCC, *clay_params)
    g_i.sps(sand_Hardening, *sand_params)

    # validate_and_modify(sand_Hardening, g_i)
    # validate_and_modify(clay_MCC, g_i)
    g_i.gotostages()

    for phase in g_i.Phases:
        phase.ShouldCalculate = True

    g_i.calculate()

    phase = g_i.Phases[-1]
    localhostport_o = g_i.view(phase)
    s_o, g_o = new_server('localhost', localhostport_o, password =
'Mypassword1')

    uy_EoC = np.abs(g_o.getcurveresults(g_o.CN_1,
EndOfConsolidationPhase, g_o.ResultTypes.Soil.Uy))
    g_o.close()

    return ((0.15/uy_EoC) - 1)

def kill_Plaxis_2D_Output():
    '''Kills Plaxis 2D Output to refresh memories'''

    try:
        subprocess.run(["taskkill", "/IM", "Plaxis2DOutput.exe",
"/F"], check=True)
    except subprocess.CalledProcessError as e:
        print(f"Failed to terminate Plaxis 2D Output: {e}")
    except Exception as e:
        print(f"An error occurred: {e}")
```

Generation of Stochastic Input Variables

```
# get lognormal moments from normal moments
def get_lognormal_moments(mean, std):
    log_std = np.sqrt(np.log(1+(std/mean)**2))
    log_mean = np.log(mean) - log_std**2/2
    return [log_mean, log_std]

# sample from lognormal distribution within a range
def sample_lognormal_within_range(log_mean, log_std, low, high, size):
    samples = []
    while len(samples) < size:
        sample = np.random.lognormal(log_mean, log_std, size=1)
        if low <= sample <= high:
            samples.append(sample)
    return np.array(samples)

N_pop = 1000000

# define stochastic characteristics of inputs
phi_mean, phi_std = 30, 1.8
c_mean, c_std = 5, 1
phi_clay_mean, phi_clay_std = 22.5, 1.8

lambda_mean, lambda_std = 0.15, 0.02
E_mean, E_std = 50000, 2500

k_clay_mean, k_clay_std = 1.5e-8, 2e-8
k_sand_mean, k_sand_std = 2e-4, 1e-4

OCR_mean, OCR_std = 2, 0.3

phi_log_moments = get_lognormal_moments(phi_mean, phi_std)
c_log_moments = get_lognormal_moments(c_mean, c_std)
phi_clay_log_moments = get_lognormal_moments(phi_clay_mean,
phi_clay_std)

lambda_log_moments = get_lognormal_moments(lambda_mean, lambda_std)
E_log_moments = get_lognormal_moments(E_mean, E_std)

k_clay_log_moments = get_lognormal_moments(k_clay_mean, k_clay_std)
k_sand_log_moments = get_lognormal_moments(k_sand_mean, k_sand_std)

OCR_log_moments = get_lognormal_moments(OCR_mean, OCR_std)

phi_pop = np.random.lognormal(phi_log_moments[0], phi_log_moments[1],
N_pop)
c_pop = np.random.lognormal(c_log_moments[0], c_log_moments[1], N_pop)
phi_clay_pop = np.random.lognormal(phi_clay_log_moments[0],
phi_clay_log_moments[1], N_pop)
M_pop = 6*np.sin(np.radians(phi_clay_pop))/(3-
```

```
np.sin(np.radians(phi_clay_pop)))

lambda_pop = np.random.lognormal(lambda_log_moments[0],
lambda_log_moments[1], N_pop)/np.log(10)
lambda_kappa_ratio = np.random.normal(5, 0.5, N_pop)
kappa_pop = lambda_pop/lambda_kappa_ratio
E_pop = np.random.lognormal(E_log_moments[0], E_log_moments[1], N_pop)

k_clay_pop = np.random.lognormal(k_clay_log_moments[0],
k_clay_log_moments[1], N_pop)
k_sand_pop = np.random.lognormal(k_sand_log_moments[0],
k_sand_log_moments[1], N_pop)

OCR_pop = sample_lognormal_within_range(OCR_log_moments[0],
OCR_log_moments[1], 1, 10000, N_pop)

X_pop = np.hstack((phi_pop.reshape(-1, 1), c_pop.reshape(-1, 1),
E_pop.reshape(-1, 1), M_pop.reshape(-1, 1), lambda_pop.reshape(-1, 1),
kappa_pop.reshape(-1, 1), k_clay_pop.reshape(-1, 1),
k_sand_pop.reshape(-1, 1), OCR_pop.reshape(-1, 1)))

N_initial_DoE = 5
random_indices = np.random.choice(N_pop, size=N_initial_DoE,
replace=False)
initial_X_DoE = X_pop[random_indices]
```

Generation of Initial DoE

```
true_function = true_function_PLAXIS

X_DoE = initial_X_DoE
Y_DoE = []

g_i = initialize_PLAXIS()

for X in X_DoE:
    Y_DoE.append(true_function(X, g_i))
```

Training with Initial DoE

```
with pm.Model() as GP_model:
    # Hyperparameters for the Gaussian Process
    ls_phi = pm.Uniform("ls_phi", lower=np.min(phi_pop)/100,
upper=np.max(phi_pop)*100)
    ls_c = pm.Uniform("ls_c", lower=np.min(c_pop)/100,
upper=np.max(c_pop)*100)
    ls_E = pm.Uniform("ls_E", lower=np.min(E_pop)/100,
upper=np.max(E_pop)*100)
    ls_M_MCC = pm.Uniform("ls_M_MCC", lower=np.min(M_pop)/100,
```

```
upper=np.max(M_pop)*100)
    ls_lambda_MCC = pm.Uniform("ls_lambda_MCC",
lower=np.min(lambda_pop)/100, upper=np.max(lambda_pop)*100)
    ls_kappa_MCC = pm.Uniform("ls_kappa_MCC",
lower=np.min(kappa_pop)/100, upper=np.max(kappa_pop)*100)
    ls_k_clay = pm.Uniform("ls_k_clay", lower=np.min(k_clay_pop)/100,
upper=np.max(k_clay_pop)*100)
    ls_k_sand = pm.Uniform("ls_k_sand", lower=np.min(k_sand_pop)/100,
upper=np.max(k_sand_pop)*100)
    ls_OCR = pm.Uniform("ls_OCR", lower=np.min(OCR_pop)/100,
upper=np.max(OCR_pop)*100)

    cov_scale = pm.Uniform("cov_scale", lower=0.000001, upper=10)
    sigma = pm.Uniform("sigma", lower=0.000001, upper=10)

    # Mean functiom
    mean_func = pm.gp.mean.Zero()

    # Covariance function
    cov_func = cov_scale ** 2 * pm.gp.cov.Matern52(input_dim=9,
ls=[ls_phi, ls_c, ls_E, ls_M_MCC, ls_lambda_MCC, ls_kappa_MCC,
ls_k_clay, ls_k_sand, ls_OCR])

    # GP prior with zero mean
    gp = pm.gp.Marginal(mean_func = mean_func, cov_func = cov_func)

    # GP likelihood
    y_ = gp.marginal_likelihood("y_", X_DoE, Y_DoE, sigma = sigma)

    # obtain MAP estimate and calculate predictive mean and variance
using MAP estimate
    start = {'ls_phi': np.mean(phi_pop), 'ls_c': np.mean(c_pop),
'ls_E': np.mean(E_pop),
             'ls_M_MCC': np.mean(M_pop), 'ls_lambda_MCC':
np.mean(lambda_pop), 'ls_kappa_MCC': np.mean(kappa_pop),
             'ls_k_clay': np.mean(k_clay_pop), 'ls_k_sand':
np.mean(k_sand_pop), 'ls_OCR': np.mean(OCR_pop),
             'cov_scale': 1, 'sigma': 1}

    map_estimate = pm.find_MAP(start = start)
    mean_pop, var_pop = gp.predict(X_pop, map_estimate, diag=True)

Pf, Pf_plus, Pf_minus, conv_criterion = calculate_Pf_MLE(mean_pop,
var_pop)

# store relevant results
hyperparams_list = [map_estimate]
Pf_list = [Pf]
Pf_plus_list = [Pf_plus]
```

```
Pf_minus_list = [Pf_minus]
conv_criterion_list = [conv_criterion]
```

Enrichment

```
learning_function_type = 'U'  # in this case, negative U is used such
that the highest value is selected

for iterations in range(N_pop-N_initial_DoE):

    # kill and restart Plaxis every 50 iterations to refresh memory
    if len(X_DoE)%50 == 0:
        time.sleep(10)
        kill_Plaxis_2D_Output()
        time.sleep(20)

    LFV = learning_function_MLE(mean_pop, var_pop,
learning_function_type)

    reverse_sorted_indices = np.argsort(LFV)[::-1]

    # iterate through LFV values in descending order and select point
with highest LFV that is not yet in DoE
    for i in range(N_pop):
        index_max = reverse_sorted_indices[i]
        if X_pop[index_max] in X_DoE:
            continue
        else:
            x_new = X_pop[index_max]
            y_new = true_function(x_new, g_i)
            break

    X_DoE = np.vstack((X_DoE, np.atleast_2d(x_new)))
    Y_DoE = np.append(Y_DoE, y_new)

    with pm.Model() as GP_model:
        # Hyperparameters for the Gaussian Process
        ls_phi = pm.Uniform("ls_phi", lower=np.min(phi_pop)/100,
upper=np.max(phi_pop)*100)
        ls_c = pm.Uniform("ls_c", lower=np.min(c_pop)/100,
upper=np.max(c_pop)*100)
        ls_E = pm.Uniform("ls_E", lower=np.min(E_pop)/100,
upper=np.max(E_pop)*100)
        ls_M_MCC = pm.Uniform("ls_M_MCC", lower=np.min(M_pop)/100,
upper=np.max(M_pop)*100)
        ls_lambda_MCC = pm.Uniform("ls_lambda_MCC",
lower=np.min(lambda_pop)/100, upper=np.max(lambda_pop)*100)
        ls_kappa_MCC = pm.Uniform("ls_kappa_MCC",
lower=np.min(kappa_pop)/100, upper=np.max(kappa_pop)*100)
```

```
        ls_k_clay = pm.Uniform("ls_k_clay",
lower=np.min(k_clay_pop)/100, upper=np.max(k_clay_pop)*100)
        ls_k_sand = pm.Uniform("ls_k_sand",
lower=np.min(k_sand_pop)/100, upper=np.max(k_sand_pop)*100)
        ls_OCR = pm.Uniform("ls_OCR", lower=np.min(OCR_pop)/100,
upper=np.max(OCR_pop)*100)

        cov_scale = pm.Uniform("cov_scale", lower=0.000001, upper=10)
        sigma = pm.Uniform("sigma", lower=0.000001, upper=10)

        # Mean functiom
        mean_func = pm.gp.mean.Zero()

        # Covariance function
        cov_func = cov_scale ** 2 * pm.gp.cov.Matern52(input_dim=9,
ls=[ls_phi, ls_c, ls_E, ls_M_MCC, ls_lambda_MCC, ls_kappa_MCC,
ls_k_clay, ls_k_sand, ls_OCR])

        # GP prior with zero mean
        gp = pm.gp.Marginal(mean_func = mean_func, cov_func =
cov_func)

        # GP likelihood
        y_ = gp.marginal_likelihood("y_", X_DoE, Y_DoE, sigma = sigma)

        # obtain MAP estimate and calculate predictive mean and
variance using MAP estimate
        start = {'ls_phi': np.mean(phi_pop), 'ls_c': np.mean(c_pop),
'ls_E': np.mean(E_pop),
                 'ls_M_MCC': np.mean(M_pop), 'ls_lambda_MCC':
np.mean(lambda_pop), 'ls_kappa_MCC': np.mean(kappa_pop),
                 'ls_k_clay': np.mean(k_clay_pop), 'ls_k_sand':
np.mean(k_sand_pop), 'ls_OCR': np.mean(OCR_pop),
                 'cov_scale': 1, 'sigma': 1}

        map_estimate = pm.find_MAP(start = start)
        mean_pop, var_pop = gp.predict(X_pop, map_estimate, diag=True)

    Pf, Pf_plus, Pf_minus, conv_criterion = calculate_Pf(mean_pop,
var_pop)

    hyperparams_list.append(map_estimate)
    Pf_list.append(Pf)
    Pf_plus_list.append(Pf_plus)
    Pf_minus_list.append(Pf_minus)

    # Check if training should stop
    if conv_criterion <= 0.05 and iterations >= 100:
        break
```

Saving of Results

```
# Assuming all the mentioned variables are already defined
results_dict = {
    'X_pop': X_pop,
    'initial_X_DoE': initial_X_DoE,
    'X_DoE': X_DoE,
    'Y_DoE': Y_DoE,
    'hyperparams_list': hyperparams_list,
    'Pf_list': Pf_list,
    'Pf_plus_list': Pf_plus_list,
    'Pf_minus_list': Pf_minus_list,
    'conv_criterion_list': conv_criterion_lis
}

# Specify the file path and name for your pickle file
file_path = 'piledembankment_MLE_data.pkl'

# Writing the dictionary to a pickle file
with open(file_path, 'wb') as file:
    pickle.dump(results_dict, file)
```