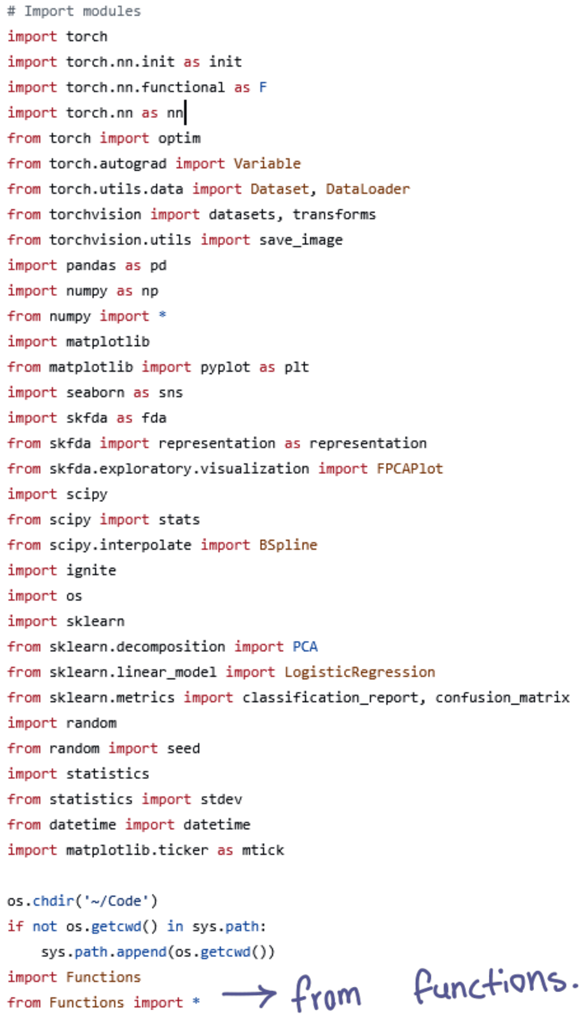
Functional Autoencoder (FAE) Python Codes: https://github.com/CedricBeaulac/FAE/tree/main

The folder Code consists of

- AE_irregular.py: code for implementing the conventional autoencoders (AE) with irregularly spaced functional data.
- AE_regular.py: code for implementing the conventional autoencoders (AE) with regularly spaced functional data.
- DataGenerator_NN.py: the data generator for simulation data sets.
- FAE_irregular.py: code for implementing the proposed functional autoencoders (FAE) with irregularly spaced functional data.
- FAE_regular.py: code for implementing the proposed functional autoencoders (FAE) with regularly spaced functional data.
- FPCA.py: code for implementing the functional principal component analysis (FPCA) with regularly spaced functional data.
- Functions.py: the self-defined functions used for running the existing and proposed methods implemented.
- Plotting.py: code for creating the plots displayed in the manuscript.
- Read_ElNino_Data.py: code for importing and pre-processing the El Nino data set in the manuscript.
- Read_Sim_Data.py: code for importing and pre-processing the simulation data sets in the manuscript.

FAE_regular:

Importing libraries:



```
# Import modules
import torch
import torch.nn.init as init
import torch.nn.functional as F
import torch.nn as nn
from torch import optim
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets, transforms
from torchvision.utils import save_image
import pandas as pd
import numpy as np
from numpy import *
import matplotlib
from matplotlib import pyplot as plt
import seaborn as sns
import skfda as fda
from skfda import representation as representation
from skfda.exploratory.visualization import FPCAPlot
import scipy
from scipy import stats
from scipy.interpolate import BSpline
import ignite
import os
import sklearn
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import random
from random import seed
import statistics
from statistics import stdev
from datetime import datetime
import matplotlib.ticker as mtick

os.chdir('~/Code')
if not os.getcwd() in sys.path:
    sys.path.append(os.getcwd())
import Functions
from Functions import *
```

→ from functions.py

The model: a simple Functional Autoencoder (class)

FAE Class

```
####################################################################
# Define the vanilla FAE architecture for regularly functional data
# Create FAE Class
####################################################################
# Below are the settings for the creating the linear FAE (adjust the architecure for nonlinear FAE)
class FAE_vanilla(nn.Module):
    def __init__(self, weight_std=None):
        super(FAE_vanilla, self).__init__()       } A basic autoencoder module
        ## Select one of the following options (comment out the other one) → two options: weight-std ↗ None  ↘ Not None
        # Opt 1: Linear FAE with 1 hidden layer
        self.fc1 = nn.Linear(n_basis_project, n_rep, bias=False) → Encoder: feature → reps
        self.fc3 = nn.Linear(n_rep, n_basis_revert, bias=False) → Decoder : reps → coeffs ( Smooth curves)
        self.activation = nn.Identity() → Linear Model

        # Opt 2: Nonlinear FAE with 3 hidden layers
        # self.fc1 = nn.Linear(n_basis_project, 100, bias=False)
        # self.fc2 = nn.Linear(100, n_rep, bias=False)
        # self.fc3 = nn.Linear(n_rep, 100, bias=False)
        # self.fc4 = nn.Linear(100, n_basis_revert, bias=False)
        # self.activation = nn.Sigmoid() → Non Linear

        # initialize the weights to a specified, constant value
        if (weight_std is not None):
            for m in self.modules():
                if isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, mean=0.0, std=weight_std) → Linear layer weights: N(0, sd)
                    #nn.init.constant_(m.bias, 0)
```

Forward Pass

```
def forward(self, x, tpts, basis_fc_project, basis_fc_revert):
    feature = self.Project(x, tpts, basis_fc_project) → Project curves onto input basis (Algo 1 - step 1)

    ## Select one of the following options accordingly (comment out the other one)
    # Opt 1: Linear FAE with 1 hidden layer
    rep = self.activation(self.fc1(feature)) → Encod to low-dimensional rep (IR^d)
    coef = self.fc3(rep) → Decode to output-basis coeffs (Algo 1 - Step 2)

    # Opt 2: Nonlinear inear FAE with 3 hidden layers
    # t1 = self.activation(self.fc1(feature))
    # rep = self.fc2(t1)
    # t2 = self.activation(self.fc3(rep))
    # coef = self.fc4(t2)

    x_hat = self.Revert(coef, basis_fc_revert) → Re-build curves from basis (Algo 1- step 3)
    return x_hat, rep, feature, coef
```

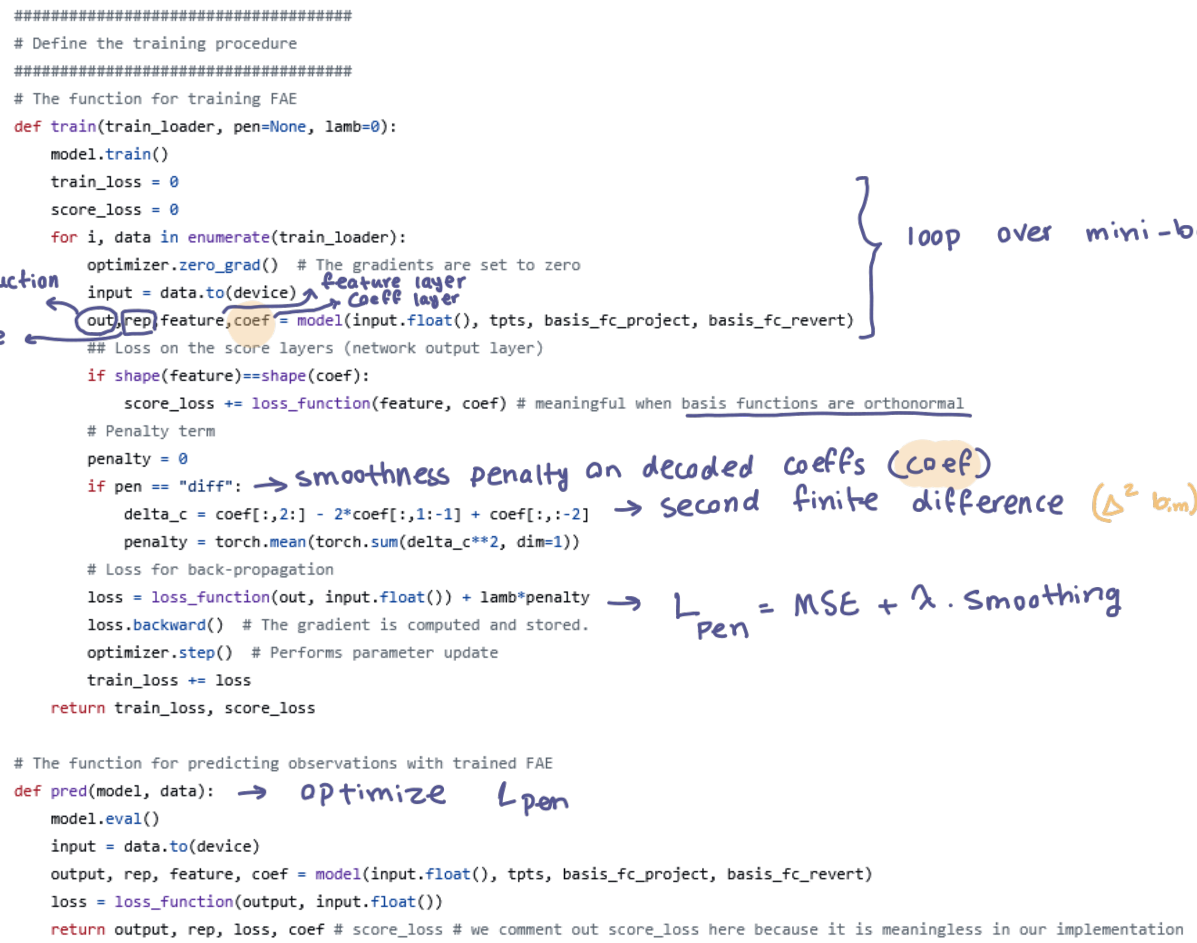
Projection (Numerical Integration → Features)

```
def Project(self, x, tpts, basis_fc): ——→ Trapezodial rule Weights (W) from time grid (tpts)
     basis_fc: n_time X nbasis
     x: n_subject X n_time
    time_diff = tpts[1:] - tpts[:-1] → Δt_K = t_K - t_{K-1}
    W = (torch.cat((torch.tensor([[0]]), time_diff))*1/2 + torch.cat((time_diff, torch.tensor([[0]])))*1/2).flatten().float() ↑ trapezodial rule weight
    f = torch.matmul(torch.mul(x, W), torch.t(basis_fc)) → ∫ x_i(t) Φ_m(t) dt ≈ (x ⊙ W) x basis^T (Algo1-Step 1 - f_m)
    return f

def Revert(self, x, basis_fc): → Re-builds curves as linear Combination of output basis
    g = torch.matmul(x, basis_fc) Coeffs x basis (Algo 1 - step 3 - x̂(t_j))
    return g
```

tpts = torch.linespace(0, 1, step= n-time) → 1D PyTorch tensor

Training & prediction helpers:



```
####################################
# Define the training procedure
####################################
# The function for training FAE
def train(train_loader, pen=None, lamb=0):
    model.train()
    train_loss = 0
    score_loss = 0
    for i, data in enumerate(train_loader):
        optimizer.zero_grad()  # The gradients are set to zero
        input = data.to(device)
        out,rep,feature,coef = model(input.float(), tpts, basis_fc_project, basis_fc_revert)
        ## Loss on the score layers (network output layer)
        if shape(feature)==shape(coef):
            score_loss += loss_function(feature, coef) # meaningful when basis functions are orthonormal
        # Penalty term
        penalty = 0
        if pen == "diff":
            delta_c = coef[:,2:] - 2*coef[:,1:-1] + coef[:,:-2]
            penalty = torch.mean(torch.sum(delta_c**2, dim=1))
        # Loss for back-propagation
        loss = loss_function(out, input.float()) + lamb*penalty
        loss.backward()  # The gradient is computed and stored.
        optimizer.step()  # Performs parameter update
        train_loss += loss
    return train_loss, score_loss

# The function for predicting observations with trained FAE
def pred(model, data):
    model.eval()
    input = data.to(device)
    output, rep, feature, coef = model(input.float(), tpts, basis_fc_project, basis_fc_revert)
    loss = loss_function(output, input.float())
    return output, rep, loss, coef # score_loss # we comment out score_loss here because it is meaningless in our implementation
```

loop over mini-batches

Reconstruction

code

feature layer

Coeff layer

→ smoothness penalty on decoded coeffs (coef)

→ second finite difference $(\Delta^2 b_{i,m})^2$ in L_{pen}

→ $L_{pen} = MSE + \lambda \cdot$ smoothing

→ optimize L_{pen}

Experiment setup

```
#####################################
# Perform FAE (Model Training)
#####################################
# Below are the settings for the implementation with ElNino data described in the real application section
niter = 20
seed(743)
niter_seed = random.sample(range(5000), niter)
# niter = 10
# seed(743)
# niter_seed = random.sample(range(1000), niter)

# Set up basis functions for input functional weights
n_basis_project = 20
basis_type_project = "Bspline"
# Get basis functions evaluated
if basis_type_project == "Bspline":
    bss_project = representation.basis.BSpline(n_basis=n_basis_project, order=4)
elif basis_type_project == "Fourier":
    bss_project = representation.basis.Fourier([min(tpts.numpy().flatten()), max(tpts.numpy().flatten())], n_basis=n_basis_project)
bss_eval_project = bss_project.evaluate(tpts, derivative=0)
basis_fc_project = torch.from_numpy(bss_eval_project[:, :, 0]).float()
```

Build basis functions (Projection & Reconstruction)

↳ Projection Basis matric (functional (basis) coeffs)

```
# Set up basis functions for output functional weights
n_basis_revert = 20
basis_type_revert = "Bspline"
# Get basis functions evaluated
if basis_type_revert == "Bspline":
    bss_revert = representation.basis.BSpline(n_basis=n_basis_revert, order=4)
elif basis_type_revert == "Fourier":
    bss_revert = representation.basis.Fourier([min(tpts.numpy().flatten()), max(tpts.numpy().flatten())], n_basis=n_basis_revert)
bss_eval_revert = bss_revert.evaluate(tpts, derivative=0)
basis_fc_revert = torch.from_numpy(bss_eval_revert[:, :, 0]).float()
```

Build basis functions for output basis used by decoder

```
# Set up lists to save training info
FAE_train_no_niter = []
FAE_reps_train_niter = []
FAE_reps_test_niter = []
FAE_reps_all_niter = []
FAE_pred_test_niter = []
FAE_pred_all_niter = []
FAE_coef_train_niter = []
FAE_coef_test_niter = []
FAE_pred_train_acc_mean_niter = []
FAE_pred_test_acc_mean_niter = []
FAE_pred_train_acc_sd_niter = []
FAE_pred_test_acc_sd_niter = []
classification_FAE_train_niter = []
classification_FAE_test_niter = []
```

Containers to record results

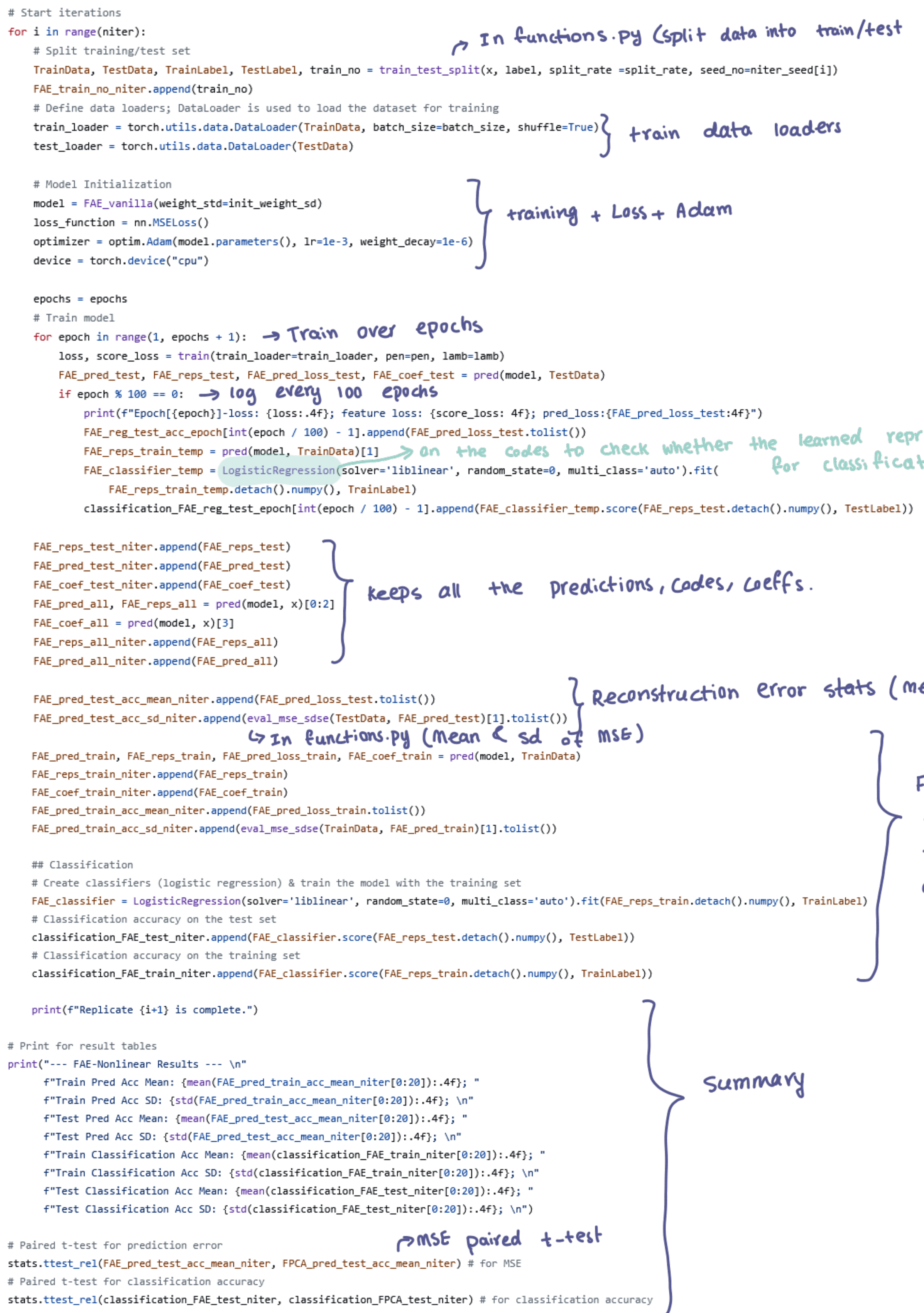
```
# Set up FAE's hyperparameters
n_rep = 5 # number of representation
lamb = 0.001 # penalty parameter
pen = "diff" # penalty type
epochs = 5000 # epochs
batch_size = 28 # batch size
init_weight_sd = 0.5 # SD of normal dist. for initializing NN weight
split_rate = 0.8 # percentage of training set
```

Hyper Parameters

```
# Set up lists for training history
FAE_reg_test_acc_epoch = [[] for x in range(int(epochs/100))]
classification_FAE_reg_test_epoch = [[] for x in range(int(epochs/100))]
```

logging test reconstruction error & classification accuracy every 100 epochs!

Main training loop



In functions.py (split data into train/test by class balance)

```
# Start iterations
for i in range(niter):
    # Split training/test set
    TrainData, TestData, TrainLabel, TestLabel, train_no = train_test_split(x, label, split_rate =split_rate, seed_no=niter_seed[i])
    FAE_train_no_niter.append(train_no)
    # Define data loaders; DataLoader is used to load the dataset for training
    train_loader = torch.utils.data.DataLoader(TrainData, batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(TestData)
```

train data loaders

```
    # Model Initialization
    model = FAE_vanilla(weight_std=init_weight_sd)
    loss_function = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-6)
    device = torch.device("cpu")
```

training + Loss + Adam

```
    epochs = epochs
    # Train model
    for epoch in range(1, epochs + 1):
        loss, score_loss = train(train_loader=train_loader, pen=pen, lamb=lamb)
        FAE_pred_test, FAE_reps_test, FAE_pred_loss_test, FAE_coef_test = pred(model, TestData)
        if epoch % 100 == 0:
            print(f"Epoch[{epoch}]-loss: {loss:.4f}; feature loss: {score_loss: 4f}; pred_loss:{FAE_pred_loss_test:4f}")
            FAE_reg_test_acc_epoch[int(epoch / 100) - 1].append(FAE_pred_loss_test.tolist())
            FAE_reps_train_temp = pred(model, TrainData)[1]
            FAE_classifier_temp = LogisticRegression(solver='liblinear', random_state=0, multi_class='auto').fit(
                FAE_reps_train_temp.detach().numpy(), TrainLabel)
            classification_FAE_reg_test_epoch[int(epoch / 100) - 1].append(FAE_classifier_temp.score(FAE_reps_test.detach().numpy(), TestLabel))
```

→ Train over epochs

→ log every 100 epochs

on the codes to check whether the learned representation is useful for classification (post-hoc probe)

```
    FAE_reps_test_niter.append(FAE_reps_test)
    FAE_pred_test_niter.append(FAE_pred_test)
    FAE_coef_test_niter.append(FAE_coef_test)
    FAE_pred_all, FAE_reps_all = pred(model, x)[0:2]
    FAE_coef_all = pred(model, x)[3]
    FAE_reps_all_niter.append(FAE_reps_all)
    FAE_pred_all_niter.append(FAE_pred_all)
```

keeps all the predictions, codes, coeffs.

```
    FAE_pred_test_acc_mean_niter.append(FAE_pred_loss_test.tolist())
    FAE_pred_test_acc_sd_niter.append(eval_mse_sdse(TestData, FAE_pred_test)[1].tolist())
```

Reconstruction error stats (mean, sd)

↳ In functions.py (mean & sd of MSE)

```
    FAE_pred_train, FAE_reps_train, FAE_pred_loss_train, FAE_coef_train = pred(model, TrainData)
    FAE_reps_train_niter.append(FAE_reps_train)
    FAE_coef_train_niter.append(FAE_coef_train)
    FAE_pred_train_acc_mean_niter.append(FAE_pred_loss_train.tolist())
    FAE_pred_train_acc_sd_niter.append(eval_mse_sdse(TrainData, FAE_pred_train)[1].tolist())

    ## Classification
    # Create classifiers (logistic regression) & train the model with the training set
    FAE_classifier = LogisticRegression(solver='liblinear', random_state=0, multi_class='auto').fit(FAE_reps_train.detach().numpy(), TrainLabel)
    # Classification accuracy on the test set
    classification_FAE_test_niter.append(FAE_classifier.score(FAE_reps_test.detach().numpy(), TestLabel))
    # Classification accuracy on the training set
    classification_FAE_train_niter.append(FAE_classifier.score(FAE_reps_train.detach().numpy(), TrainLabel))
```

Final logistic Regression fit on train Codes, record train/test accuracy

```
    print(f"Replicate {i+1} is complete.")

# Print for result tables
print("--- FAE-Nonlinear Results --- \n"
      f"Train Pred Acc Mean: {mean(FAE_pred_train_acc_mean_niter[0:20]):.4f}; "
      f"Train Pred Acc SD: {std(FAE_pred_train_acc_mean_niter[0:20]):.4f}; \n"
      f"Test Pred Acc Mean: {mean(FAE_pred_test_acc_mean_niter[0:20]):.4f}; "
      f"Test Pred Acc SD: {std(FAE_pred_test_acc_mean_niter[0:20]):.4f}; \n"
      f"Train Classification Acc Mean: {mean(classification_FAE_train_niter[0:20]):.4f}; "
      f"Train Classification Acc SD: {std(classification_FAE_train_niter[0:20]):.4f}; \n"
      f"Test Classification Acc Mean: {mean(classification_FAE_test_niter[0:20]):.4f}; "
      f"Test Classification Acc SD: {std(classification_FAE_test_niter[0:20]):.4f}; \n")

# Paired t-test for prediction error
stats.ttest_rel(FAE_pred_test_acc_mean_niter, FPCA_pred_test_acc_mean_niter) # for MSE
# Paired t-test for classification accuracy
stats.ttest_rel(classification_FAE_test_niter, classification_FPCA_test_niter) # for classification accuracy
```

summary

MSE paired t-test

↳ classification paired t-test