```
from sklearn.svm import SVC
from tqdm import tqdm
import gpflow
from gpflow import kernels
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm

np.random.seed(42)
gpflow.settings.numerics.quadrature = 'error'
import torch, torchvision
import torchvision.transforms as transforms
root = './data'



def get_idxs(max_range = 100):
    num = int(0.8 * max_range)
    trn_idxs = np.random.choice(range(max_range), num, replace=False)
    test_idxs = np.array([i for i in range(max_range) if i not in trn_idxs])
    return trn_idxs, test_idxs


def classifier_on_data(data, labels, trn_idxs, test_idxs):
    train_data = data[trn_idxs, :]
    train_labels = labels[trn_idxs]

    test_data = data[test_idxs, :]
    test_labels = labels[test_idxs]

    svc = SVC(gamma = "auto")
    svc.fit(train_data, train_labels)
    preds = svc.predict(test_data)

    return len(np.where(preds == test_labels)[0]) / len(preds)


class DeepGPLVM:
    def __init__(self,kernel_dims = [5, 5], n_layers=2, inducing_pts=[20, 20], latent_dims=[100,2
0], max_iters=10):
        self.kernel_dims = kernel_dims
        self.latent_dims = latent_dims
        self.inducing_pts = inducing_pts
        self.n_layers = n_layers
        self.max_iters = max_iters

    def train(self, Y):
        self.models = []
        N = Y.shape[0]

        for i in range(self.n_layers):
            M = self.inducing_pts[i]
            kernel = kernels.RBF(self.kernel_dims[i], ARD=True, active_dims=slice(0,self.kernel_d
ims[i]))
            Q = self.latent_dims[i]
            X_mean = gpflow.models.PCA_reduce(Y, Q)
            Z = np.random.permutation(X_mean.copy())[:M]


            m1 = gpflow.models.BayesianGPLVM(X_mean=X_mean, X_var=0.1*np.ones((N, Q)), Y=Y,
                                    kern=kernel, M=M, Z=Z)
            m1.likelihood.variance = 0.01
            m1.compile()
            opt = gpflow.train.ScipyOptimizer()
            opt.minimize(m1, maxiter=gpflow.test_util.notebook_niter(self.max_iters))
            self.models.append(m1)
            Y = m1.X_mean.read_value()

            self.means = Y.copy()
    def reconstruct(self, idx):
        x_mean = self.means[idx].reshape(1, self.latent_dims[-1])
        for model in reversed(self.models):
            x_recon = model.predict_y(x_mean)[0]
            x_mean = x_recon
        return x_recon

    def reconstruct_from_input(self, test):
        for model in reversed(self.models):
            x_recon = model.predict_y(test)[0]
            test = x_recon
        return x_recon

    def reconstructon_error(self, Y):
        total_err = 0
        for y in Y:
            recon  = self.reconstruct_from_input(y.reshape(1, Y.shape[1]))
            error = np.sqrt(((Y - recon)**2).sum()/Y.shape[1])
            total_err += error

        return total_err / Y.shape[0]

    def get_sensitivities(self, plot = False):
        sensitivities = []
        for model in reversed(self.models):
            kern = model.kern
            sens = np.sqrt(kern.variance.read_value())/kern.lengthscales.read_value()
            print(sens)
            sensitivities.append(sens)
            if plot == True:
                plt.figure()
                plt.bar(dims, sens, 0.2, color='y')
                plt.xticks(dims)
                plt.xlabel('dimension')
                plt.title('Sensitivity to latent inputs');
                plt.bar(np.arange(len(sens)), sens)
        return sensitivities

    def predict_likelihoods(self, X, Y):
        pass
```