```python
import torch
from torch import nn
from torch import optim
import numpy as np
import matplotlib.pyplot as plt
import data


class PTLogreg(nn.Module):
    def __init__(self, D, C, param_lambda=0.0):
        """Arguments:
          - D: dimensions of each datapoint
          - C: number of classes
        """
        super(PTLogreg, self).__init__()
        W = np.random.randn(D, C).astype(np.float64)
        b = np.random.randn(1, C).astype(np.float64)
        self.W = nn.Parameter(data=torch.from_numpy(W), requires_grad=True)
        self.b = nn.Parameter(data=torch.from_numpy(b), requires_grad=True)
        self.param_lambda = param_lambda

    def forward(self, X):
        N = len(X)
        C = self.b.shape[1]
        scores = torch.mm(X, self.W) + self.b
        probs = torch.softmax(scores, 1, dtype=torch.float64)
        return probs

    def get_loss(self, X, Yoh_):
        N = len(X)
        probs = self.forward(X)
        logprobs = torch.log(probs)
        loss = - 1 / N * torch.sum(logprobs * Yoh_) + self.param_lambda * torch.linalg.norm(self.W)
        return loss

    def train(self, X: torch.Tensor, Yoh_: torch.Tensor, param_niter=100, param_delta=0.05, print_frequency=10):
        """
        Arguments:
            X (torch.Tensor): model inputs [NxD]
            Yoh_ (torch.Tensor): ground truth [NxC]
            param_niter (int): number of training iterations
            param_delta (float): learning rate
        """
        optimizer = optim.SGD(self.parameters(), lr=param_delta)

        for i in range(param_niter):
            loss = self.get_loss(X, Yoh_)

            if i % print_frequency == 0:
                print(f"iteration {i} loss {loss} weights norm {torch.linalg.norm(self.W)}")

            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

    def eval(self, X):
        """Evaluate this model.

        Arguments:
            X: actual datapoints [NxD], type: np.array

        Returns:
            predicted class probabilites [NxC], type: np.array
        """
        X = torch.from_numpy(X)
        probs = self.forward(X)
```

```
        return probs.detach().numpy()


if __name__ == "__main__":
    np.random.seed(100)
    X, Y_ = data.sample_gauss_2d(3, 100)
    Yoh_ = data.class_to_onehot(Y_)

    model = PTLogreg(X.shape[1], Yoh_.shape[1], param_lambda=0.1)
    model.train(torch.from_numpy(X), torch.from_numpy(Yoh_), 1000, 0.05, 100)
    probs = model.eval(X)
    # predicted classes
    Y = np.hstack([np.argmax(probs[i][:]) for i in range(probs.shape[0])])
    # reshaping for other methods purposes
    Y_ = np.hstack(Y_)
    accuracy, pr, M = data.eval_perf_multi(Y, Y_)
    print("Accuracy: ", accuracy)
    print("Precision / Recall: ", pr)
    print("Confussion Matrix: ", M)

    def decfun(X):
        X = torch.from_numpy(X)
        to_return = model.forward(X)
        return to_return.detach().numpy().argmax(axis=1)

    bbox = (np.min(X, axis=0), np.max(X, axis=0))
    data.graph_surface(decfun, bbox, offset=0.5)
    # graph the data points
    data.graph_data(X, Y_, Y, special=[])
    # show the plot
    plt.show()
```