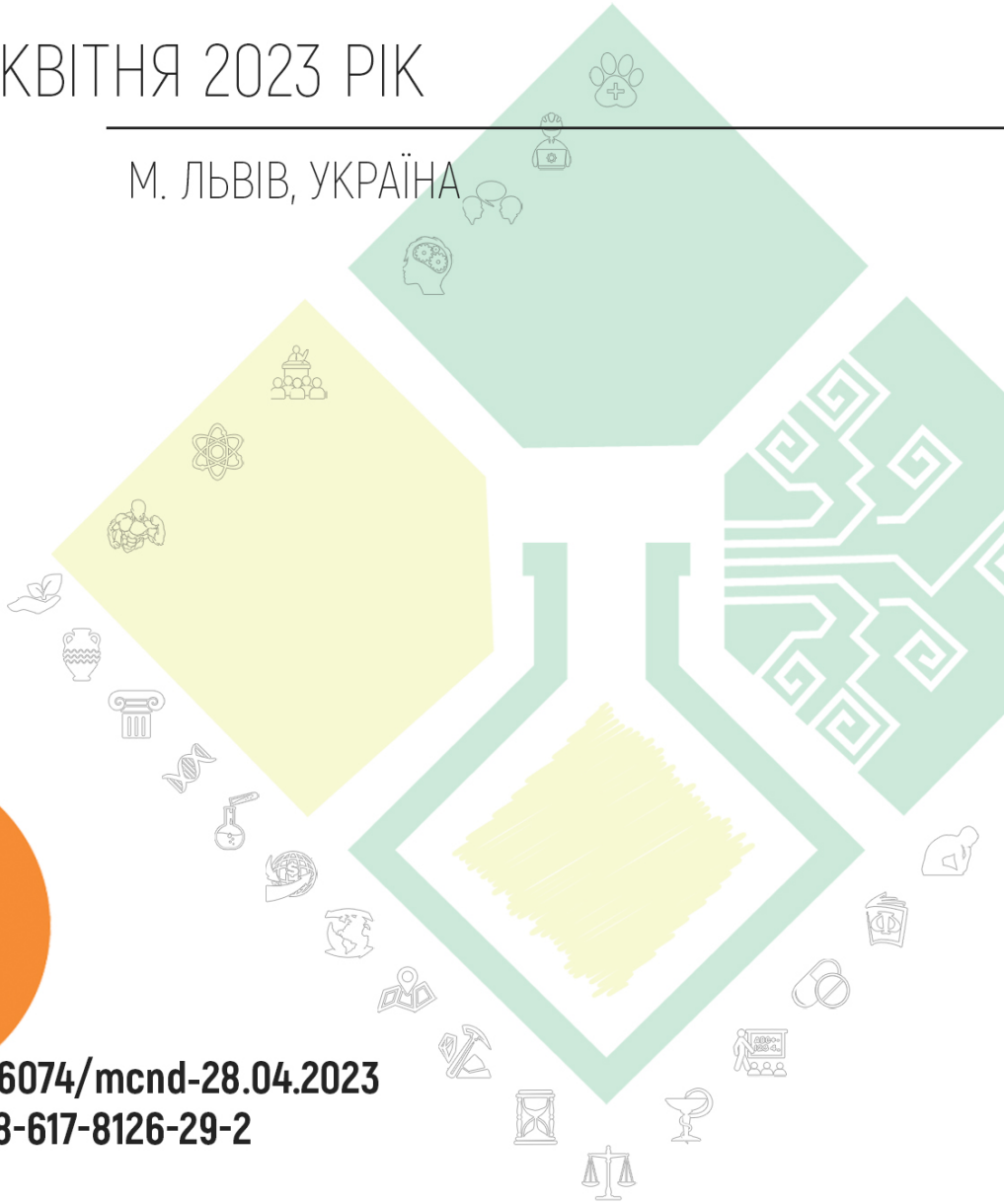
МАТЕРІАЛИ III МІЖНАРОДНОЇ НАУКОВОЇ КОНФЕРЕНЦІЇ

РОЗВИТОК НАУКОВОЇ ДУМКИ ПОСТІНДУСТРІАЛЬНОГО СУСПІЛЬСТВА: СУЧАСНИЙ ДИСКУРС

28 КВІТНЯ 2023 РІК

М. ЛЬВІВ, УКРАЇНА



OPEN ACCESS

DOI 10.36074/mcnd-28.04.2023
ISBN 978-617-8126-29-2

МАТЕРІАЛИ
III МІЖНАРОДНОЇ
НАУКОВОЇ КОНФЕРЕНЦІЇ

Міжнародний Центр Наукових Досліджень

РОЗВИТОК НАУКОВОЇ ДУМКИ ПОСТІНДУСТРІАЛЬНОГО СУСПІЛЬСТВА: СУЧАСНИЙ ДИСКУРС

28 КВІТНЯ 2023 РІК
м. Львів, Україна

Вінниця, Україна
«Європейська наукова платформа»
2023

УДК 001 (08)
Р 64

https://doi.org/10.36074/mcnd-28.04.2023

Організація, від імені якої випущено видання:
ГО «Міжнародний центр наукових досліджень»

Голова оргкомітету: Рабей Н.Р.

Верстка: Зрада С.І.
Дизайн: Бондаренко І.В.

Конференцію зареєстровано Державною науковою установою «УкрІНТЕІ» в базі даних науково-технічних заходів України та бюлетені «План проведення наукових, науково-технічних заходів в Україні» (Посвідчення № 58 від 17.01.2023).

Р 64 **Розвиток наукової думки постіндустріального суспільства: сучасний дискурс**: матеріали III Міжнародної наукової конференції, м. Львів, 28 квітня, 2023 р. / Міжнародний центр наукових досліджень. — Вінниця: Європейська наукова платформа, 2023. — 192 с.

ISBN 978-617-8126-29-2
DOI 10.36074/mcnd-28.04.2023

Викладено матеріали учасників III Міжнародної спеціалізованої наукової конференції «Розвиток наукової думки постіндустріального суспільства: сучасний дискурс», яка відбулася 28 квітня 2023 року у місті Львів.

УДК 001 (08)

ISBN 978-617-8126-29-2

СЕКЦІЯ XI.
ТЕХНОЛОГІЇ ЛЕГКОЇ ТА ДЕРЕВООБРОБНОЇ ПРОМИСЛОВОСТІ

СЕКЦІЯ XII.
ЕКОЛОГІЯ ТА ТЕХНОЛОГІЇ ЗАХИСТУ НАВКОЛИШНЬОГО СЕРЕДОВИЩА

СЕКЦІЯ XIII.
КОМП'ЮТЕРНА ТА ПРОГРАМНА ІНЖЕНЕРІЯ

СЕКЦІЯ XIV.
СИСТЕМНИЙ АНАЛІЗ, МОДЕЛЮВАННЯ ТА ОПТИМІЗАЦІЯ

СЕКЦІЯ XV.
ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ

МОДЕЛЮВАННЯ ПОВТОРЮВАНИХ ПОНЯТЬ У НЕЗБАЛАНСОВАНИХ ПОТОКАХ ДАНИХ

Холодов Станіслав Євгенович
здобувач вищої освіти, Факультет інформаційних
радіотехнологій та технічного захисту інформації
Харківський національний університет радіоелектроніки, Україна

Науковий керівник: Кузьомін Олександр Якович
доктор технічних наук, професор, професор кафедри інформатики,
Засновник Асоціації випускників ХНУРЕ
Харківський національний університет радіоелектроніки, Україна

Анотація. *Моделювання повторюваних понять у незбалансованих потоках даних - це задача машинного навчання, яка полягає в класифікації даних, коли кількість прикладів кожного класу неоднакова. Незбалансованість даних може виникнути, наприклад, коли один з класів дуже рідкісний або коли деякі приклади були відкинуті через помилки даних або неповноту. Для моделювання повторюваних понять у незбалансованих потоках даних можна використовувати різні методи, залежно від конкретної задачі і властивостей даних.*

Опис публікації:

У цьому проекті ми прагнемо змоделювати повторювані концепції в потоках даних, розподіл яких спотворений.

Для цього необхідно виконати наступні підзавдання:

1. Впровадження графічної моделі, яка моделює концепти в незбалансованому потоці даних як стани/вузли;

2. Включення одного методу передискретизації (випадкова передискретизація, локалізована передискретизація, SMOTE, ...), щоб збільшити кількість екземплярів меншості до того, як класифікатор буде призначено одному стану/вузлу, і таким чином усунути дисбаланс класів;

3. Динамічна візуалізація графічної моделі за допомогою Directed Graph.

Робочий процес

Вивчався такий алгоритм, як GraphPool. Багато інформації також було проаналізовано для роботи з цим алгоритмом. Під час роботи з ним виникали деякі труднощі, такі як витік пам'яті та дуже довге навчання моделі, але вони були виправлені шляхом виправлення деяких змінних[1].

Для ознайомлення з таким поняттям, як GraphPool, був реалізований алгоритм Hierarchical Graph Pooling with Structure Learning

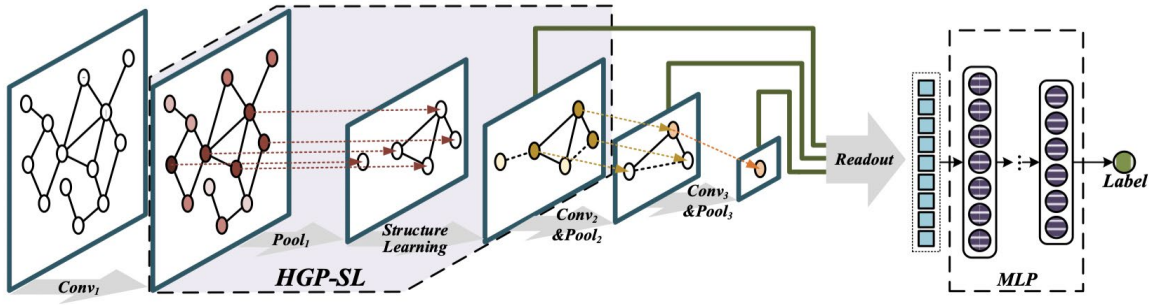


Рис. 1. **Об'єднання графів із вивченням структури**

Це реалізація PyTorch алгоритму GraphPooling, який вивчає низькорозмірне представлення для всього графіка. Зокрема, операція об’єднання графів використовує функції вузла та інформацію про структуру графа для виконання зменшення вибірки на графіках. Потім рівень навчання структури укладається в операцію об’єднання, яка спрямована на вивчення уточненої структури графа, яка може найкраще зберегти важливу топологічну інформацію[2].

Реалізація коду

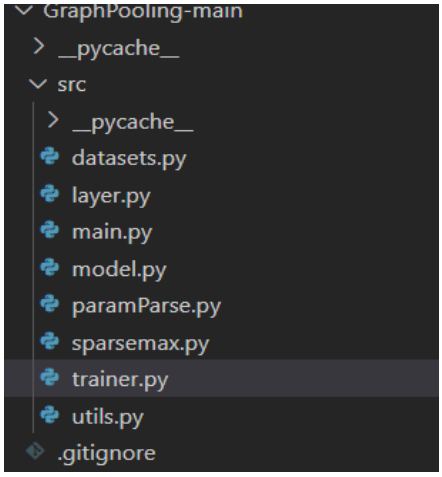


Рис. 2. **Структура проекту**

Тут ми можемо знайти об’єднані дані для моделі роботи та навантажень для моделі навчання.

Код («main.py»)

```
from paramParse import paramterParser
from trainer import Trainer
from utils import setup_seed


if __name__ == '__main__':
    setup_seed(2022)
    args = paramterParser()
    trainer = Trainer(args)
    trainer.fit()
    print("Test acc is {}".format(trainer.eval(validate=False)))
```

Функція навчання моделі

```
import torch
from torch.utils.data import DataLoader
import torch.nn.functional as F

from torch_geometric.datasets import TUDataset
from torch_geometric.transforms import OneHotDegree
from torch_geometric.utils import degree

from model import DiffPool
from datasets import GraphDataset
from paramParse import paramterParser

from tqdm import tqdm, trange

class Trainer():
    def __init__(self, args) -> None:
```

```
        self.args = args
        self.device = torch.device(self.args.gpu_id)
        self.train_data, self.val_data, self.test_data = self.prepareData()

        self.model = DiffPool(args, self.num_x, self.num_y).to(self.device)

    def prepareData(self):
        data = TUDataset("./datasets/{}.".format(self.args.dataset), name =
self.args.dataset, use_node_attr=True)

        if data[0].x == None:  # node don't have attribute, according to degree encode
onehot for each node
            max_degree = 0

            for g in data:
                max_degree = max(
                    max_degree,
                    int(max(degree(g.edge_index[0])))
                )

            one_hot_degree = OneHotDegree(max_degree, cat=False)
            # Add ont-hot feature for each node
            data.transform = one_hot_degree


        self.num_x = data[0].x.shape[1]

        y_num = set()
        max_num_nodes = 0
        for g in data:
            max_num_nodes = max(max_num_nodes, g.x.shape[0])
            y_num.add(int(g.y[0]))
        self.num_y = len(y_num)
        max_num_nodes = self.args.max_num_node
        print("Max node num is {}".format(max_num_nodes))

        data = data.shuffle()
        train_num  = int(len(data) * 0.8)
        test_num   = int(len(data) * (1-0.1))
        train_data = data[:train_num]
        val_data   = data[train_num:test_num]
        test_data  = data[test_num:]

        train_dataset = GraphDataset(train_data, max_num_nodes)
        train_loader  = DataLoader(train_dataset, batch_size = self.args.batch_size)

        val_dataset   = GraphDataset(val_data, max_num_nodes)
        val_loader    = DataLoader(val_dataset, batch_size = self.args.batch_size)

        test_dataset  = GraphDataset(test_data, max_num_nodes)
        test_loader   = DataLoader(test_dataset, batch_size = self.args.batch_size)
        return train_loader, val_loader, test_loader

    def fit(self):
        optimizer = torch.optim.Adam(
            self.model.parameters(),
            lr = self.args.lr,
            weight_decay = self.args.weight_decay)

        self.bets_acc = 0
        self.model.train()
        epochs = trange(self.args.epochs, leave = True, desc = "Epoch")
        for epoch in epochs:
            loss_sum = 0
            num = 0

            for idx, graph in tqdm(enumerate(self.train_data),
total=len(self.train_data),  desc="Batches", leave=False):
                optimizer.zero_grad()
```

```
                x = graph['x'].to(self.device)
                adj = graph['adj'].to(self.device)
                y = graph['y']
                num_nodes = graph['num_nodes'].to(self.device)

                y_pred, loss_ = self.model(x, adj, num_nodes)
                loss = F.cross_entropy(y_pred.cpu(), y.view(-1), reduction='mean') +
loss_
                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 2)
                optimizer.step()
                loss_sum += loss
                num += len(y.view(-1))

            loss = loss_sum/num
            epochs.set_description("Epoch (Loss=%g)" % loss)

            if epoch%10 == 0:
                val_acc = self.eval(validate = True)
                if val_acc > self.bets_acc:
                    self.best_model = self.model
                    print("Now best val acc {}".format(val_acc))


    def eval(self, validate = False):
        correct = 0
        total = 0

        if validate == True:
            data  = self.val_data
            model = self.model
        else:
            data  = self.test_data
            model = self.best_model

        model.eval()

        for idx, graph in tqdm(enumerate(data), total=len(data),  desc="Batches",
leave=False):
            x = graph['x'].to(self.device)
            adj = graph['adj'].to(self.device)
            y = graph['y']

            y_pred, _ = model(x, adj)
            prediction = torch.argmax(y_pred, 1).cpu()
            correct += (prediction == y.view(-1)).sum()
            total += len(y.view(-1))
        return (correct/total).detach().numpy()
```

Реалізація моделі (model.py)

```
import torch

from torch_geometric.nn import DenseGCNConv
from torch_geometric.nn.dense import dense_diff_pool

from utils import construct_mask, my_dense_diff_pool

class DiffPool(torch.nn.Module):
    '''
    DiffPool:Hierarchical Graph Representation Learning with Differentiable Pooling
    '''
    def __init__(self, args, number_of_labels, num_y) -> None:
        super().__init__()
        self.args = args
        self.number_of_labels = number_of_labels
        self.num_y = num_y
        self.device = torch.device(self.args.gpu_id)
```

```
        self.act = torch.nn.ReLU()
        self.setup_layers()
        self.init_weight()

    def init_weight(self):
        '''
        Init model paramaters
        '''
        for name, param in self.fc.named_parameters():
            if 'weight' in name:
                torch.nn.init.xavier_normal_(param)

    def apply_bn(self, x):
        ''' Batch normalization of 3D tensor x
        '''
        bn_module = torch.nn.BatchNorm1d(x.size()[1]).to(self.device)
        return bn_module(x)

    def build_conv_layers(self, input_dim, hidden_dim, output_dim):
        '''
        Construct a three layer.
        '''
        conv_1 = DenseGCNConv(in_channels = input_dim,  out_channels = hidden_dim)
        conv_2 = DenseGCNConv(in_channels = hidden_dim, out_channels = hidden_dim)
        conv_3 = DenseGCNConv(in_channels = hidden_dim, out_channels = output_dim)
        return conv_1, conv_2, conv_3

    def setup_layers(self):
        # Input GCN
        self.GCN_input  = DenseGCNConv(self.number_of_labels, self.args.hidden_dim)
        self.GCN_hidden = DenseGCNConv(self.args.hidden_dim, self.args.hidden_dim)
        self.GCN_output = DenseGCNConv(self.args.hidden_dim, self.args.hidden_dim)

        # Assign GCN
        self.GCN_ass_input  = torch.nn.ModuleList()
        self.GCN_ass_hidden = torch.nn.ModuleList()
        self.GCN_ass_output = torch.nn.ModuleList()
        self.Assign_fc      = torch.nn.ModuleList()
        assign_dim = int(self.args.max_num_node * self.args.assign_ratio)
        assign_dims = []
        assign_input_dim = self.number_of_labels

        for id in range(self.args.num_pooling):
            assign_dims.append(assign_dim)
            assign_conv_1, assign_conv_2, assign_conv_3 =
self.build_conv_layers(assign_input_dim, self.args.hidden_dim, assign_dim)
            assign_linear = torch.nn.Linear( self.args.hidden_dim*2 + assign_dim,
assign_dim)

            self.GCN_ass_input.append(assign_conv_1)
            self.GCN_ass_hidden.append(assign_conv_2)
            self.GCN_ass_output.append(assign_conv_3)
            self.Assign_fc.append(assign_linear)

            # next layer set
            assign_dim = int(assign_dim * self.args.assign_ratio)
            assign_input_dim = self.args.hidden_dim*3

        # Output GCN
        self.GCN_out_input  = torch.nn.ModuleList()
        self.GCN_out_hidden = torch.nn.ModuleList()
        self.GCN_out_output = torch.nn.ModuleList()

        for id in range(self.args.num_pooling):
            conv_1, conv_2, conv_3 = self.build_conv_layers(3*self.args.hidden_dim,
self.args.hidden_dim, self.args.hidden_dim)
            self.GCN_out_input.append(conv_1)
```

```
            self.GCN_out_hidden.append(conv_2)
            self.GCN_out_output.append(conv_3)

        self.fc = torch.nn.Sequential(
            torch.nn.Linear((self.args.hidden_dim*3) * (self.args.num_pooling+1), 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, self.num_y)
        )

    def gcn_forward(self, x, adj, input_layer, hidden_layer, output_layer):
        '''
        return: [batch, node_num, 2 * hidden_dim + output_dim]
        '''
        x_all = []

        x = input_layer(x, adj)
        x = self.act(x)
        if self.args.batch_norm:
            x = self.apply_bn(x)
        x_all.append(x)

        x = hidden_layer(x, adj)
        x = self.act(x)
        if self.args.batch_norm:
            x = self.apply_bn(x)
        x_all.append(x)

        x = output_layer(x, adj)
        x_all.append(x)

        x_tensor = torch.cat(x_all, dim = 2)

        return x_tensor

    def forward(self, feat, adj, node_num = None):
        loss = 0
        ass_feat = feat
        out_all = []

        feat = self.gcn_forward(feat, adj, self.GCN_input, self.GCN_hidden,
self.GCN_output)

        out, _ = torch.max(feat, dim=1)
        out_all.append(out)

        for i in range(self.args.num_pooling):
            if node_num != None and i == 0:
                embedding_mask = construct_mask(self.args.max_num_node,
node_num).to(self.device)
            else:
                embedding_mask = None


            ass_feat = self.gcn_forward(ass_feat, adj, self.GCN_ass_input[i],
self.GCN_ass_hidden[i], self.GCN_ass_output[i])
            ass_feat = self.Assign_fc[i](ass_feat)

            if embedding_mask != None:
                ass_feat = ass_feat * embedding_mask

            feat, adj, loss_lp, loss_e = my_dense_diff_pool(feat, adj, ass_feat)
            loss += loss_lp + loss_e

            feat = self.gcn_forward(feat, adj, self.GCN_out_input[0],
self.GCN_out_hidden[0], self.GCN_out_output[0])
            ass_feat = feat
```

```
            out, _ = torch.max(feat, dim=1)
            out_all.append(out)

        feat = torch.cat(out_all, dim=1)
        score = self.fc(feat)

        return score, loss
```

Доданий Imbalance-Learn to make data set disbalance

```
from imblearn.datasets import make_imbalance

class GraphDataset(data.Dataset):
    def __init__(self, data, max_num_nodes = None) -> None:
        super().__init__()
        self.adj_list        = []
        self.x_list          = []
        self.y_list          = []
        self.edge_index_list = []
        self.max_num_nodes = max_num_nodes
        self.prepareData(data, max_num_nodes)

    def prepareData(self, data, max_num_nodes = None):
        for g in data:
            f = torch.zeros((self.max_num_nodes, g.x.shape[1]))
            f[:g.x.shape[0], :g.x.shape[1]] = g.x
            self.x_list.append(f)
            self.y_list.append(g.y)
            self.edge_index_list.append(g.edge_index)
            adj = to_dense_adj(g.edge_index)
            self.adj_list.append(adj[0])
```

Рис. 3. **Результат роботи**

Перший алгоритм GraphPool є базовим. По-друге, після додавання незбалансованого навчання. Сині лінії – це штучні набори даних, а сірі лінії – справжні набори даних. Ми бачимо, що в обох варіантах реальні набори даних мають кращу точність, ніж штучні набори даних[3].

Висновок

Цей короткий звіт містить модель для можливого використання для вирішення поставленої задачі. Тут була робота над тим, щоб зробити роботу GraphPool зручнішою та зробити її сумісною з потоками даних. Додано незбалансоване навчання для створення незбалансованості наборів даних і перевірено, як це працює з різними потоками даних. Ми помітили, що дані, які проходять через збалансований потік, працювали чіткіше та виглядали комфортніше. Найпоширеніші методи включають перевищення, вагову функцію, вибір проб і використання алгоритмів навчання, які добре справляються з незбалансованими даними. Кожен метод має свої переваги і недоліки, тому вибір конкретного методу залежить від властивостей даних і мети моделювання.

Список використаних джерел:

1. Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. Journal of artificial intelligence research, 16, 321-357.
2. He, H., Bai, Y., Garcia, E. A., & Li, S. (2008). ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on (pp. 1322-1328). IEEE.
3. Kubat, M., & Matwin, S. (1997). Addressing the curse of imbalanced training sets: One-sided selection. In ICML (Vol. 97, pp. 179-186).