REPETITION MODELING TO UNDERSTAND UNBALANCED DATA STREAMS

Kuzomin Oleksandr, Kholodov Stanislav

***Abstract**: Modeling recurring concepts in unbalanced data streams is a machine learning task that involves classifying data when the number of examples of each class is unequal*

***Keywords**: GraphPool, Model;*

***DOI**: https://doi.org/10.54521/ijita30-02-p05*

Introduction

Modeling recurring concepts in unbalanced data streams is a machine learning task that involves classifying data when the number of examples of each class is unequal. Data imbalances can occur, for example, when one of the classes is very rare, or when some examples were dropped due to data errors or incompleteness.

Different methods can be used to model repetitive concepts in unbalanced data streams, depending on the specific task and data properties.

Post description

In this project, we aim to model recurrent concepts in data streams whose distribution is skewed. To do this, you need to complete the following subtasks:

1. Implementation of a graphical model that models concepts in an unbalanced data stream as states/nodes;
2. Inclusion of one oversampling method (random oversampling, localized oversampling, SMOTE, ...) to increase the number of minority instances before a classifier is assigned to a single state/node, and thus eliminate class imbalance;
3. Dynamic visualization of the graphic model using Directed Graph.

Work process

An algorithm such as GraphPool was studied. A lot of information was also analyzed to work with this algorithm. There were some difficulties while working with it, such as memory leaks and very long model training, but these were fixed by fixing some variables[1].

To get acquainted with such a concept as GraphPool, the Hierarchical Graph Pooling with Structure Learning algorithm was implemented

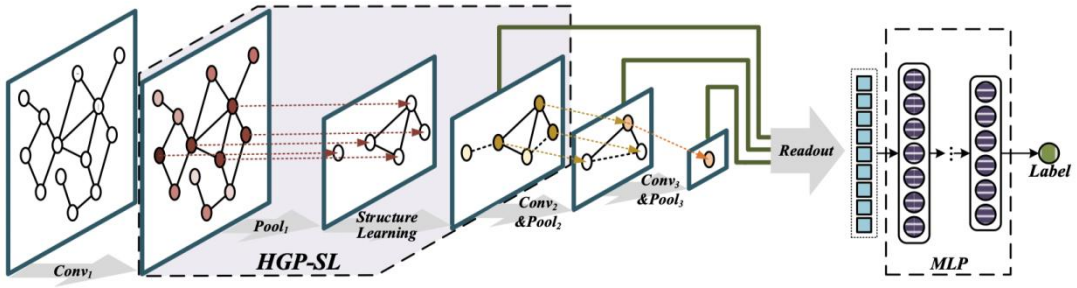


Fig. 1. Combining graphs with structure study.

This is a PyTorch implementation of the GraphPooling algorithm, which learns a low-dimensional representation for the entire graph. In particular, the graph join operation uses node functions and graph structure information to perform downsampling on graphs. Then, the structure learning layer is put into a union operation, which aims to learn the refined graph structure that can best preserve the important topological information [2].

Implementation of the code:

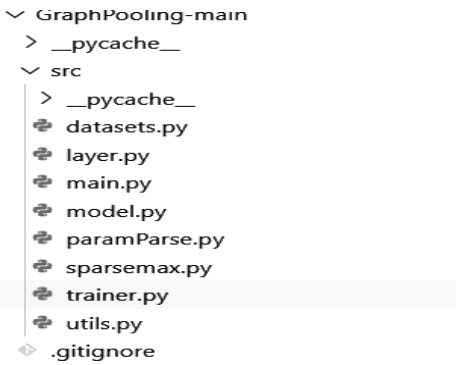


Fig. 2. Project structure

Here we can find the combined data for the work model and the loads for the learning model.

Code ("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)))
```

In this part of the code you can see the most important main function, in which the parameters are parsed and the model training function is called, and the result of the work is displayed.

Model training function

```
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         (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), nam
= 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 =
```

This part includes additional libraries.

In this part of the code there is a declaration and the main implementation of the class responsible for training the model. There is also a parameter responsible for preparing data for further training.

```
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
```

In this part of the code you can observe the main mathematical calculations of model training. This part of the function is the main one

```
            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()
```

At one time, the final calculation and return from the function were implemented here

Implementation of the model (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         (self, args, number_of_labels, num_y) -> None:
        super().        ()
        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):
        '''
```

In this part of the code, you can see the implementation of the class for the data pool

```
        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)
```

Using the DesnseGCNConv plug-in library, input GCN occurs, after which assign GCN occurs and is transmitted to output.

```
        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
```

In this part of the code, previously processed data is converted into layers, which are the main ones for using the graph pool algorithm.

Added Imbalance-Learn to make data set imbalance

```
from imblearn.datasets import make_imbalance

class GraphDataset(data.Dataset):
    def          (self, data, max_num_nodes = None) -> None:
        super().         ()
        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])
```

And in the end, the Graph pool algorithm is used with the prepared data, the result of which can be observed in the following graph for balanced and unbalanced data.

Fig. 3 The result of the work

The first GraphPool algorithm is basic. Second, after adding unbalanced training. The blue lines are the artificial datasets and the gray lines are the real datasets. We can see that in both cases, real datasets have better accuracy than artificial datasets [3].

Conclusion

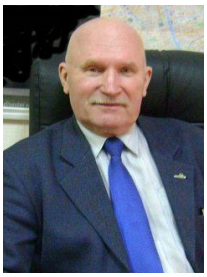
This short report provides a model for possible use to solve the problem at hand. There was work here to make GraphPool more user-friendly and compatible with data streams. Added unbalanced training to create imbalanced datasets and tested how it works with different data streams. We noticed that data passing through the balanced stream ran more clearly and looked more comfortable. The most common methods include overfitting, weight function, sample selection, and using learning algorithms that deal well with unbalanced data. Each method has its advantages and disadvantages, so the choice of a particular method depends on the properties of the data and the purpose of the simulation.

Bibliography

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 unbalanced 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 unbalanced training sets: One-sided selection. In ICML (Vol. 97, pp. 179-186).

Authors' Information



Kuzomin Oleksandr - professor, Department of Informatics of KhNURE, doctor of technical sciences, head of the PNDL "Information transfer technologies for risk reduction systems", project coordinator "Ostpartnerschaft", ERASMUS, DAAD with the University of Hanover, Germany.
oleksandr.kuzomin@nure.ua

Kholodov Stanislav - Master of Kharkiv

National University of Radioelectronics. Specialty "Architectural Programming of Information Systems".

I take part in various activities. Recently, at the end of 2022, I participated online in the implementation of deep learning projects in the DAAD program of Leibniz University Hannover. I have 3 years of commercial programming experience in C++

stanislav.kholodov@nure.ua