Deep Learning and Data Science (CSE5851) **Assignment6**

Department of Statistics and Data Science **2020321803 Song minsoo**

April 8, 2021

1. Find embedding vectors of vertices in a given network based on a random walk aided approach, the so called DeepWalk . Use one of the files karate_club.adjlist " or "karate_club.edgelist", which correspond to the adjacency list and edge list, respectively, s o that you work on the dataset for the Zachary' s karate club network. Adopt the stochastic gradient descent (SGD) optimizer and hyperparameters set to the following values:

- dimension of each embedding vector (d) : 2
- learning rate (η) : 0.02
- walk length (t) : 10
- window size (w) : 3
- walks per vertex (γ) : 5

which can however be replaced by other ones if another setting leads to a better result. You may set other hyperparameters arbitrarily. Use Hierarchical Softmax built upon a binary tree to approximate the probability distribution. To show the convergence , plot the loss versus the number of epochs using the above dataset. Additionally, plot all resulting vectors on the two dimension al space. Make discussions on how the vertices are embedded in comparison with the result based on matrix factorization but also DeepWalk with no approximation .

- loss plot

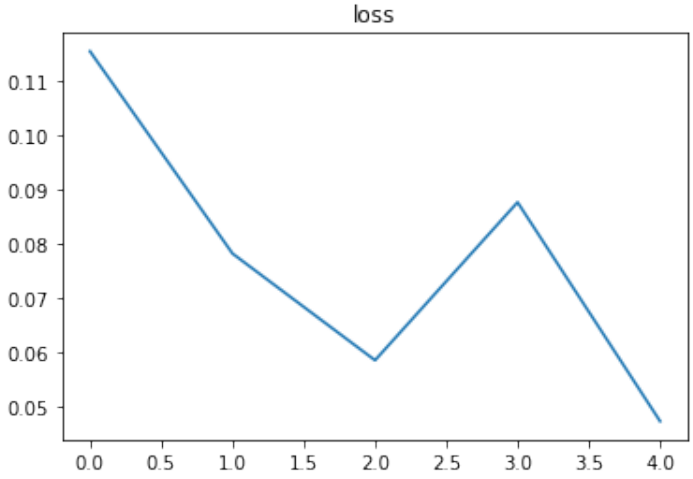

Figure 1: loss plot

- Embedding
  - It is similar to the DeepWalk without hierarchical softmax result.
  - If data is huge, this approximation algorithm is quite useful I think.

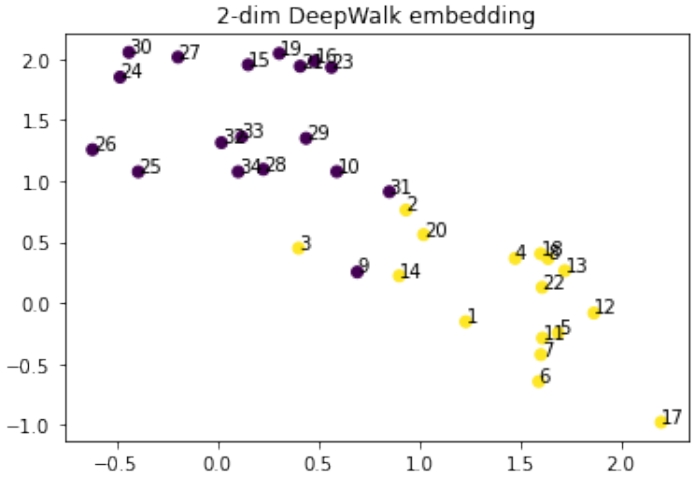

Figure 2: DeepWalk node embedding plot

```python
class Node:
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self,head):
        self.head = head
        self.left= None
        self.right= None

    def insert(self,key):
        self.current_node = self.head

        while True:
            if key < self.current_node.value:
                if self.current_node.left != None:
                    self.current_node = self.current_node.left
                else :
                    self.current_node.left = Node(key)
                    break
            else :
                if self.current_node.right !=None:
                    self.current_node = self.current_node.right
                else :
                    self.current_node.right = Node(key)
                    break

    def path(self,key):
        self.current_node = self.head
        path_list = []
        way_list = []
        while key>1:
            if key%2 ==0:
                path_list.append(int(key/2))
                way_list.append(1)
            else :
                path_list.append(int((key-1)/2))
                way_list.append(-1)
            key = int(key/2)
        return np.flip(path_list), np.flip(way_list)
```

Figure 3: binary tree code implementation

```python
class DeepWalk:
    def __init__(self,
                 adj_matrix,
                 embedding_dim=2,
                 walks_per_vertex=5,
                 walk_len=10,
                 window_size=3,
                 learning_rate=0.02):

        self.adj_matrix = adj_matrix
        self.embedding_dim = embedding_dim
        self.walks_per_vertex = walks_per_vertex
        self.walk_len = walk_len
        self.window_size = window_size
        self.learning_rate = learning_rate

        self.w1 = np.random.rand(len(adj_matrix),embedding_dim)

        self.loss = 0.0
        self.epoch_loss = []
        self.epoch_loss_de = len(self.adj_matrix)

        # randomly initialize
        # make binary tree
        self.vec = []
        start = np.random.rand(embedding_dim)
        vec.append(start)
        head = Node(0)
        self.h_softmax_tree = BinaryTree(head)

        V = len(adj_matrix)
        for key in range(1,2*V-1):
            tree_node = np.random.rand(embedding_dim)
            self.vec.append(tree_node)
            self.h_softmax_tree.insert(key)

    def _sigmoid(x):
        return 1/(1+np.exp(-x))

    def _random_walk(self, start_node: int)-> List:
        walk = [0] * self.walk_len
        walk[0] = start_node
        node = start_node
        for i in range(1, self.walk_len):
            next_node = np.random.choice(np.where(self.adj_matrix[node]==1)[0])
            walk[i] = next_node
            node = next_node
        return walk

    def _skip_gram_train(self, walk: List)-> None:
        for idx, input_node in enumerate(walk):
            # make dataset
            left_idx = idx - 3
            right_idx = idx + 3
            if left_idx < 0: left_idx = 0
            if right_idx > self.walk_len-1: right_idx = self.walk_len
            left_node = walk[left_idx:idx]
            right_node = walk[idx+1:right_idx+1]
            output_nodes = left_node + right_node

            # train
            hidden = self.w1[input_node]
            for output_node in output_nodes:
                # get path
                path, left_right = self.h_softmax_tree.path(index_to_key(output_node))
                tmp = [self.vec[i] for i in path] @ hidden
                # calcuate epoch loss
                self.loss = - np.sum(np.log(sigmoid(tmp * left_right))) / self.epoch_loss_de
                # backprop, optimization
                left_right = [1 if i==1 else 0 for i in left_right]
                EH = 0
                for i, path_val in enumerate(path):
                    tmp = sigmoid(self.vec[path_val] @ hidden) - left_right[i]
                    EH += self.vec[path_val] * tmp
                    self.vec[path_val] = self.vec[path_val] - self.learning_rate * tmp * hidden
                self.w1[input_node] = self.w1[input_node] - self.learning_rate * EH

    def train(self) -> List:
        V = np.arange(0, len(self.adj_matrix))
        for _ in range(self.walks_per_vertex):
            # shuffle vertex
            np.random.shuffle(V)
            for start_node in V:
                # random walk
                W = self._random_walk(start_node)
                # skip-gram
                self._skip_gram_train(W)
            # consider epoch as if all node be start_node
            self.epoch_loss.append(self.loss)
            self.loss = 0.0
        return self.epoch_loss

    def show_embedding(self):
        return self.w1
```

Figure 4: DeepWalk code implement