Deep Learning and Data Science (CSE5851) **Assignment5**

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

April 1, 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. Do NOT use any approximation techniques such as Hierarchical Softmax to compute 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 (refer to Assignment 4).

- loss plot

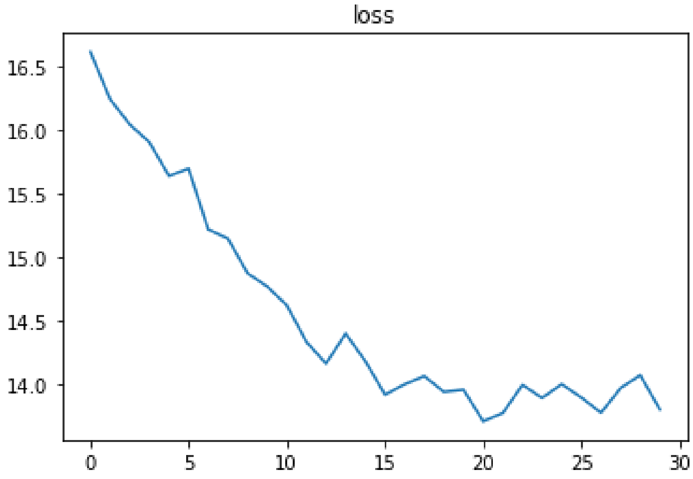


Figure 1: loss plot

- Embedding
  - DeepWalk algorithm catch the property of graph in overall case more well than MF algorithm.
  - For example, node3 is connected with opposite group not only own group in data. The MF does not catch this property well.
  - And node17 is represented well. Node17 is far from opposite group in data.

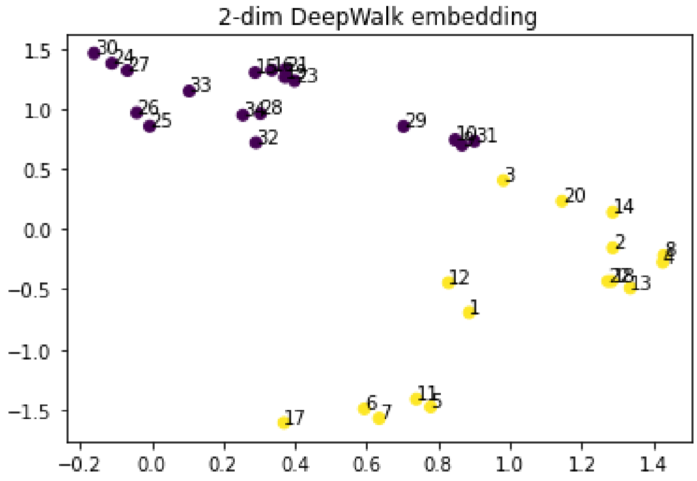
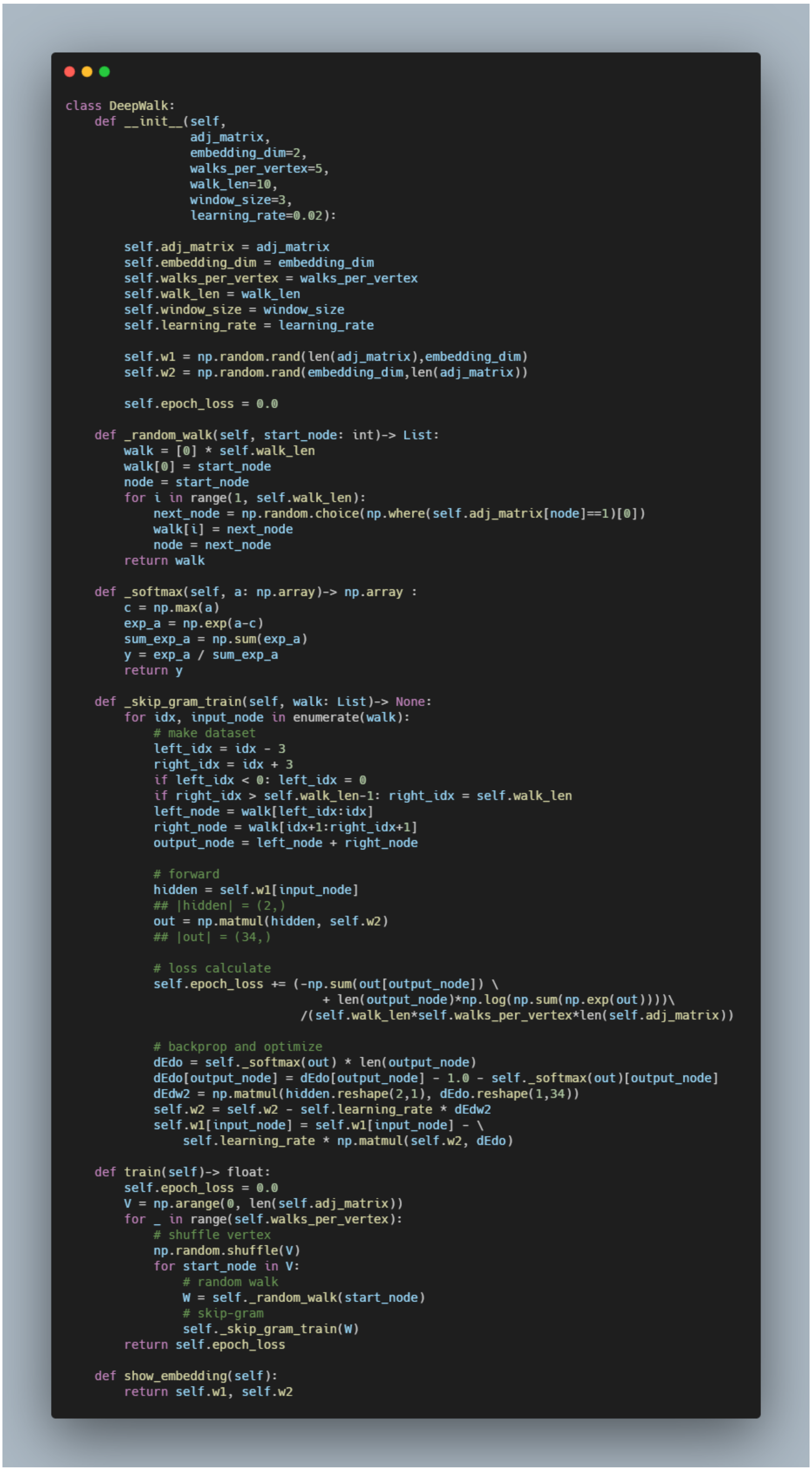
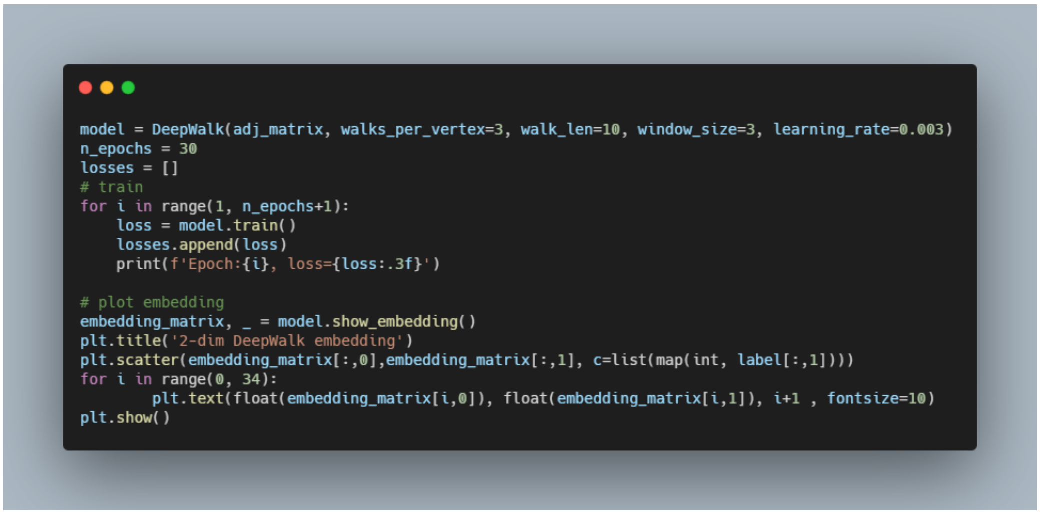


Figure 2: DeepWalk node embedding plot



```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.w2 = np.random.rand(embedding_dim,len(adj_matrix))

        self.epoch_loss = 0.0

    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 _softmax(self, a: np.array)-> np.array :
        c = np.max(a)
        exp_a = np.exp(a-c)
        sum_exp_a = np.sum(exp_a)
        y = exp_a / sum_exp_a
        return y

    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_node = left_node + right_node

            # forward
            hidden = self.w1[input_node]
            ## |hidden| = (2,)
            out = np.matmul(hidden, self.w2)
            ## |out| = (34,)

            # loss calculate
            self.epoch_loss += (-np.sum(out[output_node]) \
                                   + len(output_node)*np.log(np.sum(np.exp(out))))\
                                /(self.walk_len*self.walks_per_vertex*len(self.adj_matrix))

            # backprop and optimize
            dEdo = self._softmax(out) * len(output_node)
            dEdo[output_node] = dEdo[output_node] - 1.0 - self._softmax(out)[output_node]
            dEdw2 = np.matmul(hidden.reshape(2,1), dEdo.reshape(1,34))
            self.w2 = self.w2 - self.learning_rate * dEdw2
            self.w1[input_node] = self.w1[input_node] - \
                self.learning_rate * np.matmul(self.w2, dEdo)

    def train(self)-> float:
        self.epoch_loss = 0.0
        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)
        return self.epoch_loss

    def show_embedding(self):
        return self.w1, self.w2
```

Figure 3: DeepWalk code implementation



```python
model = DeepWalk(adj_matrix, walks_per_vertex=3, walk_len=10, window_size=3, learning_rate=0.003)
n_epochs = 30
losses = []
# train
for i in range(1, n_epochs+1):
    loss = model.train()
    losses.append(loss)
    print(f'Epoch:{i}, loss={loss:.3f}')

# plot embedding
embedding_matrix, _ = model.show_embedding()
plt.title('2-dim DeepWalk embedding')
plt.scatter(embedding_matrix[:,0],embedding_matrix[:,1], c=list(map(int, label[:,1])))
for i in range(0, 34):
        plt.text(float(embedding_matrix[i,0]), float(embedding_matrix[i,1]), i+1 , fontsize=10)
plt.show()
```

Figure 4: other code implement