Documentation HAN

April 2023

1 DATA

ACM 3025.mat, un Total de 3025 papiers (nœuds). Donné prétraité à partir de ACM.mat pour obtenir un fichier de type dict contenant les listes des attributs des nœuds, les différents méta-path, les labels, et les sets de training, de test et d'évaluation

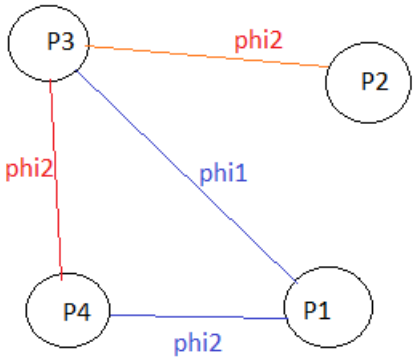
1.1 Raw data

- **Label** : Un array de taille (3025,3) , représentant le label de chaque nœuds sous la forme d'un vecteurs de taille 3 à un élément non nul de la forme [0,1,0] (lié au nombre de classes souhaité), les labels sont attribué en fonction des conférences ou elle sont publié. (Database, Wireless Communication, Data Mining).

- **feature** : Un array de taille (3025,1870), représentant les features(attributs) de chaque nœuds, il s'agit de 1870 ensembles de mots clé représenté par un vecteur de taille 1870, ou si un ensembles est présent dans le nœuds on a 1 en valeur, 0 sinon.

- **PAP, PSP ou PLP** : Matrices de taille (3025,3025), représentant les meta-path par la matrice d'adjacence de ce meta-path, c-à-d si un papier partage un auteur en commun avec un autre papier, on a 1 dans la matrice sinon on a 0.

Exemple 4 noeuds 2 meta-path :



$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Meta-path phi1

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Meta-path phi2

- **train-idx**: échantillon d'entrainement contenant 600 nœuds randomisé, représenté par un vecteur de taille 600 contenant le numéros des nœuds

- **val-idx**: échantillon de validation contenant 300 nœuds randomisé, représenté par un vecteur de taille 300 contenant le numéros des nœuds

- **test-idx**: échantillon de test contenant 2125 nœuds randomisé, représenté par un vecteur de taille 2125 contenant le numéros des nœuds

1.2 Data traité pour le gnn

- **train-mask**: Donné d'entrainement modifier par la fonction sample-mask() pour les faire passer dans la session d'entrainement, array de taille 3025 ayant la valeur True pour les coordonnées des nœuds présent dans train-idx on a alors 600 True sur 3025 , ex : si le nœuds 10 est dans train-idx, alors la coordonné 10 de train-mask sera True

- **val-mask**: Même principe que pour train-mask, on aura alors 300 valeurs True sur 3025

- **test-mask**: Même principe que pour train-mask, on aura alors 2125 valeurs True sur 3025

- **y-train**: : array de taille (600,3) contenant les labels des nœuds de test

- **y-val**: : array de taille (300,3) contenant les labels des nœuds de test

- **y-test**: : array de taille (2125,3) contenant les labels des nœuds de test

- **adj-list**:liste des meta-path sous forme de matrice ou la diagonal est enlevé, ($PAP–Id, PSP–Id$,ou $PLP–Id$)

- **fea-list**: liste de taille 3 contenant la matrice des features pour chaque élément de la liste

- **biases-list**: ajout d'un biais aux meta-paths

- **nb-nodes**: 3025 nœuds

- **ft-size**: 1870 features (1870 ensembles de mots clé)

- **nb-classes**: 3 classes

- **nb-heads**: Nb d'attention head, un nombre définie pour la multi-head attention (opération qui va améliorer l'entrainement), due au fait que la variance sera grande, alors l'entrainement sera plus stable avec cette méthode

2 Model

Models : GAT, HeteGAT, HeteGAT-multi

Les trois model hérite de la classe BaseGAttN contenant des fonctions de training et de loss. Ici le model utilisé est le model HeteGAT-multi

2.1 Fonction principale

Chaque model possède une fonction principal nommé **inference**

La fonction prend en entré : fea-list , biases-list , nb-classes, nb-nodes, n-heads.

La fonction retourne :logist(pour le calcul de la loss), final-embed(le Z (nœuds) final), att-val(les coefficients Beta) (optionnel)

- Première Boucle for : Parcours les features des nœuds donnés en entré et les meta-path, il s'agit de l'operation du choix du meta-path pour faire les opérations.

```
for inputs, bias_mat in zip(inputs_list, bias_mat_list):
```

  - Deuxième Boucle for : Multi-head attention avec K = 8 (n-heads[0]) et calcule de la Node-Level Aggregating.

```
for _ in range(n_heads[0]):
    attns.append(layers.attn_head(inputs, bias_mat=bias_mat,
                                  out_sz=hid_units[0], activation=activation,
                                  in_drop=ffd_drop, coef_drop=attn_drop, residual=False))
```

    On stock les K valeurs obtenue dans une liste *attns*

    Fin de la deuxième boucle

  - Concaténation des K tensors de la multi-head attention pour avoir les nœuds (z) après agrégation pour chaque meta-path, la liste des nœuds est stocké dans une liste L.

```
h_1 = tf.concat(attns, axis=-1)
```

  Fin premiere boucle

- Concaténation de L pour créer multi-embed.

- Opération pour calculé la Semantic-Level Aggregating, On obtient final-embed(les nœuds finaux), att-val (coeff Beta).

```
final_embed, att_val = layers.SimpleAttLayer(multi_embed, mp_att_size,
                                             time_major=False,
                                             return_alphas=True)
```

- Calcule de logits pour la loss.

```
for i in range(n_heads[-1]):

    out.append(tf.compat.v1.layers.dense(final_embed, nb_classes, activation=None))


logits = tf.add_n(out) / n_heads[-1]
```

2.2 Fonction secondaire dans inférence:

La fonction inférence utilise 2 sous fonction, attn-head et SimpleAttLayer

2.2.1 attn-head :

Prends-en entré : les features des nœuds, et le meta-path associé
En sortie : un tensors representant le z

attn-head performe l'opération de node level agreggating.

- `seq_fts = tf.compat.v1.layers.conv1d(seq, out_sz, 1, use_bias=False)`

  Cette ligne correspond au calcul des features dans leur nouvel espace elle correspond à cette formule :

  $$\mathbf{h}_i' = \mathbf{M}_{\phi_i} \cdot \mathbf{h}_i,$$

  Ou h_i correspond au features du noeuds i, M_{ϕ_i} une projection, et $h_i^{'}$ correspond a la projection des features du noeuds i.
  Dans notre cas, cette opération est effectuer uniquement par une couche de convolution.
  On obtient un tensors de taille (1,3025,8)

- ```
  f_1 = tf.compat.v1.layers.conv1d(seq_fts, 1, 1)

  f_2 = tf.compat.v1.layers.conv1d(seq_fts, 1, 1)


  logits = f_1 + tf.transpose(f_2, [0, 2, 1])
  ```

  ```
  coefs = tf.nn.softmax(tf.nn.leaky_relu(logits) + bias_mat)
  ```

  Cette partie du code correspond à:
  ```

$$\alpha_{ij}^{\Phi} = softmax_j(e_{ij}^{\Phi}) = \frac{\exp\left(\sigma(\mathbf{a}_{\Phi}^{\mathrm{T}} \cdot [\mathbf{h}_i' \| \mathbf{h}_j'])\right)}{\sum_{k \in \mathcal{N}_i^{\Phi}} \exp\left(\sigma(\mathbf{a}_{\Phi}^{\mathrm{T}} \cdot [\mathbf{h}_i' \| \mathbf{h}_k'])\right)},$$

L'oppération $^{\mathbf{a}_{\Phi}^{\mathrm{T}} \cdot [\mathbf{h}_i' \| \mathbf{h}_j']}$ est traduire par la variable logits, en particulier le terme a_{ϕ}^{T}
qui correspond au niveau d'attention du noeuds est traduit par des opération de convolution.

- ```
  vals = tf.matmul(coefs, seq_fts)
  b = tf.random.uniform(shape=[vals.shape[2]],minval=0,maxval=2)
  ret = tf.nn.bias_add(vals,b)
  ```

  Ces lignes codes l'opération final pour obtenir le terme z_i^{ϕ}, j'ai modifier le code original car la methode qu'ils ont utilisé pour ajouter du biais ne marchait plus.
  On fait le lien avec ce termes.

$$\mathbf{z}_i^{\Phi} = \sigma\left(\sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot \mathbf{h}_j'\right).$$

  Ici $ret = z_i^{\phi}$, $vals = \sum_{j \in N_i^{\phi}} \alpha_{ij}^{\phi} \cdot h_i^{'}$
  On remarque que ret est une version de $vals$ biaisé.
  La fonction σ est soit ReLu, LeakyReLu ou Linear.

2.2.2 SimpleAttLayer :

Prends-en entré : Les z calculé et concaténé, la taille du vecteurs q ici 128
En sortie : les Z finaux, et le coeff Beta

SimpleAttLayer performe la Semantic-Level Aggregating

- ```
  w_omega = tf.Variable(tf.compat.v1.random_normal([hidden_size, attention_size], stddev=0.1))
  b_omega = tf.Variable(tf.compat.v1.random_normal([attention_size], stddev=0.1))
  u_omega = tf.Variable(tf.compat.v1.random_normal([attention_size], stddev=0.1))
  ```

  Definition des variable pour :

$$\mathbf{W} \cdot \mathbf{z}_i^{\Phi_p} + \mathbf{b}, \mathbf{q}^{\mathrm{T}}$$

  On a $w\_ omega = W$, $b\_ omega = b$, $u\_ omega = q^T$

- On a ensuite les lignes suivante :

  ```
  v = tf.tanh(tf.tensordot(inputs, w_omega, axes=1) + b_omega)
  vu = tf.tensordot(v, u_omega, axes=1, name='vu')
  alphas = tf.nn.softmax(vu, name='alphas')
  ```

  qui correspond à cette formule:

$$\sum_{i \in \mathcal{V}} \mathbf{q}^{\mathrm{T}} \cdot \tanh(\mathbf{W} \cdot \mathbf{z}_i^{\Phi_p} + \mathbf{b})$$

v est l'application de tanh sur le produit de matrice $W \cdot z_i^{\phi} + b$.
vu est le produit entre q^T et $\tanh(W \cdot z_i^{\phi} + b)$
Enfin on a *alphas* qui performe l'opération $softmax$ pour avoir les coefficients:

$$\beta_{\Phi_p} = \frac{\exp(w_{\Phi_p})}{\sum_{p=1}^{P} \exp(w_{\Phi_p})},$$

- on finit par faire une somme su produit des coéfficients par les inputs (les noeuds calculé par $attn - head$):

```
output = tf.reduce_sum(inputs * tf.expand_dims(alphas, -1), 1)
```

Cette ligne correspond à la dernière partie des calculs.

$$\mathbf{Z} = \sum_{p=1}^{P} \beta_{\Phi_p} \cdot \mathbf{Z}_{\Phi_p}.$$

Avec *inputs* représentant les Z_Φ, et *alphas* les β_ϕ

3 Commentaire:

Pour ce modèle, il faut effectuer un travail de traitement de donnée pour avoir les données sous le bon format (extraire les labels, features, matrices d'adjacence sous le bon format).