ELU 501 Data science, graph theory and social network studies

Rapport du défi 2 : *The illusion of hidden personal data*

BOUKHDHIR Saif Eddine, HAFID Fayçal, SANT'ANNA Louis

Sommaire

Nous avons suivi les 6 étapes de la méthodologie CRISP :

1. Connaissance du métier

De nos jours, les marques commerciales sont de plus en plus présentes sur les réseaux sociaux pour y exercer leur e-marketing et les utiliser comme moyen de communication pour attirer plus de clients et/ou travailler leur image et leur réputation.

Dans le cas de notre défi, un restaurant souhaite trouver les personnes qui pourraient influer le maximum d'autres personnes pour faire connaître leur restaurant, et ce à travers un réseau social.

Le responsable marketing du restaurant, qui représente pour ce cas le décideur concerné par le problème, attend donc de nous qu'on lui fournisse une liste des 5 personnes qui soient les plus aptes à faire la meilleure promotion qui conduira à la plus grande visibilité possible pour le restaurant.

Pour faire cela, il faudrait disposer des localisations de chacun dans les données du réseau social, or un grand nombre d'utilisateurs ne mettent pas toutes leurs informations telle que leur localisation. En effet, dans nos données, 60% des utilisateurs n'ont pas transmis leurs informations.

Nos données sont modélisées sous forme d'un graphe, dans lequel chaque noeud représente une personne, et une arête entre deux noeuds représente l'existence d'une liaison entre les deux personnes sur Linkedin.

On modélise les informations personnelles des utilisateurs par des attributes liés aux noeuds du graphe. On souhaite alors inférer les attributs manquants à partir des attributs d'utilisateurs qu'on possède et de la structure du graphe.

Pour cela, il a fallu émettre des hypothèses :

- Hypothèses pour l'inférence d'attributs manquants :
    - On infère des attributs uniquement aux personnes qui n'ont ni localisation, ni université, ni employeur (noeud vide). En effet, si une personne possède une localisation et une université mais pas tous ses précédents employeurs, il n'est pas possible de savoir qu'il lui manque des attributs.
    - L'hypothèse de la théorie de l'homophilie : les gens forment généralement une communauté avec des personnes qui leur sont similaires.
    - On peut considérer que pour chaque personne et pour chaque attribut, elle a une valeur en commun avec au moins une personne de son entourage.
- Hypothèses pour trouver des influenceurs :
    - Une personne possède de l'influence si elle habite à San Francisco bay area et/ou qu'elle connaît des personnes qui habitent à San Francisco bay area.

Avant toute chose, on tâchera d'approfondir nos connaissances sur les données qu'on possède grâce à des statistiques et des mesures. Puis on les utilisera pour préparer les données avant d'implémenter différents modèles de prédiction et d'en choisir un selon différents critères d'évaluation.

2. Connaissance des données

Le graphe contient 811 noeuds. Un noeud a en moyenne 3.94 voisins, c'est-à-dire que chaque personne de nos données est liée à en moyenne 4 autres personnes. Aussi, chaque noeud a en moyenne 2.46 voisins vides (sans attributs/informations) et en moyenne 1.47 voisins non vides, ce qui est beaucoup car cela entend qu'un noeud tend à avoir plus de voisins vides que de voisins non vides. La Figure 1 montre des statistiques sur la distribution des attributs sur les noeuds du graphe.

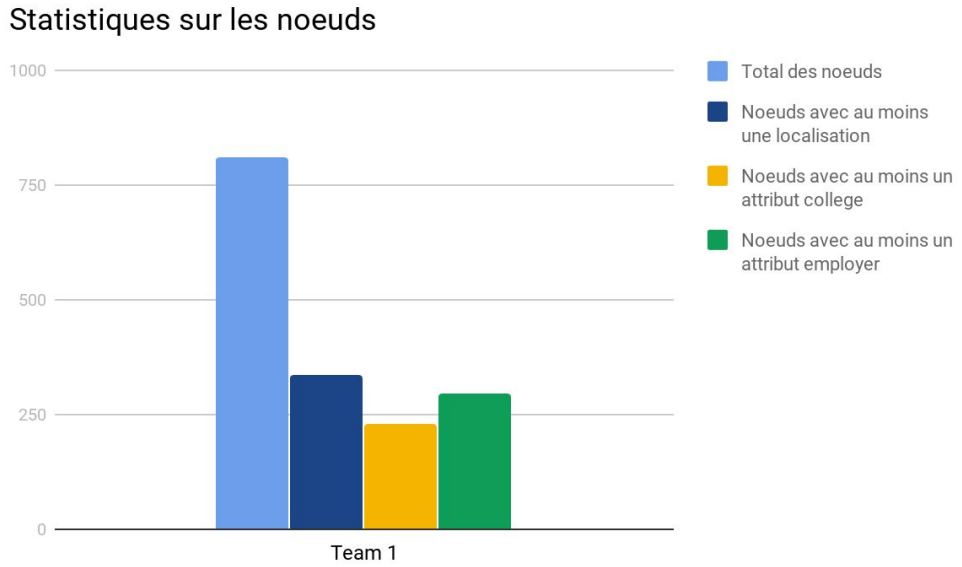

Figure 1

La Figure 1 montre que sur un total de 811 noeuds, seulement 336 personnes affichent leur localisation, ce qui représente 40%.
Parmi les données qu'on possède, 24.1% (respectivement 30.43% et 21.55%) d'entre eux ont au moins un voisin qui partage la même localisation (respectivement le même attribut college et le même attribut employer). Ces pourcentages sont considérables car ils montrent que parmi le peu de voisins

non vides d'un noeud lambda, on a une chance sur 4 de retrouver quand même son attribut dans les attributs de ses voisins, ce qui vérifie l'hypothèse de l'homophilie. Enfin, on retrouve 24 personnes dans nos données qui sont localisées à San Francisco, dans la région du restaurant. Le but est donc d'augmenter correctement ce nombre afin d'avoir plus de personnes potientielles à devenir des influenceurs.

3. Préparation des données

Avant de faire un choix de modèle, nous avons préparé nos données de manière à :

- disposer d'une liste qui énumère toutes les localisations possibles (89), une liste qui énumère toutes les valeurs possibles de l'attribut college (109) et celles de l'attribut employer (723).
- subdiviser le graphe en 20 communautés grâce à la méthode Louvain (via la fonction *best_partition* du package community de Python) sur lesquels tester les modèles.
- établir une liste des personnes vivant à san francisco.

4. Modélisation des données

En utilisant la documentation fournie sur Moodle ainsi que d'autres papiers de recherche notés en annexe, nous avons pu exploiter plusieurs modèles possibles pour l'inférence d'attributs:

1. La méthode naïve/baseline method :

Se base sur l'idée que deux noeuds voisins entre eux ont de fortes chances d'avoir la même valeur pour un attribut. Pour un noeud vide, on sélectionne donc la valeur majoritaire de l'attribut de ses voisins (ceux qui possèdent une valeur connue pour l'attribut) comme étant la prédiction de son attribut.

2. La méthode naïve améliorée :

Même idée que la méthode naïve, mais au lieu de traiter les noeuds dans un ordre aléatoire, on les traite par ordre de nombre de voisins vides croissant. Ainsi, en traitant plus tard un noeud qui aurait beaucoup de voisins vides, une partie de ces derniers aura été traitée, et on aurait alors des attributs dessus pour les utiliser dans la prédiction du noeud courant. Dans cette méthode, nous ne regardons donc pas que les voisins qui ont un attribut déjà connu mais aussi ceux qu'on aura traité auparavant au préalable.

3. Méthode inspirée de Naive Bayes :

On s'inspire de la méthode de Naive Bayes pour les probabilités conditionnelles. Pour chaque attribut, on construit une matrice M de probabilités de taille N*N, N étant le nombre de valeurs possibles de l'attribut. L'élément M_{ij} représente la probabilité d'occurrence de la valeur i sachant que la valeur j est présente dans les attributs d'au moins un voisin. Ainsi, pour chaque noeud vide, nous prendrons la valeur ayant la plus grande probabilité d'occurrence sachant les attributs de ses voisins.

4. Méthode "*Egonetwork coprofiling*" :

C'est la méthode décrite sur le papier de recherche retrouvé sur Moodle. La plupart des travaux réalisés sur cette problématique ne prennent en considération que la structure du graphe pour inférer les attributs d'un noeud, alors que dans un réseau social comme celui qu'on étudie, l'homophilie est

valable beaucoup plus sur les attributs personnels plutôt que sur les mesures liées à la théorie des graphes. Cette méthode fait aussi l'hypothèse que l'attribut d'un noeud se trouve forcément chez l'un de ses voisins. Cependant, elle propose de ne pas le chercher chez tous les voisins mais plutôt départager ces derniers en plusieurs "cercles" tels que "collègues" et "camarades d'université". Ainsi, on cherchera l'attribut employer chez les collègues seulement et l'attribut college chez les camarades d'université. Pour l'attribut location, on peut le retrouver chez l'un comme chez l'autre car on peut avoir la même localisation que nos collègues tout comme avec nos camarades d'université. La problématique repose alors sur l'affectation efficace des voisins d'un noeud en cercles de collègues et de camarades d'université, ce qui représente le travail de recherche présenté dans le papier. En finalité, cela revient, pour chaque noeud, à initialiser une attribution aléatoire d'un cercle, qu'on stockera dans un dictionnaire x (x_i étant l'affectation du noeud i), et à initialiser un vecteur de probabilités pour chaque valeur d'attribut, sous la forme d'un dictionnaire f ($f_{i,v}$ étant la valeur de la probabilité que le noeud i possède l'attribut v).
Puis, en faisant plusieurs itérations, on met à jour les différentes valeur de x et de f pour chaque noeud jusqu'à convergence. Au final, à chaque noeud sera attribué l'attribut dont la probabilité aura été maximale.

->Initialisations :

- **xi :** aléatoirement 1 ou 2 pour chaque paire de noeuds voisins. Pour deux noeuds voisins i et j ayant comme valeur x[i][j]=x[j][i]=1 (respectivement =2), ils sont collègues (respectivement camarades d'université).
- **fi :** Soit L le nombre de localisations possibles, C le nombre des universités possibles, et E le nombres d'employeurs possibles. Pour chaque noeud N, le vecteur f aura comme dimension (1,L+C+E). Chacun de ses éléments E (qui représente la valeur d'un attribut) indiquera la probabilité que N ait la valeur e comme attribut
    - Pour chaque noeud N dont on connaît les attributs :
      Chaque élément E est initialisé à 1 si N possède la valeur correspondante à l'attribut, 0 sinon.
    - Pour chaque noeud dont on ne connaît pas les attributs :
      Tous les éléments de f(N) sont initialisés à 0.5, car il a au départ la même chance d'avoir un attribut ou l'autre.

-> Mise à jour (itérations): Des équations issues du papier de recherche permettent de sélectionner une nouvelle valeur de f0 et de x0 pour le noeud courant v0 de telle sorte à maximiser l'application des hypothèses suivantes : 1- pour chaque deux noeuds vi et vj voisins entre eux, voisins avec le noeud courant v0 et appartenant au même cercle de v0, les valeurs de leurs vecteurs fi et fj doivent être proches pour les attributs communs au cercle. 2- Notre modèle doit agir de sorte à préférer sélectionner la valeur d'attribut qui se répète le plus dans les attributs des noeuds voisins dont on connaît les attributs. 3- Notre modèle doit minimiser le cardinal de l'intersection entre les deux cercles du noeud courant v0 (peu de ses amis sont à la fois ses collègues et ses camarades de classe).
Traduction en équations :

- **fi :** (issue du papier de recherche)

$$f_{i,y} = \frac{f_{0,y} + \sum_{e_{ij} \in E', v_j \in C_t} f_{j,y}}{1 + \sum_{e_{ij} \in E', n_j \in C_t} 1}, v_i \in U \cap C_t, w_{t,y} = 1$$

$$f_{0,y} = \frac{\sum_{t=1, w_{t,y}=1}^{K} \sum_{v_j \in C_t} f_{j,y}}{\sum_{t=1, w_{t,y}=1}^{K} \sum_{v_j \in C_t} 1}, w_{t,y} = 1, \forall t = 1, ...K.$$

Pseudo-code :

pour chaque n dans les noeuds vides :
    pour chaque voisin v dans les voisins de n :
        si v est dans le cercle 1 :
            f[v] <- (f[n] + somme(f[v], v : noeuds dans le cercle 1))
            f[v] /= (1 + nombre de noeuds dans le cercle 1)
        si v est dans le cercle 2 :
            f[v] <- (f[n] + somme(f[v], v : noeuds dans le cercle 2))
            f[v] /= (1 + nombre de noeuds dans le cercle 2)

- **xi :** (issue du papier de recherche, de leurs résultats on prend lambda1=0.7 et lambda2=13)

$$x_i = \arg\max_{t=1,...,K} \Big[\sum_{e_{ij} \in E', v_j \in C_t} \big(1 - \lambda_1 (w_t \cdot (f_i - f_j))^2\big) - \lambda_1 (w_t \cdot (f_i - f_0))^2 \Big], v_i \in U \quad (4)$$

$$x_i = \arg\max_{t=1,...,k} \Big[\sum_{e_{ij} \in E', v_j \in C_t} \big(1 - \lambda_1 (w_t \cdot (f_i - f_j))^2\big) - \lambda_1 (w_t \cdot (f_i - f_0))^2 - \lambda_2 (w_t \cdot f_i - 1)^2 \Big], v_i \in L \quad (5)$$

Pseudo-code :

pour chaque noeud n dans les noeuds vides :
    pour chaque noeud vi dans les voisins de n :
        si vi est un noeud vide :
            # pour le cercle 1 :
            a= somme (1 - L1 * (wt . (fi - fj))**2 , vj : voisins de n dans le cercle 1)
            #pour le cercle 2 :
            b= somme (1 - L1 * (wt . (fi - fj))**2 , vj : voisins de n dans le cercle 2)
        sinon :
            # pour le cercle 1 :
            a=somme (1 - L1 * (wt . (fi - fj))**2 - L2 * (wt . fi - 1)**2 , vj : voisins de n dans le cercle 1)
            #pour le cercle 2 :
            b=somme (1 - L1 * (wt . (fi - fj))**2 - L2 * (wt . fi - 1)**2 , vj : voisins de n dans le cercle 2)
        si a>b :
            x[vi][n] = x[n][vi] <- 1 (-> cercle 1)
        sinon :
            x[vi][n] = x[n][vi] <- 2 (-> cercle 2)

Avantages et inconvénients des méthodes :

Méthode	1	2	3	4
Avantages	Très rapide, ne demande pas de prétraitement	Rapide, tente d'utiliser même les voisins qui étaient à la base vides	Exploite la distribution des attributs sur tout le graphe et pas seulement sur le voisinage	Exploite à la fois les attributs des voisins ET la structure du graphe; prédit l'appartenance des voisins à un groupe
Inconvénients	N'exploite que les attributs des voisins dont on connaît les attributs, or certains noeuds vides ne possèdent que des noeuds vides et donc inutilisables comme voisins	N'exploite que les attributs des voisins, demande un prétraitement (déterminer pour chaque noeud vide, combien il a de voisins vides)	Prétraitement pour générer la matrice qui prend quelques minutes. L'ajout d'un nouveau noeud sur le graphe nécessiterait la mise à jour de toute la matrice	La précision augmente au fur et à mesure des itérations, une bonne précision demande beaucoup d'itérations et donc beaucoup de temps (des heures/jours de calcul)

Méthodes pour trouver les influenceurs :

1. Une première méthode :

Afin de trouver une liste des cinq influenceurs, on peut penser premièrement à calculer pour chaque noeud du graphe le nombre des voisins qui habitent à San Francisco et les répertorier dans un dictionnaire de la forme { noeud : nombre de voisins qui habitent à San Francisco } qu'on triera par ordre croissant de ses valeurs. Enfin, on prend les cinq premières personnes dans notre dictionnaire "trié" comme la liste finale des influenceurs. On note que si on cherche pour chaque influenceurs sa localisation, on ne la trouve pas forcément identique à celle du restaurant.
On peut aussi améliorer cette méthode en poussant le calcul plus loin jusqu'aux voisins des voisins. En d'autres termes, on cherche le noeud qui a un nombre maximal des voisins qui habitent à San Francisco et qui eux-mêmes ont un nombre maximal des voisins qui habitent à cet endroit. Et au final, on fait le tri selon la somme de deux nombres maximaux.

2. Une deuxième méthode :

La première méthode ne tient pas compte du fait d'avoir un voisin commun entre deux influenceurs. On peut alors l'améliorer en cherchant à maximiser le réseau des voisins de nos influenceurs.
On procède de la même façon qu'au début de la première méthode (recherche des voisins qui habitent à San Francisco). Ensuite, à chaque fois qu'on détermine les voisins vivant près du restaurant pour un noeud quelconque, on ne tient plus en compte ses voisins dans nos calculs pour les noeuds suivants. De cette façon, on évite la redondance dans les listes des voisins de nos

éventuels influenceurs et on garantie un nombre maximal des personnes qui pourraient être intéressées par le restaurant . Enfin, on fait le tri et affiche la liste de nos cinq influenceurs.

5. Evaluation

5.1. Inférence d'attributs manquants

Pour évaluer les méthodes citées dans la section 4., nous avons instancié chacune d'entre elles plusieurs fois sur chaque communauté et nous avons pris la moyenne des précisions comme étant la précision du modèle (pour la méthode 4, nous l'avons instanciée une seule fois seulement et sur tout le graphe à la fois, car une exécution contenait déjà plusieurs itérations (30 itérations) et demandait 15 heures de calculs à la machine), les résultats obtenus :

Méthode	**1**	**2**	**3**	**4**
Précision sur la localisation	33.82%	32.02%	29.62%	42.56%
Temps de calcul approximatif	< 1sec	< 1min	De l'ordre de 5-10mins	> 12 heures

On voit que malgré la différence entre les temps d'exécution, les précisions restent toutes relativement insatisfaisantes, nous n'avons pas abouti à une méthode qui permet de prédire des attributs manquants avec une précision supérieure à 80%. Cependant, le but principal étant de livrer une liste d'influenceurs, la priorité n'est pas d'inférer les attributs manquants de l'ensemble des noeuds du graphe mais d'au moins augmenter le nombre de personnes vivant dans la localisation du restaurant, ce but a été atteint (entre 6 et 10 personnes ont eu cette prédiction, selon la méthode).

5.2. Sélection des influenceurs

Les deux méthodes qu'on a citées plus haut , permettent chacune de fournir une liste des cinq influenceurs. La vraisemblance entre les deux listes est de 80%, c'est à dire que ⅘ des influenceurs sont identiques. Le temps de calcul pour les deux méthodes est négligeable (inférieur à 3ms), cela s'explique par la taille relativement petite de notre graphe (811 noeuds). D'autres méthodes auraient été de prendre parmi les noeuds qui vivent à San Francisco, ceux qui ont la plus grande centralité (idée du cours), mais ce serait une centralité absolue qui ne garantie pas que cette personne aurait beaucoup de voisins vivant à San Francisco. Il aurait été intéressant de modifier la fonction qui calcule la centralité pour lui faire prendre en compte, en plus de la structure du graphe, des attributs des noeuds, ceci pourrait faire l'objet d'un article de recherche.
Pour évaluer les deux méthodes implémentées, on a exécuté chacune en utilisant : 1- les localisations réelles (groundtruth_location), 2- les localisations prédites grâce à la méthode 4 de la section 5.1 :

	Méthode 1	**Méthode 2**	**Résultats en commun entre 1 et 2**
avec les prédictions de localisation (méthode 4	[('U27287', 11), ('U7024', 4), ('U27575', 3), ('U24095', 3),	['U27287', 'U7024', 'U24095', 'U2691', 'U22747']	3 noeuds sur 5 (souglinés) : 'U27287', 'U7024',

d'inférence d'attributs)	('U24087', 2)]		'U24095'
avec les vraies localisations (groundtruth_location)	[('U27287', 22), ('U7024', 21), ('U11566', 7), ('U27475', 5), ('U3955', 5)]	['U27287', 'U7024', 'U11566', 'U3955', 'U2691']	4 noeuds sur 5 (soulignés): 'U27287', 'U7024', 'U11566', 'U3955'
Résultats en commun entre prédictions et réalité	2 noeuds en commun (en bleu): 'U27287', 'U7024'	3 noeuds en commun (en vert): 'U27287', 'U7024', 'U2691'	

6. Déploiement et conclusion

La quatrième méthode d'inférence d'attributs manquants est la plus précise en terme de bonnes prédictions, mais demande beaucoup de temps de calcul. Ceci ne devrait pas être un problème car notre problématique ne nécessite pas de prédiction en temps réel, le fait que l'entraînement d'un modèle prenne plusieurs jours, causant un délai sur l'identification des meilleurs influenceurs, n'impacterait pas le bon fonctionnement du restaurant tant qu'il assure de meilleurs résultats par la suite.

Pour le choix de la méthode de sélection d'influenceurs, le tableau de comparaison dressé en section 5.2 montre que la méthode 2 est plus fiable car ses résultats sont plus similaires pour prédictions et pour les valeurs réelles.

En plus, elle garantit un plus grand nombre d'habitants de San Francisco en sommant le nombre des voisins uniques habitant à San Francisco pour chaque influenceur.

Ainsi, on peut conclure que la méthode 4 d'inférence d'attributs combinée avec la méthode 2 pour la sélection des influenceurs sont des solutions fiables qui ne feront qu'augmenter la visibilité du restaurant.

Annexes :

1. Bibliographie :

- User Profiling in an Ego Network: Co-profiling Attributes and relationships, Rui Li, Chi Wang, Kevin Chen-Chuan Chang
- Birds of a Feather: Homophily in Social Networks, Miller McPherson, Lynn Smith-Lovin, and James M Cook
- Homophily, wikipedia (https://en.wikipedia.org/wiki/Homophily)
- Latent Attribute Inference of Users in Social Media with Very Small Labeled Dataset (https://www.jstage.jst.go.jp/article/transinf/E99.D/10/E99.D_2016EDP7049/_pdf/-char/en)

2. Codes Python :

- Code pour lister toutes les valeurs possibles d'un attribut :

```
def getallpossible(attribute):
    attr = []
    for node in attribute:
        for a in attribute[node]:
            if a not in attr:
                attr.append(a)
    return attr
```

- Inférence d'attributs manquants :

- Méthode 2 :

```
def betterNaive(G, empty_nodes, attribute):
    #dictionnary { node : number of empty neighbors }
    nb_empty_neigh={}
    for node in empty_nodes :
        nb=0
        for neigh in G.neighbors(node):
            if neigh in empty_nodes :
                nb+=1
        nb_empty_neigh[node]=nb
    newAttr=attribute
    predictedAttr={}
    for node in sorted(nb_empty_neigh, key=lambda kv:kv[1]):
        nbrs_attr_values=[]
        for nbr in G.neighbors(node):
            if nbr in newAttr :
                for val in newAttr[nbr]:
                    nbrs_attr_values.append(val)
```

```
            predictedAttr[node]=[]
            if nbrs_attr_values:
                cpt=Counter(nbrs_attr_values)
                a, nb_occ=max(cpt.items(),key=lambda t:t[1])
                predictedAttr[node].append(a)
            newAttr={**attribute,**predictedAttr}
        return predictedAttr
```

- Méthode 3 :
  - Générer la matrice :

```
def LocMatrix(attribut, lis):
    import numpy as np
    loca=attribut
    locationsList=list(loca.keys())
    LocM=np.zeros((len(locationsList),len(locationsList)))
    ind=dict(zip(locationsList,[i for i in range(len(locationsList))]))
    inv=dict(zip([i for i in range(len(locationsList))],locationsList))
    for loc1 in locationsList :
        for loc2 in locationsList :
            n2=loca[loc1]
            n1=0
            for node in lis :
                if loc1 in lis[node] :
                    for neigh in G.neighbors(node):
                        if neigh in lis :
                            if loc2 in lis[neigh]:
                                n1+=1
            LocM[ind[loc2]][ind[loc1]]=n1/n2
    return LocM, ind, inv
```

  - Prédire à partir de la matrice :

```
def methode2():
    locations = []
    for node in location:
        for l in location[node]:
            locations.append(l)
    locations = dict(Counter(locations))
    p_l, newempty={}, list(empty_nodes)
    LocM, indL, invL=LocMatrix(locations, location)
    while newempty:
        for node in newempty :
            neigh_locs=[]
            neigh_cols=[]
            neigh_emps=[]
            for neigh in G.neighbors(node):
                if neigh in location :
                    for l in location[neigh]:
                        neigh_locs.append(l)
            scores=[]
            if neigh_locs :
                for l in neigh_locs :
                    i=np.argmax(LocM[indL[l]])
                    lo=invL[i]
```

```
                element=(lo , LocM[indL[l]][np.argmax(LocM[indL[l]])])
                scores.append(element)
            best_l, max_pr=scores[0]
            for l, pr in scores :
                if pr > max_pr :
                    max_pr=pr
                    best_l=l
            p_l[node]=[best_l]
            newempty.remove(node)
return p_l
```

- Méthode 4 :
    - Initialiser les vecteurs x :

```
def x_init(G):
    from random import randint
    xValues={}
    for node in G.nodes :
        if node not in xValues :
            xValues[node]={}
        for n in G.neighbors(node):
            x = randint(1, 2)
            if n not in xValues :
                xValues[n]={}
                xValues[node].update({n:x})
                xValues[n].update({node:x})
            else :
                if node not in xValues[n]:
                    xValues[n].update({node:x})
                    xValues[node].update({n:x})
    return xValues
```

    - Initialiser les vecteurs f :

```
def f_init(G, empty_nodes, location, employer, college):
    fVectors={}
    locations=getallpossible(location)
    colleges=getallpossible(college)
    employers=getallpossible(employer)
    totLen=len(locations)+len(colleges)+len(employers)
    indC=dict(zip(colleges,[i for i in range(0,len(colleges))]))
    invC=dict(zip([i for i in range(0,len(colleges))],colleges))
    indE=dict(zip(employers,[i for i in
range(len(colleges),len(colleges)+len(employers))]))
    invE=dict(zip([i for i in
range(len(colleges),len(colleges)+len(employers))],employers))
    indL=dict(zip(locations,[i for i in
range(len(colleges)+len(employers),totLen)]))
    invL=dict(zip([i for i in
range(len(colleges)+len(employers),totLen)],locations))
    ind={**indC, **indE, **indL}
    inv={**invC, **invE, **invL}
    omega1=[0 for i in range(len(colleges))]
```

```
    omega1.extend([1 for i in range(len(employers))])
    omega1.extend([1 for i in range(len(locations))])
    omega2=[1 for i in range(len(colleges))]
    omega2.extend([0 for i in range(len(employers))])
    omega2.extend([1 for i in range(len(locations))])
    for node in G.nodes :
        if node not in empty_nodes :

             fVectors[node]=np.zeros((1,totLen)).tolist()[0]
            if node in location :
                for loc in location[node]:
                    fVectors[node][ind[loc]]=1
            if node in college :
                for col in college[node]:
                    fVectors[node][ind[col]]=1
            if node in employer :
                for emp in employer[node]:
                    fVectors[node][ind[emp]]=1
        else :
            fVectors[node]=[0.5 for i in range(totLen)]
    return fVectors, omega1, omega2, ind, inv, locations, colleges,
employers
```

- Prédire étant donné des vecteurs f :

```
def predict(empty_nodes, location, employer, college, fVectors,
colleges, employers, inv):
    pred_l, pred_c, pred_e=[], [], []
    for node in empty_nodes :
        fVec=fVectors[node]
        colVect=fVec[:len(colleges)]
        empVect=fVec[len(colleges):len(colleges)+len(employers)]
        locVect=fVec[len(colleges)+len(employers):]
        bestColleges=np.argwhere(colVect ==
np.amax(colVect)).flatten().tolist()
        bestEmployers = np.argwhere(empVect ==
np.amax(empVect)).flatten().tolist()
        bestLocations= np.argwhere(locVect ==
np.amax(locVect)).flatten().tolist()
        if len(bestColleges)<2:
            pred_c[node] = []
            for col in bestColleges :
                pred_c[node].append(inv[col])
        if len(bestEmployers)<5:
            pred_e[node] = []
            for emp in bestEmployers :
                pred_e[node].append(inv[emp+len(colleges)])
        if len(bestLocations)<2:
            pred_l[node] = []
            for loc in bestLocations:
                    pred_l[node].append(inv[loc+len(colleges)+len(employer
             s)])
```

```
            return pred_l, pred_e, pred_c
```

- Faire les itérations :

```
def coprofiling(G, empty_nodes, location, employer, college):
   location2, employer2, college2, newempty= location, employer, college,
empty_nodes
   fVectors, omega1, omega2, ind, inv, locations, colleges, employers = f_init(G,
newempty, location2, employer2,college2)
   xValues = x_init(G)
   nbnode = 0
   for xm in range (3) :
       for empty_node in newempty:
           print("Ongoing : %d / %d" % (nbnode, len(newempty)))
           fVectorsnext = fVectors
           circle1 = []
           circle2 = []
           for num in range(30):
               # print("Iteration %d sur 10"%num)
               # UPDATE f
               for neigh in G.neighbors(empty_node):
                   if xValues[neigh][empty_node] == 1:
                       circle1.append(neigh)
                   else:
                       circle2.append(neigh)
               for neigh in circle1:
                   if neigh in newempty:
                       L = len(omega1)
                       for i in range(L):
                           if omega1[i] == 1:
                               fVectorsnext[neigh][i] = fVectors[empty_node][i]
                               for n2 in circle1:
                                   if n2 != neigh:
                                       fVectorsnext[neigh][i] += fVectors[n2][i]
                               fVectorsnext[neigh][i] /= 1 + len(circle1)
               for neigh in circle2:
                   if neigh in newempty:
                       L = len(omega2)
                       for i in range(L):
                           if omega2[i] == 1:
                               fVectorsnext[neigh][i] = fVectors[empty_node][i]
                               for n2 in circle2:
                                   if n2 != neigh:
                                       fVectorsnext[neigh][i] += fVectors[n2][i]
                               fVectorsnext[neigh][i] /= 1 + len(circle2)
               for i in range(L):
                   den = 1
                   if omega1[i] == 1:
                       den += len(omega1)
                       for neigh in circle1:
                           fVectorsnext[empty_node][i] += fVectorsnext[neigh][i]
                   if omega2[i] == 1:
```

```
                        den += len(omega2)
                        for neigh in circle2:
                            fVectorsnext[empty_node][i] += fVectorsnext[neigh][i]
                    fVectorsnext[empty_node][i] /= den
                fVectors = fVectorsnext
                # UPDATE X
                xValuesnext = xValues
                for neigh in G.neighbors(empty_node):
                    soust = [fi - f0 for (fi, f0) in zip(fVectors[neigh],
fVectors[empty_node])]
                    a = -0.7 * ((np.dot(omega1, soust)) ** 2)
                    b = -0.7 * ((np.dot(omega2, soust)) ** 2)
                    for nj in circle1:
                        soust = [fi - fj for (fi, fj) in zip(fVectors[neigh],
fVectors[nj])]
                        a += 1 - 0.7 * ((np.dot(omega1, soust)) ** 2)
                    for nj in circle2:
                        soust = [fi - fj for (fi, fj) in zip(fVectors[neigh],
fVectors[nj])]
                        b += 1 - 0.7 * ((np.dot(omega2, soust)) ** 2)
                    if neigh not in empty_nodes:
                        a += -7 * ((np.dot(omega1, fVectors[neigh]) - 1) ** 2)
                        b += -7 * ((np.dot(omega2, fVectors[neigh]) - 1) ** 2)
                    if a > b:
                        xValuesnext[neigh][empty_node] = 1
                        xValuesnext[empty_node][neigh] = 1
                    else:
                        xValuesnext[neigh][empty_node] = 2
                        xValuesnext[empty_node][neigh] = 2
                xValues = xValuesnext
            nbnode += 1
            predLoc, predEmp, predCol = predict(empty_nodes, location, employer,
college, fVectors, colleges, employers, inv)
            if predLoc :
                print("Location (",len(predLoc.items()),") : ",
evaluation_accuracy(groundtruth_location, predLoc))
            if predEmp :
                print("Employers (",len(predEmp.items()),") : ",
evaluation_accuracy(groundtruth_employer, predEmp))
            if predCol :
                print("Colleges (",len(predCol.items()),") : ",
evaluation_accuracy(groundtruth_college, predCol))
    return xValues, fVectors, predLoc, predEmp, predCol
```

- Sélection des influenceurs
  - Première méthode :

```
def influenceurs(G,location):
    restau_location=['san francisco bay area']
    list_influenceurs=[]
```

```
        for node in G.nodes():
    #         if node in location.keys()and location[node]==restau_location:
            k=0
            for neig in G.neighbors(node):
                if neig in location.keys():
                    if location[neig]==restau_location:
                        k+=1
            if k>0:
                list_influenceurs.append((node,k))
        list_influenceurs=sorted(list_influenceurs,key=lambda x:
    x[1],reverse=True)
        return(list_influenceurs[:5])
```

- Seconde méthode :

```
def get_influencer(dic):
    dictx=list()
    for node in dic:
        dictx.append((node,len(dic[node])))
    dictx=sorted(dictx, key=lambda x: x[1], reverse=True)
    return(dictx[0][0])
def update(dic,liste):
    for ele in liste:
        for node in dic:
            if ele in dic[node]:
                dic[node].pop(dic[node].index(ele))
    return(dic)
def find_infl(G,location):
    restau_location=['san francisco bay area']
    influencers={}
    list_influencers=[]
    for node in G.nodes():
        liste=list()
        for nb in G.neighbors(node):
            if nb in location.keys():
                if location[nb]==restau_location:
                    liste.append(nb)
        influencers[node]=liste
    while len(list_influencers)<5:
        if len(influencers.keys())>0:
            best_influencer=get_influencer(influencers)
            list_influencers.append(best_influencer)
            a=influencers[best_influencer]
            del influencers[best_influencer]
            influencers=update(influencers,a)
    return(list_influencers)
```