Mathieu REINE GI06 Rapport de TX

Gestion des incertitudes pour le diagnostic apprenant.

Doctorant suiveur :

Mme.RAHMOUNI Nesrine

Enseignant suiveur :

Mme.LOURDEAUX Domitile

Sommaire :

1.Introduction

Dans le but de déterminer l'acquisition de compétence d'apprenants dans le domaine de l'algorithmique, Nesrine qui est doctorante à l'UTC a déterminé des modèles qui pourront :

- Diagnostiquer les compétences évaluées d'un apprenant.
  - Le modèle Bayésien ne tiens pas assez compte des incertitudes dues à la problématique de l'acquisition de compétence.
  - On utilise alors des fonctions de croyance et de révision.

- Prédire les compétences non évaluées.
  - Algorithmique : les compétences du domaine étudié sont liées entre elles
  - Elle fait donc l'hypothèse qu'il est possible de prédire l'état d'acquisition d'une compétences en partant d'évaluations d'autre compétence liée
  - Dans ce but elle utilise une ontologie basée sur le lien de requis et de la propagation des masses de croyance.

J'ai pu dans le cadre de ma TX l'assister dans son projet en m'occupant de la partie implémentation informatique de ses modèles, principalement des fonctions back end (traitant de la partie modèle la solution), j'ai aussi mis en place une interface graphique qui permet de se servir de ces fonctions back end ; cette dernière n'étant pas la priorité de cette TX elle ne sera pas traité dans les détails dans ce rapport.

Ces modèles et les fonctionnalités qu'il implique seront traités tout au long de ce rapport, les spécificités de la solution informatique afin de répondre à leurs besoins seront aussi détaillé et illustré par du code qui sera remis en même temps.

2.Structures de donnée

2.a. Compétences

Le but premier de cette structure de données pour les compétences est la création d'une ontologie :

```
class Evaluation(): #Ici on définit la classe Evaluation pour la création des onthologies
                    #Elle est composée de son nom, de ceux qu'il requière et de ceux dont il est requis
                    #dernier attribut est peu util mais bien pour voir si erreur



    def __init__(self): #Son initialisation
        self.nom = 'àdef'
        self.isArequirementTo = [[],] #Toutes les évaluations qui requièrent celle-ci et à quelle distance
        self.requires = [[],]#Toutes les évaluations qui ont comme requierement celle-ci et à quelle distance
        Evaluations.append(self)#On garde un historique de toutes les évalutations du système


    def change_nom(self,new_nom): #La méthode permettant de changer de nom
        self.nom =new_nom

    def add_requirement(self,evaluation): # La méthode permettant la création des requirements

        self.requires.append([evaluation,1])
        self.requires[0].append(evaluation.nom)

        evaluation.isArequirementTo.append([self,1])
        evaluation.isArequirementTo[0].append(self.nom)
```

Pour parvenir à cette création nous avons besoin d'un moyen d'identifier les compétences, on leur donne donc un nom, et il nous faut les liens qui sont ici hiérarchique (relation de requérant/requis), nous attachons donc à chacune des compétences une liste des requis qui va d'abord stocker les noms des compétences requises, puis des tuples contenant à la fois le « pointeur » vers la compétence requise et la distance entre les deux.

Afin de garder une trace des données créer, les compétences se placent à leur création dans un tableau qui sert de variable global. (Les listes sauvegardent les changements subis dans une fonction contrairement au variables simples).

Remarques :

Il y a également une liste des compétences pour lesquelles nous sommes un requis, mais comme il est expliqué dans le script elle n'est pas utilisée ici, c'est également le cas de la distance qui aurait pu être implémenter plus tard.

J'ai néanmoins décider de garder cette structure pour éventuellement servir de base à une fonctionnalité que je n'ai pas mise en place : la taxonomie de l'ontologie (comme évoqué plus tard dans le rapport).

2.b. Elèves et Evaluations

Nous devons ici créer 2 structures : une pour les élèves qui auront chacun un certain nombre d'évaluations et une autre pour les évaluations en elles-mêmes.

```
class Eleve(): #Ici on définit la classe Evaluation pour la création des onthologies
                    #Elle est composée de son nom, de ceux qu'il requière et de ceux dont il est requis
                    #dernier attribut est peu util mais bien pour voir si erreur

    def __init__(self): #Son initialisation
        self.nom = 'àdef'#Nom de l'eleve
        self.Evaluations_note_date=[]#l'ensemble des évaluations de l'eleve, l'évaluation sera définie dans la cl
        Eleves.append(self)#On garde un historique de toutes les évalutations du système


    def change_nom(self,new_nom): #La méthode permettant de changer de nom
        self.nom =new_nom
```

Chaque élève aura besoin d'un nom pour être identifié (si homonyme, il faudra les différencier d'une façon ou d'une autre : Mathieu et Mathieu2 …).

Ils auront également chacun une liste d'évaluations qui sera constituée d'évaluations qui répondront à la structure que nous détaillerons en-dessous.

Enfin afin de sauvegarder les données déjà entrées, les élèves vont être stockés à leur création dans une liste qui servira de variable globale (concept expliqué pour la structure de la compétence).

```
class Evaluation_note_date():#Ici on définit la classe qui est déjà utilisée à l
                            #Evaluations_note_date de la classe Eleve ; cette de
                            #Elle est composé de son id unique, de l'élève assoc
    def __init__(self): #Son initialisation
        self.eleve = eleve_actif[0]
        self.eleve.Evaluations_note_date.append(self)
        self.id = len(eleve_actif[0].Evaluations_note_date)
        self.type_ev = 'none'
        self.note = 0
        self.date = date.today().strftime('%d/%m/%Y')


    def change_type(self,new_type_ev): #La méthode permettant de changer de nom
        self.type_ev =new_type_ev

    def change_id(self,new_id): #La méthode permettant de changer de nom
        self.id =new_id

    def change_note(self,new_note):
        self.note = new_note

    def change_date(self,new_date):
        self.date= new_date
```

Les évaluations contiennent un nom de compétence, une note et une date.

Elles sont également automatiquement associées à l'élève actif lors de leur création ; et donc stockées directement dans son tableau contenant toutes ses évaluations.

Cette structure permettra d'utiliser la transformation de masses (sa note) et la fonction de révision (ses masses et ses dates), la fonction de prédiction (l'ontologie, son nom, et ses révisions).

Le fait que les évaluations soient stockées dans la classe élèves permet de découper les processus par élèves et ainsi de stocker les résultats par élèves.

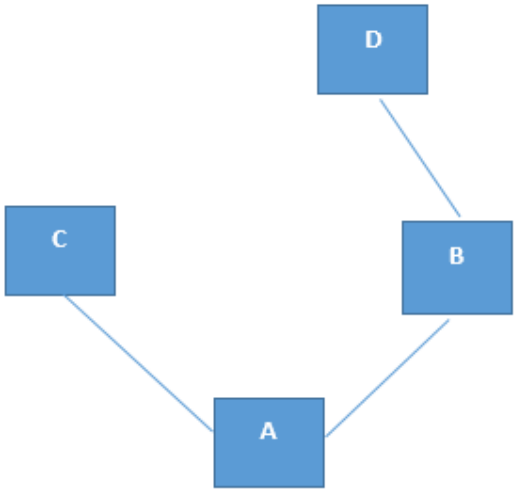
3.Fonctionnalités à implémenter

3.a. Création de l'ontologie

Création de l'ontologie :

Comme suit :

- B et C sont des prérequis de A.
- D est prérequis de B.



L'ontologie que nous cherchons à créer pour prédire l'acquisition des compétences non évaluées se basent sur des relations de requis/requérant.

La création de cette ontologie se basera sur la structure que nous avons défini pour les compétences : c'est-à-dire des objets de class Compétence principalement constitués d'un nom et d'une liste contenant ses requis (si la compétence en a).

Nous procédons d'abord à l'ajout de toutes les relations de distance 1 dans l'ontologie via la méthode de notre class add_requirement qui ajoute la compétence désignée dans les requis de notre compétence de départ.

```
def add_requirement(self,evaluation): # La méthode permettant la création des requirements

    self.requires.append([evaluation,1])
    self.requires[0].append(evaluation.nom)

    evaluation.isArequirementTo.append([self,1])
    evaluation.isArequirementTo[0].append(self.nom)
```

Une fois que nous avons rentré toute les relations de distance 1, il faut que l'on propage la relation de requis dans le graphe (si A requiert B et B requiert C, alors A requiert C à une distance de 2). De plus nous voulons garder le chemin optimal pour chacune des relations.

Nous créons alors un tableau des liaisons, nous vérifions pour toutes les compétence les cas où un requis est aussi un requérant et nous propageons la relation.

```
def Extend_Lier(Liaisons,a): #Cette fonction retourne le tableau des liaisons en connectant à une distan
    for Liai in Liaisons :
        stock_new_Liai = [] #On s'apprête à stocker de nouvelles liaisons
        one_who_requires = Liai[0] #Pour plus de lisibilité
        one_that_is_required = Liai[1]
        if one_that_is_required in a: #Si celui que l'on dont on a besoin a également besoin de quelqu'u
            for Liai2 in Liaisons : #Pour chaque liaison
                if one_that_is_required == Liai2[0]: #Si c'est l'une de ses liaisons
                    stock_new_Liai.append([one_who_requires,Liai2[1],Liai[2]+1]) #On ajoute la nouvelle
        if len(stock_new_Liai)>0:                #[Celui qui requiert, Celui que requiere celui qui est r
            Liaisons.extend(stock_new_Liai) #On ajoute à Liaisons
    return(Liaisons)#On retourne le nouveau tableau
```

Après avoir récupérer les liaisons d'une profondeur supplémentaire, nous ne gardons que la distance minimale pour un tuple (requis, requérant).

```
def Minimize_Dist(Liaisons):    #Cette fonction fait en sorte de ne garder que la distance minimale de chacune
    ar = np.array(Liaisons)
    df = pd.DataFrame(ar, columns = ['one_who_requires', 'one_that_is_required', 'distance']) #Transformations
    df1 = df.drop_duplicates(subset=['one_who_requires','one_that_is_required'], keep='first')#On élimine les d
    try :
        df1['one_who_requires'] = df1['one_who_requires'].apply(lambda x : x.nom)#On prend le nom de l'évaluati
        df1['one_that_is_required'] = df1['one_that_is_required'].apply(lambda x : x.nom)
    except:
        pass

    df1.to_csv('eval_ontologie.csv', sep = ';',index = False)#On stocke l'ontologie dans un tableau csv
    Liaisons=df1.values.tolist()#On retransforme en tableau
    return(Liaisons)#On le retourne
```

En utilisant ce processus n fois (où n est le nombre de compétence de l'ontologie), on a l'ontologie complète et aux distance minimale.

Une ligne de ce tableau de liaisons se présente comme ceci : [requérant, requis, distance].

Nous envoyons ensuite ce tableau dans un tableau csv afin de le garder en mémoire pour une utilisation de la fonction de prédiction de masses.

	A	B	C
1	one_who_re	one_that_is_	distance
2	Histoire	Math	1
3	Histoire	Geographie	1
4	Geographie	Math	1
5	Geographie	Coloriage	1
6	Coloriage	Couleur	1
7	Histoire	Coloriage	2
8	Geographie	Couleur	2
9	Histoire	Couleur	3
10			

3.b. Transformation de masse

L'une des premières fonctionnalités que l'outil devait traiter était naturellement la fonction de transformation de masse, elle sert à associer à une note des masses qui illustre l'état d'apprentissage de la compétence.

Masses définie telle que :

$m(a) = m$ **(Acquise)** est la masse de croyance attribuée à l'acquisition de la compétence par l'apprenant.
$\mathbf{m}(\neg \mathbf{a}) = m$ **(Non acquise)** est la masse de croyance attribuée à la non acquisition de la compétence par l'apprenant.
$m(i) = m$ **{Acquise, Non acquise}** est la masse de croyance sur l'ignorance qu'a le système quant à **l'acquisition** ou **la non acquisition** de la compétence par l'apprenant.
$m(c) = m$ est le conflit résultant de la contradiction entre les observations pour la même compétence.

Avec: $m(a) + \mathbf{m}(\neg \mathbf{a}) + m(i) + m(c)=1$.

La fonction de transformation de masse qui a été choisie est de cette forme, l'état est forcément acquis ou non acquis quand on s'éloigne de r ou plus de x ; tandis qu'il suit un comportement plus mitigé entre (x-r) et (x+r): (ici x =10 et r=5)

Note **ϵ [0,5[** alors **m(a) =0, m(¬a) =1, m(i)=0, m(c)=0 et état =non acquise**

Note **ϵ]15,20]** alors **m(a) =1, m(¬a) =0, m(i)=0, m(c)=0 et état = acquise**

Note **ϵ [5,10[** alors **m(a) =0, m(¬a) >0(à déterminer), m(i)>0, m(c)=0 et état = probablement non acquise**

Note **ϵ [10,15[** alors **m(a) >0, m(¬a) =0, m(i)>0, m(c)=0 et état= probablement acquise**

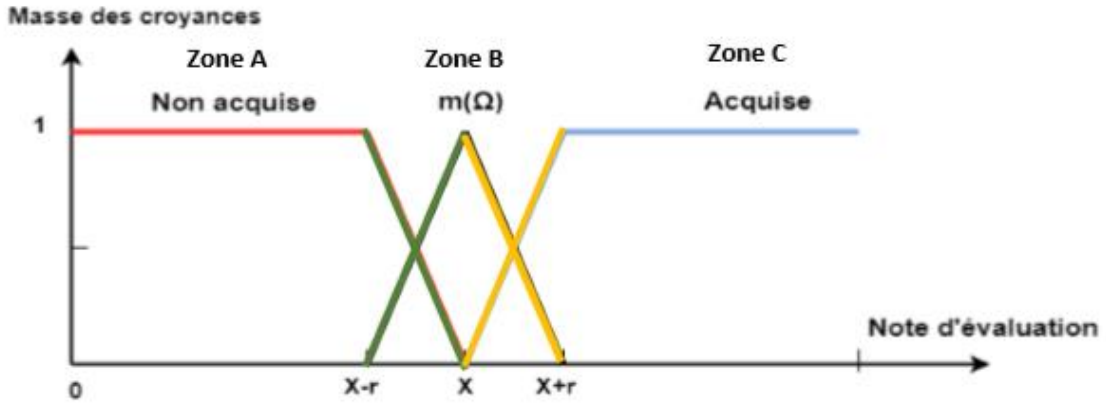
L'outil permet de changer les paramètre x et r :

```
param = [] #On init le tableau param

def Definition_parametre(x,r):
    x = float(x)#Au cas où x et r ne sont pas des float
    r = float(r)
    param.append(x)#param = [x,r]
    param.append(r)
    return(param)#On retourne les paramètres
```

Il a ensuite été choisi que le comportement des masses dans cette zone intermédiaire serait affine.

Graphique illustrant le modèle choisi pour la transformation de masse.



L'utilisateur fixant x et r, on peut déterminer les fonctions affines suivie par chacune des composantes des masses, et ceux dans toute les zones possibles pour l'évaluation.

Cela donne 2 façon de le calculer :

Fonction affine

```
def masses_x(note,param): #champs : ma, mna, mi, mc

    m = x = float(param[0]) #au cas où param[0] n'est pas un float
    r = float(param[1]) #au cas où param[1] n'est pas un float
    note = float(note) #au cas où note n'est pas un float
    if note<=(x-r): #si note < x-r
        mna=1 #Clairement non acquis
        ma=mi=mc=0
    elif note>=(x+r): #si note > x+r
        ma=1 #Clairement acquis
        mna=mi=mc=0
    elif note<=x:#si entre x-r et x
        mna = m/r - (1/r)*note #m(nona) = x/r - (1/r)*note
        mi = (r-m)/r + (1/r)*note #m(i) = (r-x)/r + (1/r)*note
        ma=mc=0
    else: #si entre x et x+r
        mi = r-m/r - (1/r)*note #m(i) = (r-x)/r -(1/r)*note
        ma = -m/r + (1/r)*note #m(a) = (-x)/r + (1/r)*note
        mna=mc=0
    Masses=[ma,mna,mi,mc]
    return(Masses) #On retourne les masses
```

En faisant un changement de variable :

```
def masses_cgt(note,param):#champs : ma, mna, mi, mc
    for truc in param:
        truc = float(truc) #au cas ou l'entrée ne soit pas du type float, (si passé par l'interface)
    note = float(note)# même chose
    if note<=(param[0]-param[1]):#si moins de x-r
        mna=1
        ma=mi=mc=0#Clairement non acquise
    elif note>=(param[0]+param[1]): #Si plus que x+r
        ma=1
        mna=mi=mc=0 #Clairement acquis
    elif note<=param[0]: #si entre x-r et x
        mna = (1/2) - (note-(param[0]-(param[1]/2)))/param[1] #m(nona) = 0.5 - (note -(x-r/2))/r
        mi = (1/2) + (note-(param[0]-(param[1]/2)))/param[1] #m(i) = 0.5 + (note -(x-r/2))/r
        ma=mc=0
    else: #si entre x et x+r
        mi = (1/2) - (note-(param[0]+(param[1]/2)))/param[1]#m(i) = 0.5 -(note -(x+r/2))/r
        ma = (1/2) + (note-(param[0]+(param[1]/2)))/param[1]#m(a) = 0.5 +(note -(x+r/2))/r
        mna=mc=0
    Masses=[ma,mna,mi,mc]
    return(Masses)#On retourne les masses
```

L'outil peut utiliser les deux fonctions de transformation de masse.

Voici l'un des deux scripts permettant d'effectuer cette transformation sur tout un tableau csv :

(Il créer ensuite un tableau csv contenant les masses).

```
rep_principal = os.getcwd ()


def masses_file(f): #Fonction qui prend un fichier f et qui effectue la transformation de masses sur chacune des éval
    os.chdir(rep_principal+'\\eleves')#On va dans le dossier des élèves
    df = pd.read_csv(f,sep=';')#On créer un dataframe grâce au tableau csv
    df.astype({'note': 'float'}) #Si non float alors il devient un float

    df['masses'] = df['note'].apply(lambda x:masses_cgt(x,param)) #On applique à la colonne des notes la fonction de
    df = df.sort_values(by=['date'])#On classe les évaluations dans l'ordre de date
    df = df.reset_index(drop=True)#On reset les index
    os.chdir(rep_principal+'\\eleves_masses')#On se place dans le dossier des masses des élèves
    df.to_csv(f[:-4]+'_masses.csv', sep = ';',index = False)#On créer un tableau csv avec les masses
```

Un équivalent existe pour l'autre fonction de transformation.

3.c. Fonction de révision

Fonctionnalité 2 : Fonction de révision

Révision positive :

$$m(a_R) = \frac{m(a_I)}{m(a_I)+m(\neg a_I)+m(i_I)} * m(i_A) + m(a_A)$$

$$m(\neg a_R) = \frac{m(\neg a_I)}{m(a_I)+m(\neg a_I)+m(i_I)} * m(i_A) + m(\neg a_A)$$

$$m(i_R) = m(i_I) * m(i_A)$$

$$m(c_R) = 1 - m(a_R) - m(\neg a_R) - m(i_R)$$

```
#Positif

def revision_pos(masse_now,masse_ant): #Masse now : masse de la nouvelle note, masse ant : masse d'avant
    masse_rev=[0,0,0,0]
    masse_rev[0]=((masse_now[2]*masse_now[0])/(masse_ant[0]+masse_ant[1]+masse_ant[2]))+masse_now[0]
    masse_rev[1]=((masse_now[2]*masse_now[1])/(masse_ant[0]+masse_ant[1]+masse_ant[2]))+masse_now[1]
    masse_rev[2]=masse_ant[2]*masse_now[2]
    masse_rev[3]=1-masse_rev[0]-masse_rev[1]-masse_rev[2]
    return(masse_rev)
```

Avec :

- $(m(a_I), m(\neg a_I), m(i_I), m(c_I))$ est la distribution de masse de croyance initialement affectée à la compétence **Cp** à partir de la note d'évaluation ;
- $(m(a_A), m(\neg a_A), m(i_A), m(c_A))$ est la distribution de masse de croyance affectée à la compétence **Cp** à partir de la nouvelle note d'évaluation ;
- $(m(a_R), m(\neg a_R), m(i_R), m(c_R))$ est la distribution de masse de croyance affectée à la compétence **Cp,** et révisée par la nouvelle note d'évaluation.

Cette fonction est à utiliser quand l'utilisateur passe d'une zone incertaine (entre x-r et x+r) à une zone de certitude (évaluation donne plus d'information), ou quand l'évaluation reste dans la même zone mais est meilleure que la précédente.

Révision négative

La révision des valeurs : (m(a), m(¬a), m(i), m(c))

$$m(a_R) = \frac{m(a_I) + m(a_N)}{2}$$

$$m(\neg a_R) = \frac{m(\neg a_I) + m(\neg a_N)}{2}$$

$$m(i_R) = \frac{m(i_I) + m(i_N)}{2}$$

$$m(C_R) = 1 - m(a_R) - m(\neg a_R) - m(i_R)$$

```python
#Négatif

def revision_neg(masse_now,masse_ant):#Masse now : masse de la nouvelle note, masse ant : masse d'avant
    masse_rev=[0,0,0,0]
    masse_rev[0]=(masse_now[0]+masse_ant[0])/2
    masse_rev[1]=(masse_now[1]+masse_ant[1])/2
    masse_rev[2]=(masse_now[2]+masse_ant[2])/2
    masse_rev[3]=1-masse_rev[0]-masse_rev[1]-masse_rev[2]
    return(masse_rev)
```

Au contraire cette révision est à utiliser quand l'évaluation se trouve dans la même zone que la dernière évaluation et que la nouvelle évaluation est moins bonne, que la nouvelle évaluation est entre (x-r) et x alors que l'ancienne était entre x et (x+r), ou que l'on passe d'une zone de certitude d'acquisition, à une zone de certitude de non acquisition.

Quelle fonction utiliser ?

```python
#Définition de quel modèle utiliser en fonction des 2 notes

def revision(masse_now,masse_ant):#Masse now : masse de la nouvelle note, masse ant : masse d'avant
    if (masse_now[0]==1 or masse_now[1]==1) and (masse_ant[0]!=1 and masse_ant[1]!=1): #Passage de zone incertaine
        return(revision_pos(masse_now,masse_ant))#On utilise la révision positive
    elif ((masse_now[0]>=masse_ant[0]) or ((masse_now[0]==0 and masse_ant[0]==0) and (masse_now[2]>=masse_ant[2]))
        return(revision_pos(masse_now,masse_ant))#On utilise la révision positive
    else :#Sinon : pas de passage à zone certitude et résultat moins bon
        return(revision_neg(masse_now,masse_ant))#On utilise la révision négative
```

On définit cette fonction qui permet de reconnaitre dans quel cas de figure nous nous trouvons et ainsi utiliser la bonne fonction de révision.

```python
#Fonction finale : transformation des datas en dataframe, calcul des révisions et réécriture en csv
def revision_masses_df(df,nom_eleve):
    rep_principal = os.getcwd()
    df.sort_values(by=['date'])    #On trie par date pour que les revisions se fasse bien dans le temps
    evals = [] #Afin de ne réviser que sur les évaluations faites au moins 2 fois, on stocke les évaluations déjà v
    evals_masses = []#Pour stocker l'historique des évaluations et leurs masses déjà effectuées (savoir les masses
    df['compte'] = df['matiere'].apply(lambda x:x.capitalize())# Init de la colonne compte
    df['durée'] = df['date']#on initialise la durée à la date
    try: #En fonction de la fonction choisie
        df['revision'] = df['note'].apply(lambda x:masses_x(x,param)) #On définit par défaut la première révision
        df['masses'] = df['note'].apply(lambda x:masses_x(x,param))
    except :
        df['revision'] = df['note'].apply(lambda x:masses_cgt(x,param))#On définit par défaut la première révisior
        df['masses'] = df['note'].apply(lambda x:masses_cgt(x,param))
    for u in range(len(df)):#Pour toutes les évaluations
        df['compte'] = df['matiere'][0:u].eq(df['matiere'][u]).sum() #fonction compte (ne marche pas)
        if df['matiere'][u] not in evals :#Si on a pas encore vu cette évaluation alors il ne sagit pas d'une révis
            evals.append(df['matiere'][u])# on stocke l'information pour que la prochaine itération soit une révis:
            evals_masses.append([df['matiere'][u],df['masses'][u],df['date'][u]])#On stocke les infos de cette der
        else :#Si on a déjà vu cette évaluation alors il s'agit d'une révision
            for k in range(len(evals)):#On cherche la bonne évaluation sur laquelle on va faire une révision
                if evals[k]==df['matiere'][u]:#C'est la même compétence, c'est donc la dernière révision
                    id_ant=k#On mémorise sa place pour remplassez la valeur de la dernière révision
                    anterieur = evals_masses[k][1]#On récupère la dernière révision
                    df['revision'][u]=revision(df['masses'][u],anterieur)#On met à jour la valeur de la révision
                    evals_masses[id_ant][1]=df['revision'][u]#On met la valeur de la revisions en mémoire pour rév:
                    df['durée'][u] = datetime.strptime(df['date'][u], '%Y-%m-%d %H:%M:%S')-datetime.strptime(evals_
                    evals_masses[id_ant][2] = df['date'][u]#On stocke la date de la dernière révision
    os.chdir(rep_principal+'\\eleves_masses_rev')#On définie le répertoire de travail comme le dossier eleves_masse
    df.to_csv(nom_eleve+'_masses_rev.csv', sep = ';',index = False) #On créer un tableau csv avec les masses révise
    os.chdir(rep_principal) # on retourne dans le répertoire de travail
```

Enfin cette fonction permet en partant du dataframe des évaluations d'un élève et de son nom (qui ne figure pas dans le dataframe) créer les révisions de ces évaluations et les stocke dans un tableau csv.

On peut remarquer que les révisions sont toujours faites par rapport à la dernière révision (et non pas par rapport à la masse de la dernière évaluation).

Nous pouvons également remarquer que nous avons trier les données en fonction de la date afin de bien avoir les évaluations de la plus ancienne à la plus récente ce qui est important pour la fonction de révision.

3.d. Fonction de prédiction

Fonction de propagation de Masse :

La Fonction de propagation s'appuie sur l'ontologie des compétences ; en effet nous nous servons des évaluations d'une compétence pour prédire les masses d'une compétence requise non évaluée.

Nous utilisons également un paramètre alpha qui permet de déterminer à quelle point nous perdons de la précision à mesure que la distance de 2 compétence est grande du point de vue de l'ontologie.

Exemple si A requiert B à la distance d alors :

Fonction de propagation £ (A, B)

$\forall \ x \epsilon \ P$

$m(a_B) = £(A, B) = (1-\propto)^d \times m(a_A)$

$m(\neg a_B) = £(A, B) = (1-\propto)^d \times m(\neg a_A)$

$m(i_B) = £(A, B) = (1-\propto)^d \times (m(i_A) - 1) + 1$

$m(c_B) = £(A, B) = (1-\propto)^d \times m(c_A)$

Avec :

d : la distance entre la compétence A et la compétence B ;

Maintenant que nous avons une ontologie facilement récupérable on peut facilement implémenter la fonction de propagation de masse :

```
def prop_masses(masse,distance,alpha):#Définition de la fonction de propagation de masse
    new_masse=[0,0,0,0]#On initialise
    distance = int(distance)#On passe distance en integer
    alpha = float(alpha)#On passe alpha en flottant
    new_masse[0]=(1-alpha)**distance*float(masse[0])#On calcule les nouvelles masses selon le modèle
    new_masse[1]=(1-alpha)**distance*float(masse[1])
    new_masse[2]=1+(1-alpha)**distance*float(masse[2])-(1-alpha)**distance
    new_masse[3]=1-float(new_masse[0])-float(new_masse[1])-float(new_masse[2])
    return(new_masse)#On retourne les nouvelles masses
```

Les informations nécessaires pour cette fonction sont :

Pour alpha : donné par l'user.

Pour la masse : trouvée à partir de la masse révisée de toute évaluations se trouvant dans la colonne des compétences des requérants dans l'ontologie.

Pour la distance : également dans le tableau de l'ontologie.

```
def Prop_Masses(Eleves,alpha):#Cette fonction effectue la prédiction en fonction de la dernière révision de l
    df_ont = pd.read_csv(rep_principal+'\\eval_ontologie.csv',sep=';')#On récupère les informations de l'onto
    for eleve in Eleves:#Pour tous les eleves
        preds = []#On stocke les predictions
        #df_eval = pd.read_csv(rep_principal='\\eleves'+eleve.nom+'.csv',sep = ';') #*a passer en non comment
        df_eval = pd.read_csv(rep_principal+'\\eleves_masses_rev\\'+eleve.nom+'_masses_rev.csv',sep = ';')#On
        """try :
            df_masses = df_eval['note'].apply(lambda x:masses_x(x,param))
        except:
            df_masses = df_eval['note'].apply(lambda x:masses_cgt(x,param))"""#*a passer en non commentaire s
        df_masses = df_eval['revision'].apply(lambda x: retab_list_str(x))#On recupère les masses revisées
        for i in range(len(df_eval['note'])):#pour toute les evaluation du dataframe de cet eleve
            for k in range(len(df_ont['distance'])):#pour tous les liens de l'ontologie
                if df_ont['one_who_requires'][k]==df_eval['matiere'][i]:#On regarde si cette compétence en re
                    preds.append([df_ont['one_who_requires'][k],str(df_masses[i]),df_ont['one_that_is_require
                    #On stocke la prediction avec les informations utiles, quelle compétence on a évalué, que
        try :#Si prédiction n'est pas vide (cela est possible)
            ar = np.array(preds)#on stocke le résultat dans un tableau csv dans le dossier eleve_masses_preds
            df_preds = pd.DataFrame(ar, columns = ['Evalué','ses masses évaluées', 'Prédit', 'ses masses préd
            df_preds.to_csv(rep_principal+'\\eleves_masses_preds\\'+eleve.nom+'_masses_preds_'+str(alpha)+'.c
        except:
            pass#Sinon on passe à l'élève suivant
```

Cette fonction part de l'ensemble des élèves et du paramètre alpha pour créer le tableau csv regroupant les masses prédites pour l'ensemble des élèves sous ce format :

EvaluÃ©	ses masses Ã©valuÃ©es	PrÃ©dit	ses masses prÃ©dites	date
Histoire	[0.6, 0, 0.4, 0]	Math	[0.57, 0.0, 0.38, 0.050000000000000044]	17/06/2019 00:00
Histoire	[0.6, 0, 0.4, 0]	Geographie	[0.57, 0.0, 0.38, 0.050000000000000044]	17/06/2019 00:00
Histoire	[0.6, 0, 0.4, 0]	Coloriage	[0.5415, 0.0, 0.361, 0.09750000000000003]	17/06/2019 00:00
Histoire	[0.6, 0, 0.4, 0]	Couleur	[0.5144249999999999, 0.0, 0.34295, 0.1426250(	17/06/2019 00:00

Il montre comment les données sont récupérées dans les différents tableau csv déjà stockés ainsi que dans les structures de données (plus clair directement dans le script, les commentaires sont longs).

L'utilisation de cette fonction nécessite : alpha, Eleves remplit au moins avec des élèves sans évaluations, et les tableaux csv de révision de masse pour les élèves présents dans Eleves.

On pourra remarquer dans l'annexe que la prédiction se fait sur les masses révisées

4.Pistes d'amélioration pour l'outil

Les pistes d'amélioration ne tournent pas tellement autour des fonctions qui était à implémenter mais plutôt sur l'emballage de ces solutions.

Interface :

J'ai en parallèle de ces fonctions implémenter une interface graphique qui n'est ni esthétique ni pratique. Si un élève pouvait améliorer cette dernière il serait bien plus aisé pour un utilisateur de se servir de l'outil sans même avoir à regarder le code.

Du point de vue graphique, je n'ai pas eu le temps de m'intéresser à la taxonomie de l'ontologie (représentation graphique du graphe de l'ontologie).

Système de stockage :

Pour le moment les données sont chargées depuis des tableaux csv et sont gardés en back end jusqu'à ce qu'on ordonne une nouvelle exportation dans un tableau csv.

Cette méthode n'est pas conseillée et rend des fonctionnalités telle que la modification et la suppression de données très compliquée car les données sont difficilement dynamiques.

C'est pourquoi avec plus de temps il aurait été envisagé de passer les données sur une base de données SQL qui en plus d'être une meilleure solution à plus long terme, rendra beaucoup plus simple l'implémentation de la gestion de données complète.

Fonctionnalités manquantes :

En plus de ces changements conséquents, certaines fonctionnalités manquent à la solution, l'ajout d'un compteur pour les révisions, modification et suppression de données, lisibilités des données déjà entrée dans le système.

Ces problèmes seront très facilement réglés par les deux points mentionnés au-dessus.

5.Conclusion

Voici les fonctionnalités qui ont pu être traité durant cette TX :

- Création d'une ontologie
- Création des évaluations des différents étudiants
- Transformation de leurs évaluations en masses de croyance
- Fonction de révision permettant de déterminer le niveau d'acquisition d'une compétence grâce à la fusion de plusieurs évaluations
- Fonction de prédiction qui s'appuie sur les évaluations liées à une compétence afin de déterminé le niveau d'acquisition d'une compétence donnée
- Implémentation d'une interface permettant d'utiliser ces fonctionnalités et dans une moindre mesure de gérer les données (sera sûrement remplacée)

Voici les fonctionnalités qui n'ont pas pu être implémenter ou des pistes d'amélioration de l'outil :

- Modification ou suppression de données mal gérer
- Système de stockage des données imparfait
- Dans le but d'améliorer ces points je préconise l'usage de bases de données type SQL

- Affichage de la taxonomie de l'ontologie
- Compteur de l'itération de révision d'une compétence (régler par le SQL facilement)

- Mise en place d'une interface plus esthétique et fonctionnelle

Mon retour d'expérience est très positif, c'est la première fois que je mène un projet informatique de cette envergure sur un si petit laps de temps et que je gère en grande partie tout seul.

En revanche je n'ai pas su avoir une vue d'ensemble du projet dès le début, j'aurai dû demander plus tôt à Nesrine sa vision finalisée de l'outil afin de structurer mes données de la bonne façon dès le départ et non pas réadapter à chaque nouvelle fonctionnalité demandée.

Je pense que les parties les plus importantes de ce projet, à savoir les fonctions back end qui correspondent au modèle de Nesrine ont été implémentées dans leur grande majorité ; et que le reste (gestion des données, interface utilisateur efficace et attractive) pourrait être le sujet d'une autre TX.

Je reste à disposition encore quelques temps de Nesrine pour corriger certains détails de la solution et pour répondre aux questions qu'elle pourrait se poser après lecture des scripts et de leurs commentaires.