Parcours Data Scientist, projet n°6, Rapport

Christophe Coudé, 2017

Stack Overflow est un site célèbre de question-réponses liées au développement informatique. Pour poser une question sur ce site, il faut entrer plusieurs tags de manière à retrouver facilement la question par la suite. Afin d'aider les nouveaux utilisateurs qui ne savent pas forcément choisir des tags judicieux, il serait judicieux de leur suggérer quelques tags relatifs à la question posée.
Les données sont accessibles via un outil d'export de données mis à disposition par Stack Overflow: "stackexchange explorer" :
https://data.stackexchange.com/stackoverflow/query/new
Ce projet a donc pour but de développer un système de suggestion de tag pour ce site.

Exploration

1- Choix des tables et champs de la base

- La base de donnée contient une trentaine de tables dont Posts, Users, Comments, Badges, etc.
- Je ne me suis servi que de la table Posts avec les champs Title et Tags.
- Il est possible d'utiliser d'autres tables, je l'indique dans les pistes d'amélioration à la fin du rapport.
- Pour des raisons des limites de mon modeste ordinateur (temps d'exécution trop long, memory error ou arrêt automatique des méthodes par Juyter par crainte de crashage du kernel), je n'ai récupéré de la base que les 500 000 premiers enregistrements de la table Posts.

Requête utilisée :
select Id, Title, Tags from Posts where Title is not null and Tags is not null order by Id;
Puis export des 500 000 enregistrements proposés par le site.

In [7]:
```
tagsAll = pd.read_csv("tags.csv")
tagsAll.head(3)
```

Out[7]:

	Id	Title	Tags
0	4	While applying opacity to a form should we use...	<c#><winforms><type-conversion><decimal><opacity>
1	6	Percentage width child element in absolutely p...	<html><css><css3><internet-explorer-7>
2	9	Calculate age in C#	<c#><.net><datetime>

Les différents champs adoptent des formats différents (même Posts.Body) qu'il faut gérer. J'ai récupéré les deux champs essentiels, Posts.Title (qui contient le titre de la question posée par l'utilisateur) et Posts.Tags les tags qu'il a renseignés.

2- Importance des stop_words

Quand on cherche à déterminer quels sont les mots les plus proches (les plus liés) d'un autre dans une phrase, on tombe souvent sur des mots de liaison, des mots qui n'ont pas vraiment une importance prépondérante sémantiquement dans la phrase.
Par exemple avec les deux phrases :

- The python is a snake
- The pythone is not a snake

In [9]:
```
getNearestWordsFromPythonWithStopWords()
```

```
Mots les plus proches de 'python' :  [('not', 0.44353973865509033), ('is', 0.3584551811218261
7), ('snake', 0.2383769452571869), ('pythone', 0.034875497221946716), ('a', -0.062948524951934
81), ('The', -0.367592990398407)]
```

Si on n'enlève pas les stop_words, on trouve des mots peu liés.

In [10]:
```
getNearestWordsFromPythonWithoutStopWords()
```

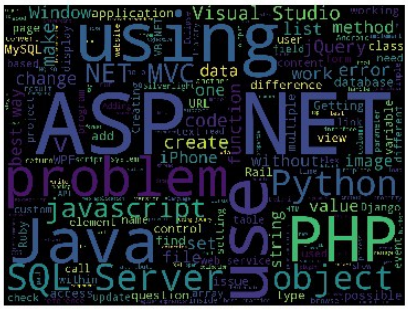
```
Mots les plus proches de 'python' :  [('snake', 0.2390412837266922), ('pythone', 0.0351759195327
7588)]
```

Sans ces mots, on se rapproche d'un choix de mots bien plus proche de l'idée principale de la phrase.
Il est donc important d'appliquer des "stop_words" aux phrases afin d'en extraire la quintessence si l'on puit dire.

3- Wordcloud des mots des questions et des tags

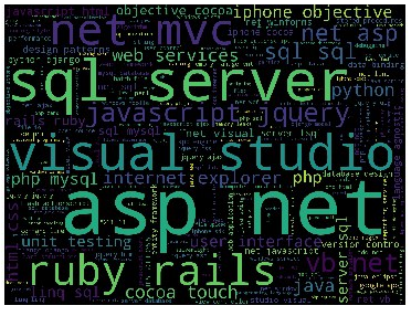
Pour se faire une première idée des mots les plus fréquents dans les questions, on peut dessiner un wordcloud:

In [16]:
```
drawTitleWordCloud(tagsAll)
```



De la même façon avec les tags les plus fréquents:

In [18]:
```
drawTagsWordCloud(tagsAll)
```



On retrouve parfois les mêmes mots, ce qui est normal car souvent les tags sont présents dans les titres des questions.

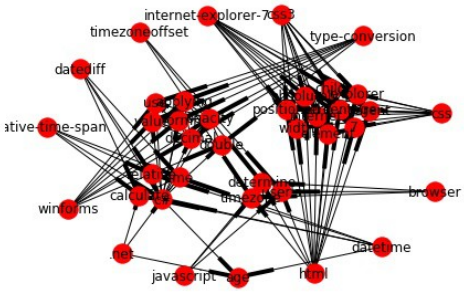
4- Représentation des liens entre tags

On peut tenter de représenter les liens entre les mots des titres des questions et les tags:

In [20]:
```
drawGraphTagsLinks(tagsAll)
```

```
Liste des tags :  {'css', 'internet-explorer-7', 'relative-time-span', 'css3', 'timezone', 'browser', 'html', 'datediff',
'.net', 'decimal', 'javascript', 'opacity', 'winforms', 'timezoneoffset', 'c#', 'datetime', 'type-conversion', 'time'}

Liste des mots :  {'age', 'parent', 'percentage', '7', 'c#', 'relative', 'use', 'value', 'element', 'time', 'explorer', 'ab
solutely', 'child', 'internet', 'timezone', 'width', 'calculate', 'opacity', "user's", 'positioned', 'form', 'decimal', 'do
uble', 'determine', 'applying'}
```



Au-delà de 5 questions, le schéma devient illisible mais on peut déjà observer que certains mots et tags sont plus fortement liés ensemble et se regroupent dans des nuages denses.

On peut alors se demander si certains tags ne sont pas corrélés ensemble, si lorsqu'un tag apparaît, un autre tag n'est jamais loin, cela pourrait être utile pour lier ces tags dans les propositions faites aux utilisateurs:

5- Corrélation des tags

Comme il y a énormément de tags, on se limitera aux 10 premières questions:
Rien qu'avec ces 10 questions, on obtient un vocabulaire important illustré ci-dessous

In [25]:
```
dfVocab = createVocabFromTagsDataFrame(tagsAll, 10, True)
dfVocab.head()
```

```
['.net', '.net-3.5', 'algorithm', 'binary-data', 'browser', 'c#', 'css', 'css3', 'data-storage', 'database', 'datediff', 'd
atetime', 'decimal', 'html', 'internet-explorer-7', 'javascript', 'language-agnostic', 'linq', 'math', 'mysql', 'opacity',
'performance', 'pi', 'relative-time-span', 'time', 'timezone', 'timezoneoffset', 'triggers', 'type-conversion', 'unix', 'we
b-services', 'winforms']
```

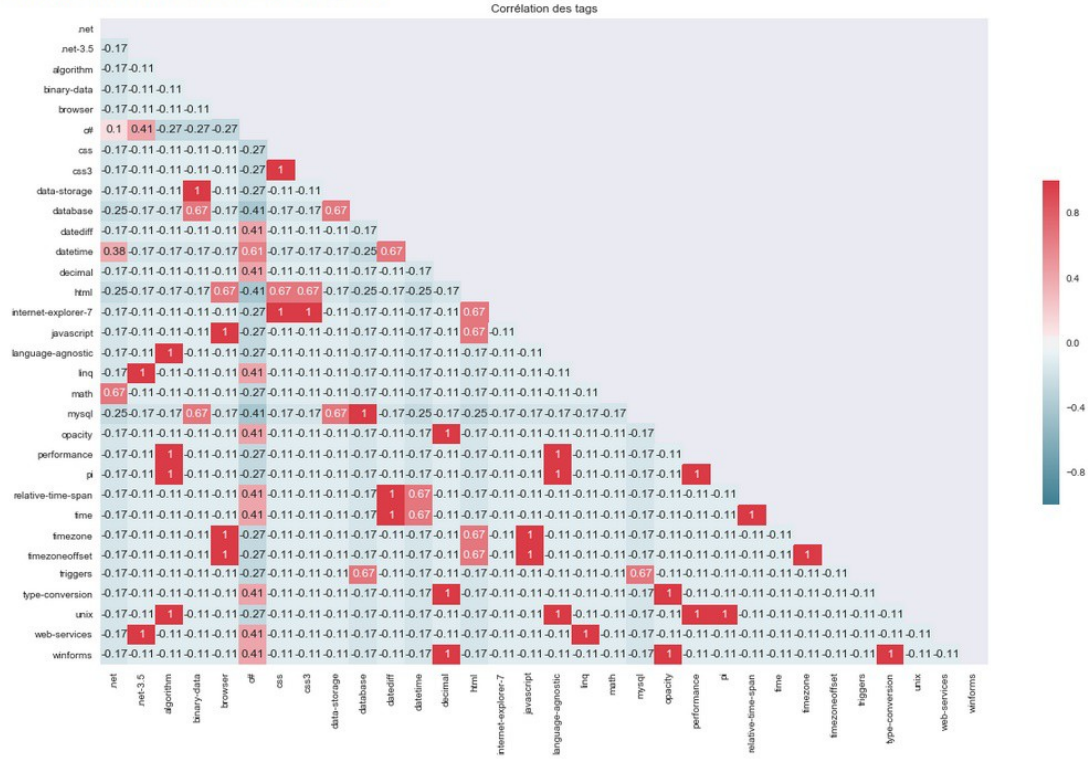
Out[25]:

	.net	.net-3.5	algorithm	binary-data	browser	c#	css	css3	data-storage	database	...	pi	relative-time-span	time	timezone	timezoneoffset	triggers	conv
0	0	0	0	0	0	1	0	0	0	0	...	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	1	0	0	...	0	0	0	0	0	0	0
2	1	0	0	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	...	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	...	0	0	0	1	1	0	0

5 rows × 32 columns

In [24]:
```
heatmap(dfVocab, "pearson", "Corrélation des tags", True)
```

```
<matplotlib.figure.Figure at 0x21a86e48c50>
```



Toutefois, les fortes valeurs sont trompeuses, comme nous l'avons vu dans le notebook d'exploration, cela est dû au fait qu'il y a très peu de questions. Lorsqu'on prend l'ensemble des questions, le valeur de corrélation baisse beaucoup, comme dans l'exemple suivant qui passe de 1 à 0.008.

In [27]:
```
print(corrBetweenTags(tagsAll, 'pi', 'performance', "pearson"))
```

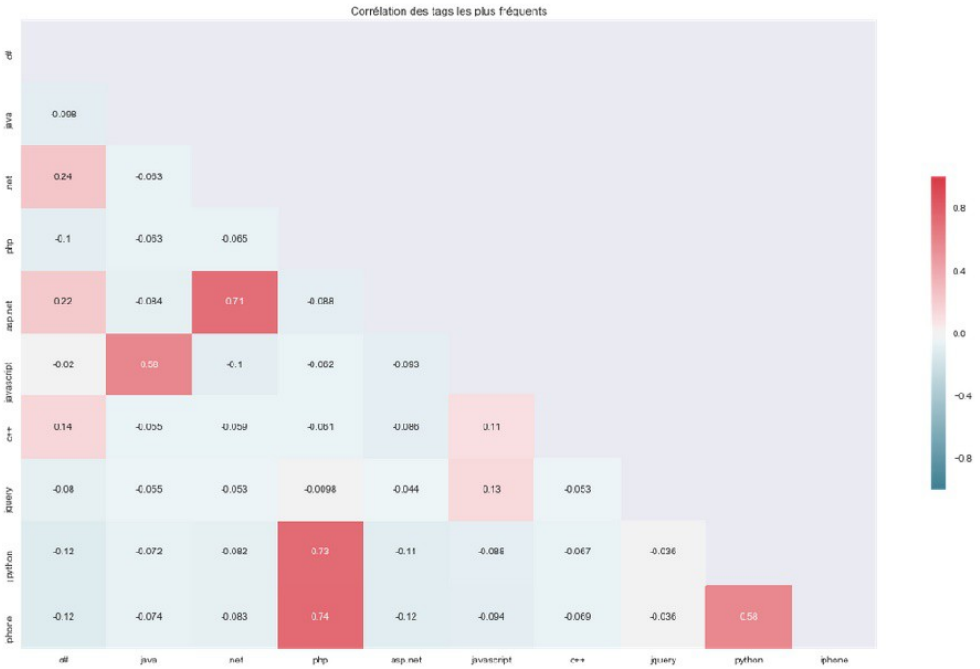
```
0.00823129785887
```

Lorsqu'on prend les tags les plus fréquents dont la liste se situe juste en-dessous, on ne trouve également pas de corrélations fortes.

In [31]:
```
listeTags = createTagHierarchie(tagsAll)
dfMostCommonTags = createMostCommonTagsDataFrame(listeTags, 10, tagsAll, True)
```

```
['c#', 'java', '.net', 'php', 'asp.net', 'javascript', 'c++', 'jquery', 'python', 'iphone']
```

In [32]:
```
heatmap(dfMostCommonTags, "pearson", "Corrélation des tags les plus fréquents", True)
```



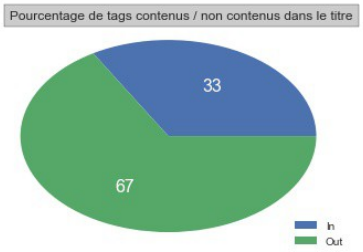
Mais même si on n'atteint pas les sommets de la corrélations, il pourrait être intéressant de voir pour les corrélations les plus élevées, si le fait de lier les tags dans les suggestions de tags n'améliorerait pas les modèles. Je ne l'ai pas fait, mais cela pourrait être une piste intéressante d'amélioration.

6- Pourcentage de tags présents dans les titres et titres sans aucun tags

In [40]:
```
nbTagInTitle = 0
nbTags = 0
def tagInTitle(row):
    global nbTagInTitle
    global nbTags
    tags = row['Tags'].lower()
    listeRowTags = tags[1:-1].split("><")
    listeRowWords = getGreatWordsFromSentence(row['Title'].lower())
    for tag in listeRowTags:
        nbTags += 1
        if tag in listeRowWords:
            nbTagInTitle += 1
    return row
n = 50000
tags10 = tagsAll.iloc[:n,:]
tags10.apply(tagInTitle, axis = 1)
pcTagsInTitle = 100 / nbTags * nbTagInTitle
print("Pourcentage de tags contenus dans les titres : {0}".format(pcTagsInTitle))
```

```
Pourcentage de tags contenus dans les titres : 33.48633624451148
```

In [41]:
```
drawCamembertOfPcTagsInTitle(pcTagsInTitle)
```

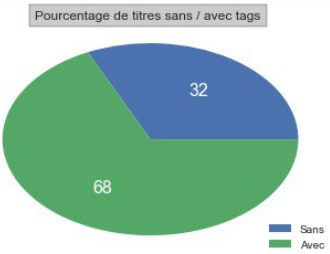


On observe que sur les 50 000 premières questions, environ un tiers des tags sont déjà contenus dans les titres.

In [43]:
```
nbTitleWithoutTags = 0
def tagInTitle(row):
    nbTagInTitle = 0
    global nbTitleWithoutTags
    tags = row['Tags'].lower()
    listeRowTags = tags[1:-1].split("><")
    listeRowWords = getGreatWordsFromSentence(row['Title'].lower())
    for tag in listeRowTags:
        if tag in listeRowWords:
            nbTagInTitle += 1
    if nbTagInTitle == 0:
        nbTitleWithoutTags += 1
    return row
n = 50000
tags10 = tagsAll.iloc[:n,:]
tags10.apply(tagInTitle, axis = 1)
pcTitleWithoutTags = 100 / n * nbTitleWithoutTags
print("Pourcentage de titres sans tags: {0}".format(pcTitleWithoutTags))
```

```
Pourcentage de titres sans tags: 31.944
```

In [44]:
```
drawCamembertOfPcTitlesWithoutTags(pcTitleWithoutTags)
```



Ainsi, à peu près un tiers des titres ne possèdent aucun tag. Cela signifie qu'il peut être très difficile pour cette catégorie de titres de trouver les tags.

Modèles

Nous pouvons envisager deux catégories de méthodes, des méthodes supervisées puis des non supervisées

Modèles

Nous pouvons envisager deux catégories de méthodes, des méthodes supervisées puis des non supervisées

A- Apprentissage supervisé

7- Recherche des tags pertinants à partir du champ Title correspondant aux questions

Pour ce premier modèle, le but est de lier aux mots d'un titre d'une nouvelle question, les tags des "anciennes" questions d'entraînement contenant ces mots :

- pour un nouveau titre, récupérer les mots
- pour chaque nouveau mot, prendre toutes les questions d'entraînement possédant ce mot
- pour chacune de ces anciennes questions, collecter tous les tags
- rassembler ces tags liés au mot de la nouvelle question
- réunir enfin tous ces tags qui sont liés à au moins un mot de la nouvelle question
- proposer les tags les plus fréquents

Tests avec proposition de 15 tags pour:

- 5000 questions d'entraînement et 100 questions de test (d'indices supérieurs à ceux de l'entraînement)
- 5000 questions d'entraînement et 1000 questions de test
- 10000 questions d'entraînement et 2000 questions de test

In [49]:
```
time1 = time.time()
print("score : ", scoreModel(5000, 100, 15, tagsAll, False))
print("temps : ", time.time()-time1)
```

```
score :  51.8
temps :  23.270901918411255
```

In [36]:
```
time1 = time.time()
print("score : ", scoreModel(5000, 1000, 15, tagsAll, False))
print("temps : ", time.time()-time1)
```

```
score :  53.4233333333333
temps :  217.12663531303406
```

In [37]:
```
time1 = time.time()
print("score : ", scoreModel(10000, 2000, 15, tagsAll, False))
print("temps : ", time.time()-time1)
```

```
score :  49.56666666666664
temps :  487.95096349716187
```

On obtient déjà des scores intéressants avec près de la moitié des tags prédits.

Utiliser des questions de test déjà existantes afin de calculer le score du modèle est pratique, mais qu'est-ce que cela donne avec des questions qui n'existent pas encore ?

Prenons une question existante puis modifions-là afin de la rendre inédite, tout en gardant les mots les plus importants:

In [41]:
```
# question déjà existante index n°5003, résultat attendu : <c++><c><memory-leaks>
sentence = "How do you detect/avoid Memory leaks in your (Unmanaged) code?"
print("Tags attendus : ", tagsAll['Tags'][5003])
suggestGeneratedTagsFromSentence(tagsAll, sentence)
```

```
Tags attendus :  <c++><c><memory-leaks>
[('c#', 3608), ('c++', 2128), ('.net', 1995), ('java', 1985), ('php', 1702), ('iphone', 1533), ('javascript', 1394), ('asp.
net', 1300), ('xcode', 1111), ('python', 1028), ('c', 1013), ('memory', 937), ('memory-management', 885), ('memory-leaks',
840), ('unicode', 755)]
```

In [42]:
```
sentence = "Causes of Memory leaks though code?"
suggestGeneratedTagsFromSentence(tagsAll, sentence)
```

```
[('c#', 3008), ('c++', 1798), ('java', 1709), ('.net', 1670), ('php', 1498), ('iphone', 1347), ('asp.net', 1171), ('xcode',
1101), ('javascript', 967), ('memory', 923), ('c', 915), ('python', 914), ('memory-management', 863), ('memory-leaks', 80
1), ('unicode', 739)]
```

On observe bien que malgré que la question a été changée, le fait de garder les mots les plus importants ('memory', 'leaks', 'code') a permis de retrouver les bons tags.

8- Recherche des tags pertinants à partir des plus proches voisins du champ Title, méthodes NearestNeighbors, CountVectorizer

Autre méthode, ne pas récupérer pelle-melle tous les tags historiques liés à tous les mots d'une nouvelle question, mais d'abord rechercher dans l'historique des titres de questions eux-même, les plus proches voisins du nouveau titre:

- générer un vocabulaire à partir des questions d'entraînement
- créer la matrice des mots des questions d'entraînement relativement à ce vocabulaire
- projeter le titre de la nouvelle question sur ce vocabulaire afin d'en retirer un vecteur
- récupérer les plus proches voisins de ce vecteur dans la matrice
- collecter (en manque de synonymes...) les tags de ces titres voisins
- proposer les plus fréquents

Exemples avec 500 questions d'entraînement pour les 15 tags proposés:
(pour l'illustration je prends des données d'entraînement, mais pour le score, je prends des données de tests)

In [46]:
```
trainNbLine = 500
nbTagsAProposer = 15
df = tagsAll
listCol = ['Title']

cv, matrix = createVocMatrixFromTrainingPhrases(df, trainNbLine, ['Title'])
for index in range(3):
    print("\nAvec {0} lignes, score pour ligne n° {1} : {2} %\n".format(trainNbLine, index, getScore(index, nbTagsAProposer,
df, trainNbLine, cv, matrix, listCol, True)))
```

```
indices =  [316 476 272 307 430  82 210  52  98  86 111 119 150   8 116]
existing :  ['c#', 'winforms', 'type-conversion', 'decimal', 'opacity']
generated :  ['c#', 'ruby', 'language-agnostic', 'unix', 'recursion', 'computer-science', 'openid', 'sql', 'sql-server', 'g
roup-by', '.net', 'colors', 'rgb', 'hsl', 'messaging']

Avec 500 lignes, score pour ligne n° 0 : 20.0 %

indices =  [355  86 422 111 185 316 252  96 110 243 423 157  50 322 195]
existing :  ['html', 'css', 'css3', 'internet-explorer-7']
generated :  ['glossary', 'javascript', 'internet-explorer', 'debugging', 'php', 'encryption', 'passwords', 'algorithm', 's
orting', 'graph-algorithm', 'directed-graph', 'html', 'css', 'text', 'statistics']

Avec 500 lignes, score pour ligne n° 1 : 50.0 %

indices =  [185  86 252 111   3 345 322  96 110 422 195 243  50 316 157]
existing :  ['c#', '.net', 'datetime']
generated :  ['c#', 'algorithm', 'glossary', 'statistics', 'spss', 'php', 'encryption', 'passwords', 'ajax', 'progressive-e
nhancement', 'html', 'css', 'text', 'datetime', 'time']

Avec 500 lignes, score pour ligne n° 2 : 66.66666666666667 %
```

On voit par exemple pour le troixième enregistrement, on propose 2 des 3 tags ('c#' et 'datetime')

In [47]:
```
trainNbLine = 5000
testNbLine = 100
nbTagsAProposer = 15
df = tagsAll
listCol = ['Title']

print("début = ", time.time())
print("\nAvec {0} lignes et {1} tags proposés, score pour les {2} premières lignes de test hors entraînement: {3} %\n".format
(trainNbLine, nbTagsAProposer, testNbLine, getScoreModel(trainNbLine, testNbLine, nbTagsAProposer, df, listCol)))
print("fin = ", time.time())
```

```
début =  1507586731.0960424

Avec 5000 lignes et 15 tags proposés, score pour les 100 premières lignes de test hors entraînement: 33.99999999999999 %

fin =  1507586905.9571187
```

Pour bien comparer ce modèle avec le modèle précédent, il faudrait faire un test avec 1000 questions à tester (manque de temps). On peut toutefois comparer avec 5000 questions d'entraînement et 100 questions de test, et avec 34%, ce modèle donne de moins bons résultats que le premier modèle (51.8%).

9- Recherche des tags pertinants à partir des plus proches voisins des champs Title et Body, puis Title, Tags et Body

Intégrer les tags dans le titre permet au niveau des données d'entraînement de lier les autres mots des titres aux tags adéquats. On peut donc essayer la méthode des plus proches voisins sur les champs Title et Body. C'est-à-dire qu'au lieu de faire un vocabulaire uniquement sur les mots des titres, on lui adjoint les tags.
Il n'y a pas de risque de leakage puisque lorsqu'on teste une nouvelle question, on n'intègre pas les tags, on se contente des informations qu'on a, c'est-à-dire les mots du nouveau titre.

In [53]:
```
trainNbLine = 5000
testNbLine = 100
nbTagsAProposer = 15
df = tagsAll
listCol = ['Title', 'Tags']

time1 = time.time()
print("\nAvec {0} lignes et {1} tags proposés, score pour les {2} premières lignes de test hors entraînement: {3} %\n".format
(trainNbLine, nbTagsAProposer, testNbLine, getScoreModel(trainNbLine, testNbLine, nbTagsAProposer, df, listCol)))
print("Temps = ", time.time() - time1)
```

```
Avec 5000 lignes et 15 tags proposés, score pour les 100 premières lignes de test hors entraînement: 65.53333333333333 %

Temps =  208.89473509788513
```

Il s'avère qu'avec cette méthode, et toujours avec la réserve que les tests ne sont pas faits sur de gros volumes, on obtient un meilleur résultat (65.53% contre 51.8%).

Pour le champ Body, requête utilisée :
select Id, Title, Tags, Body from Posts where Title is not null and Tags is not null and Body is not null and id < 87422 order by Id;
Pour éviter de télécharger un trop gros volume, j'ai limité l'export à 10 000 enregistrements.

12- Occurences sur un des modèles

Même si le modèle avec l'utilisation seule des tags liés aux mots des questions n'a pas été le meilleur, on peut l'utiliser pour illustrer l'idée de faire participer l'utilisateur. A chaque itération :
1- on propose à l'utilisateur une liste de tags (comme actuellement)
2- l'utilisateur choisit le tag qui lui semble le plus adéquat dans cette liste (si l'utilisateur n'en choisit pas, on arrête l'algorithme)
3- on incorpore ce tag dans la question afin de donner plus de poids aux autres tags qui seraient liés à ce tag, et de lier les mots de la question à ce tag
4- on passe à l'itération suivante

L'exemple choisi dans le notebook des modèles a montré que cette méthode pouvait se révéler fructueuse, puisqu'elle a permis de récupérer un tag supplémentaire. Il pourrait alors être intéressant de voir à plus grande échelle. Toutefois, comme cette méthode incorpore un choix humain, cela est difficile. Pour se faire une idée, on peut cependant essayer de faire un choix automatique en proposant un panel de tags adéquats comme reprendre un par un les tags réels s'ils font partie des tags proposés et sinon prendre le tag le plus fréquent; afin de voir si on fait "émerger" les tags restants comme ci-dessous:

In [75]:
```
tags5000 = tagsAll.iloc[:5000, :]
listTagsChoisis = automaticTagsGeneratedList(tagsAll, 5000, 2, 15, True)
print("\n Liste des tag finalement choisis = ", listTagsChoisis)
```

```
Title =  calculate age in c#

Liste des tags réels =  ['c#', '.net', 'datetime']

Itération avec le tag :  c#
dicoOrdered =  [('c#', 249), ('.net', 99), ('asp.net', 44), ('java', 21), ('c++', 19), ('image', 19), ('html', 18), ('datab
ase', 17), ('sql-server', 16), ('windows', 16), ('sql', 15), ('winforms', 15), ('language-agnostic', 14), ('performance', 1
3), ('php', 13)]
tagChoisi =  c#

Itération avec le tag :  .net
dicoOrdered =  [('c#', 249), ('.net', 99), ('asp.net', 44), ('java', 21), ('c++', 19), ('image', 19), ('html', 18), ('datab
ase', 17), ('sql-server', 16), ('windows', 16), ('sql', 15), ('winforms', 15), ('language-agnostic', 14), ('performance', 1
3), ('php', 13)]
tagChoisi =  .net

Itération avec le tag :  datetime
dicoOrdered =  [('c#', 355), ('.net', 296), ('asp.net', 207), ('asp.net-mvc', 61), ('vb.net', 45), ('windows', 31), ('sql-s
erver', 30), ('java', 28), ('javascript', 27), ('visual-studio', 27), ('.net-3.5', 25), ('c++', 25), ('database', 24), ('ht
ml', 23), ('sql', 23)]
tagChoisi =  asp.net

 Liste des tag finalement choisis =  ['c#', '.net', 'asp.net']
```

On voit donc que dans cet exemple que la liste des tags automatiquement choisis change 'datetime' pour 'asp.net'. Toutefois, même en incluant c# et .net dans le titre, 'datetime' se situe très loin, au-delà de la 70ème place des tags proposés avec un nombre d'occurence quasiment pas amélioré (4 au lieu de 3). On peut en conclure que sur les 5000 premiers enregistrements, 'datetime' est rarement associé aux termes 'calculate', 'age' et 'c#' (ni même avec '.net'). C'est donc un "cas rare" difficile à faire remonter.

On peut toutefois calculer le score plus général et voir si cela améliore le modèle initial :

In [76]:
```
score = scoreAutomaticModel(5000, 1000, 15, tagsAll, False)
print("Score global : ", score)
```

```
Score global :  55.09999999999996
```

In [ ]:
```
# print(scoreModel(5000, 1000, 15, tagsAll, False)) avait donné 53.42% (voir plus haut)
```

Commentaires :
Sur les 1000 premières questions, on obtient quand même une légère amélioration du score qui n'est pas négligeable :
53.42 % --> 55.10 %

13- Tunning sur les paramètres

On pourrait jouer sur différents paramètres pour obtenir de meilleurs résultats, comme le nombre de plus proches voisins ou le nombre de tags proposé, tout cela étant une question de compromis entre temps d'exécution et performance des modèles.
La méthode sklearn.model_selection.GridSearchCV pourrait être intéressante à utiliser à cet égard.

B- Apprentissage non supervisé

14- Modèle uniquement basé sur les titres

Si on ne veut pas intégrer les tags historiques dans le modèle, on peut par exemple rechercher les mots les plus fréquents de l'historique (ou du moins des données d'entraînement) des titres.
Toutefois, afin de s'adapter à toute nouvelle question et ne pas proposer toujours les mêmes mots, on ne fournira pas les mots les plus fréquents de tout l'historique (ou des données d'entraînement), mais seulement les mots les plus fréquents des titres de l'historique qui contiennent au moins un mot du

In [48]:
```
print(suggestWordsFromTitleHistoric(tagsAll['Title'][101], tagsAll, False)[:15])
```

```
[('list', 5921), ('lists', 727), ('using', 629), ('python', 536), ('-', 424), ('c#', 398), ('items', 396), ('item', 377),
('way', 374), ('jquery', 304), ('sorted', 280), ('listbox', 279), ('efficiently', 273), ('asp.net', 269), ('data', 266)]
```

Comme on peut le voir ci-dessus, on retrouve des mots issus de la même racine, comme list / lists, item / items. Il serait alors utile de les rassembler. Pour ce faire, on peut utiliser un stemmer:

12- Occurences sur un des modèles

Même si le modèle avec l'utilisation seule des tags liés aux mots des questions n'a pas été le meilleur, on peut l'utiliser pour illustrer l'idée de faire participer l'utilisateur. A chaque itération :
1- on propose à l'utilisateur une liste de tags (comme actuellement)
2- l'utilisateur choisit le tag qui lui semble le plus adéquat dans cette liste (si l'utilisateur n'en choisit pas, on arrête l'algorithme)
3- on incorpore ce tag dans la question afin de donner plus de poids aux autres tags qui seraient liés à ce tag, et de lier les mots de la question à ce tag
4- on passe à l'itération suivante

L'exemple choisi dans le notebook des modèles a montré que cette méthode pouvait se révéler fructueuse, puisqu'elle a permis de récupérer un tag supplémentaire. Il pourrait alors être intéressant de voir à plus grande échelle. Toutefois, comme cette méthode incorpore un choix humain, cela est difficile. Pour se faire une idée, on peut cependant essayer de faire un choix automatique en proposant un panel de tags adéquats comme reprendre un par un les tags réels s'ils font partie des tags proposés et sinon prendre le tag le plus fréquent; afin de voir si on fait "émerger" les tags restants comme ci-dessous:

In [75]:
```
tags5000 = tagsAll.iloc[:5000, :]
listTagsChoisis = automaticTagsGeneratedList(tagsAll, 5000, 2, 15, True)
print("\n Liste des tag finalement choisis = ", listTagsChoisis)
```

```
Title =  calculate age in c#

Liste des tags réels =  ['c#', '.net', 'datetime']

Itération avec le tag :  c#
dicoOrdered =  [('c#', 249), ('.net', 99), ('asp.net', 44), ('java', 21), ('c++', 19), ('image', 19), ('html', 18), ('datab
ase', 17), ('sql-server', 16), ('windows', 16), ('sql', 15), ('winforms', 15), ('language-agnostic', 14), ('performance', 1
3), ('php', 13)]
tagChoisi =  c#

Itération avec le tag :  .net
dicoOrdered =  [('c#', 249), ('.net', 99), ('asp.net', 44), ('java', 21), ('c++', 19), ('image', 19), ('html', 18), ('datab
ase', 17), ('sql-server', 16), ('windows', 16), ('sql', 15), ('winforms', 15), ('language-agnostic', 14), ('performance', 1
3), ('php', 13)]
tagChoisi =  .net

Itération avec le tag :  datetime
dicoOrdered =  [('c#', 355), ('.net', 296), ('asp.net', 207), ('asp.net-mvc', 61), ('vb.net', 45), ('windows', 31), ('sql-s
erver', 30), ('java', 28), ('javascript', 27), ('visual-studio', 27), ('.net-3.5', 25), ('c++', 25), ('database', 24), ('ht
ml', 23), ('sql', 23)]
tagChoisi =  asp.net

 Liste des tag finalement choisis =  ['c#', '.net', 'asp.net']
```

On voit donc que dans cet exemple que la liste des tags automatiquement choisis change 'datetime' pour 'asp.net'. Toutefois, même en incluant c# et .net dans le titre, 'datetime' se situe très loin, au-delà de la 70ème place des tags proposés avec un nombre d'occurence quasiment pas amélioré (4 au lieu de 3). On peut en conclure que sur les 5000 premiers enregistrements, 'datetime' est rarement associé aux termes 'calculate', 'age' et 'c#' (ni même avec '.net'). C'est donc un "cas rare" difficile à faire remonter.

On peut toutefois calculer le score plus général et voir si cela améliore le modèle initial :

In [76]:
```
score = scoreAutomaticModel(5000, 1000, 15, tagsAll, False)
print("Score global : ", score)
```

```
Score global :  55.09999999999996
```

In [ ]:
```
# print(scoreModel(5000, 1000, 15, tagsAll, False)) avait donné 53.42% (voir plus haut)
```

Commentaires :
Sur les 1000 premières questions, on obtient quand même une légère amélioration du score qui n'est pas négligeable :
53.42 % --> 55.10 %

13- Tunning sur les paramètres

On pourrait jouer sur différents paramètres pour obtenir de meilleurs résultats, comme le nombre de plus proches voisins ou le nombre de tags proposé, tout cela étant une question de compromis entre temps d'exécution et performance des modèles.
La méthode sklearn.model_selection.GridSearchCV pourrait être intéressante à utiliser à cet égard.

B- Apprentissage non supervisé

14- Modèle uniquement basé sur les titres

Si on ne veut pas intégrer les tags historiques dans le modèle, on peut par exemple rechercher les mots les plus fréquents de l'historique (ou du moins des données d'entraînement) des titres.
Toutefois, afin de s'adapter à toute nouvelle question et ne pas proposer toujours les mêmes mots, on ne fournira pas les mots les plus fréquents de tout l'historique (ou des données d'entraînement), mais seulement les mots les plus fréquents des titres de l'historique qui contiennent au moins un mot du

In [48]:
```
print(suggestWordsFromTitleHistoric(tagsAll['Title'][101], tagsAll, False)[:15])
```

```
[('list', 5921), ('lists', 727), ('using', 629), ('python', 536), ('-', 424), ('c#', 398), ('items', 396), ('item', 377),
('way', 374), ('jquery', 304), ('sorted', 280), ('listbox', 279), ('efficiently', 273), ('asp.net', 269), ('data', 266)]
```

Comme on peut le voir ci-dessus, on retrouve des mots issus de la même racine, comme list / lists, item / items. Il serait alors utile de les rassembler. Pour ce faire, on peut utiliser un stemmer:

In [49]:
```
print(suggestWordsFromTitleHistoric(tagsAll['Title'][101], tagsAll, True)[:15])
```

```
[('list', 6739), ('use', 898), ('item', 747), ('python', 569), ('sort', 474), ('file', 446), ('-', 424), ('valu', 421), ('s
elect', 408), ('object', 399), ('c#', 398), ('listen', 385), ('way', 379), ('creat', 310), ('listbox', 295)]
```

- On vérifie bien que les fréquences de list et lists se sont additionné, ainsi que des fréquences d'autres dérivés de list puisqu'il y a un reliquat à cette addition dans la nouvelle fréquence.
- Ce modèle propose donc ici les 15 mots les plus fréquents que l'on trouve dans les titres d'apprentissage où au moins un mot du titre de test est présent.
- (Remarque : pour cet exemple illustrant l'utilisation du stemming, je n'ai pas pris un titre de test mais un titre d'entraînement, par contre, dans ce qui suit, la distinction a bien été faite entre entraînement et test en ce qui concerne le calcul du score.)

Illustration du modèle avec 10 000 questions d'entraînement et la question d'index 10 000 pour le test:

In [53]:
```
getScoreSheerTitle(tagsAll, 10000, 1, True)
```

```
Exemple :
Titre : get contacts from email account
Liste des tags : ['email', 'extract']
Tags proposés : ['email', 'send', 'account', 'use', 'user', 'way', 'server', 'address', 'best', '.net', 'php', 'sql', 'asp.
net', 'client', 'creat']
Score : 50.0 %
```

Out[53]:
```
50.0
```

A cause d'un problème de MemoyError, je n'ai fait un test que sur 10 000 questions d'entraînement et 1000 questions de test:

In [54]:
```
# MemoyError sur (50000, 1000)
getScoreSheerTitle(tagsAll, 10000, 1000, False)
```

Out[54]:
```
22.381666666666643
```

Sur ce test, on obtient un score de 22.38 %

15- LDA (Latent Dirichlet Allocation)

- Le Latent Dirichlet Allocation est un modèle probabiliste génératif de corpus.
- L'idée consiste à considérer les documents (dans notre contexte ce sont les titres des questions) comme des mélanges aléatoires sur des topics sous-jacents, où chaque topic est caractérisé par une une distribution sur les mots.
- Dans notre contexte, les topics pourraient être exprimés par des mots servant de tags.

In [58]:
```
scoreGlobal = getScoreModelLDA(tagsAll, 1000, 1000, 100)
```

In [59]:
```
print(scoreGlobal)
```

```
13.799999999999997
```

Le score pour le modèle LDA avec 1000 questions d'entraînement, 1000 thèmes et 100 questions de test est de : 13.80 %

16- API, affichage des résultats en format JSON à partir d'un Web Service REST sur AWS

L'API consiste en un service REST / Flask hébergé sur AWS (Amazon Web Service) appelé par HTTP et qui rend le résultat sous format Json.

Sur AWS, dans le fichier OCRDSP6app.py, j'ai fait un entraînement sur les 1000 premières questions.
L'exemple suivant est une question d'index 1072 (donc hors de l'entraînement du modèle):

http://127.0.0.1:5000/projet6/tags/15:LINQ%20query%20on%20a%20DataTable (dans l'url, on peut mettre des espaces)


```
{
"Initial question": "LINQ query on a DataTable",
"Suggested tags list": [
    {
        "tag 1": "linq",
        "tag 10": "left-join",
        "tag 11": "database",
        "tag 12": "loops",
        "tag 13": "dataset",
        "tag 14": ".net-2.0",
        "tag 15": "orm",
        "tag 2": "c#",
        "tag 3": ".net-3.5",
        "tag 4": "linq-to-sql",
        "tag 5": ".net",
        "tag 6": "sql",
        "tag 7": "performance",
        "tag 8": "sql-server",
        "tag 9": "mysql"
    }
]
}
```

Pour cette question, les tags réellement attendus sont : <\c#><.net><\linq><\datatable><.net-3.5> (sans les \, petit problème d'affichage markdown)
On voit donc qu'on les obtient tous les cinq en positions 1, 2, 3, 5 et 11.
(C'est bien sûr un cas presque "idéal", mais comme on l'a vu, avec ce modèle on obtient de bons résultats.)

Autre exemple avec la question d'index 1005 :
http://127.0.0.1:5000/projet6/tags/15:How%20do%20I%20bind%20a%20regular%20expression%20to%20a%20key%20combination%20in%20emacs?


```
{
"Initial question": "How do I bind a regular expression to a key combination in emacs",
"Suggested tags list": [
    {
        "tag 1": "regex",
        "tag 10": "sql-server",
        "tag 11": "html",
        "tag 12": "editor",
        "tag 13": "collections",
        "tag 14": "scheme",
        "tag 15": "http",
        "tag 2": "emacs",
        "tag 3": "c#",
        "tag 4": ".net",
        "tag 5": "osx",
        "tag 6": "data-structures",
        "tag 7": "vim",
        "tag 8": "user-interface",
        "tag 9": "asp.net"
    }
]
}
```


Pour cette question, les tags réellement attendus sont : regex, emacs et lisp.
On n'en obtient que 2 sur 3 aux positions 1 et 2, lisp se situant un peu au-delà de la 15ème position.

17- Conclusions

- Ce projet a mis en exergue la difficulté de traiter les mots considérés comme non pertinents pour le sens recherché des phrases.
- Il a donc été nécessaire de mettre des filtres comme les stop_words afin de commencer à prétraiter les textes.
- On s'aperçoit vite qu'avec les textes, le volume des dictionnaires peut très vite s'accroître, ce qui encourage certaines méthodes (TFID) ou en disqualifie d'autre (utilisation du champ Body).
- Dans l'ensemble, les modèles ont donné de bons résultats (au moins la moitié des tags étaient bien prédits).
- J'ai manqué de temps mais il pourrait être intéressant, de la même façon qu'avec la méthode Naïve Bayes, d'utiliser une régression logistique pour prévoir si un tag sera présent ou non, même si on arrive rapidement à une limite de temps d'exécution, car si cela peut être faisable avec quelques tags, cela peut rapidement devenir ingérable de faire autant de régressions.
- La difficulté de faire des tests avec beaucoup de données ne permet pas de se faire une idée définitive sur l'efficacité réelle des modèles présentés, toutefois il semble que parmi les modèles à apprentissage supervisé, les modèles avec TFID et les plus proches voisins couplés des champs Title et Tags soient les plus efficaces.
- Le modèle "incrémental" semble également améliorer les performances, on pourrait même envisager une sorte de blending avec à chaque itération un modèle différent adapté à l'évolution de la configuration.
- Les modèles à apprentissage non supervisé ont donné des scores inférieurs, toutefois, ils sont plus aptes à donner des prévisions pour de toutes nouvelles questions, car avec des modèles utilisant un historique, on ne propose que des tags existant déjà dans cet historique, ce qui pose problème par exemple si une question portait sur un tout nouveau langage ou tout sujet récent.

Quelques pistes d'améliorations:

- SHUFFLE : il serait préférable de mettre en place un shuffle sur les données d'entraînement et les données de test, je n'ai pas eu le temps de le mettre en place. Même s'il ne semble pas y avoir d'ordre apparent dans les questions de la base de données il serait préférable de choisir les données au hasard afin d'éviter tout biais.
- TAG CORR : comme nous l'avons vu avec la matrice de corrélation, il faudrait jouer sur les corrélations entre tags pour voir comment proposer des choix tenant compte de ces corrélations.
- AUTRES TABLES : j'ai choisi de n'utiliser que la table Posts car elle regroupait les champs essentiels Title et Tags. Mais il pourrait être utile d'en considérer d'autre, comme par exemple la table TagSynonyms qui permet de relier des tags comme Javascript et JS, de même que la table Users qui permettait de voir si un utilisateur n'utilise pas toujours les mêmes tags et donc de les lui proposer le cas échéant.
- STERMMER : je n'ai utilisé un stemmer que dans un modèle afin de l'illustrer, mais il pourrait être intéressant de l'utiliser partout afin de regrouper les mots d'une même famille. Cela est toutefois parfois risqué car je suis par exemple tombé sur un exemple où s'il n'y avait qu'un seul terme dans le titre ('profilers'), l'utilisateur avait spécifié deux tags différents ('profiler' et 'profiling'). Des tests sont donc à conduire pour voir dans quelle mesure on obtient un gain ou de quelle manière gérer les éventuelles exceptions

In [52]:

```
import pandas as pd
import numpy as np
from fbprophet import Prophet
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings(action='ignore', category=UserWarning, module='gensim')
%matplotlib inline
import logging
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
import gensim
import wordcloud
from wordcloud import WordCloud
import networkx as nx
import seaborn as sns
import collections
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import GaussianNB
import re
import time
from sklearn.neighbors import NearestNeighbors
from collections import Counter
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem.snowball import SnowballStemmer
from gensim.models.ldamodel import LdaModel

def getNearestWordsFromPythonWithStopWords():
    """ Méthode qui affiche les mots les plus proches du mot 'Python' en gardant les stop_words
    """
    phrases = [['The', 'python', 'is', 'a', 'snake'], ['The', 'pythone', 'is', 'not', 'a', 'snake']]
    modelP = gensim.models.Word2Vec(phrases, min_count=1, size=10)
    print("Mots les plus proches de 'python' : ", modelP.most_similar('python'))

def getNearestWordsFromPythonWithoutStopWords():
    """ Méthode qui affiche les mots les plus proches du mot 'Python' en supprimant les stop_words
    """
    phrases = [['python', 'snake'], ['pythone', 'snake']]
    modelP = gensim.models.Word2Vec(phrases, min_count=1, size=10)
    print("Mots les plus proches de 'python' : ", modelP.most_similar('python'))

def drawTitleWordCloud(_df):
    """ Méthode qui dessine le WordCloud des mots des titres
    _df -- le dataframe des données
    """
    titresToCloud = pd.Series(_df['Title'].tolist()).astype(str)
    cloud = WordCloud(width=1440, height=1080).generate(" ".join(titresToCloud.astype(str)))
    plt.figure(figsize=(10, 5))
    plt.imshow(cloud)
    plt.axis('off')

def drawTagsWordCloud(_df):
    """ Méthode qui dessine le WordCloud des tags
    _df -- le dataframe des données
    """
    tagsToCloud = pd.Series(_df['Tags'].tolist()).astype(str)
    cloud = WordCloud(width=1440, height=1080).generate(" ".join(tagsToCloud.astype(str)))
    plt.figure(figsize=(10, 5))
    plt.imshow(cloud)
    plt.axis('off')

def formText(_text):
    """ Méthode qui supprime les stop_words d'une phrase
    _text -- le texte dont on veut supprimer les stop_words
    Retour:
    _text -- le texte sans les stop_words
    """
    vectorizerW = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)", stop_words='english')
    vectorizerW.fit_transform(_text.split())
    feature_namesW = vectorizerW.get_feature_names()
    _text = " ".join(feature_namesW)
    return _text
def getTags(_df, _n):
    """ Méthode qui renvoie la liste des tags réels pour un index donné
    _df -- le dataframe des données
    _n -- l'index de l'enregistrement dont on veut la liste des tags réels
    Retour:
    listTags -- la liste des tags
    """
    listTags = _df['Tags'][_n][1:-1].split("><")
    return listTags
def getWords(_df, _n):
    return formText(_df['Title'][_n]).split()

def setEdges(_dG, _df, _n):
    listTags = []
    listWords = []
    for tag in getTags(_df, _n):
        listTags.append(tag)
        for word in getWords(_df, _n):
            _dG.add_edge(tag, word)
            listWords.append(word)
    return _dG, listTags, listWords
```

```
def drawGraphTagsLinks(_df):
    dGr = nx.DiGraph()
    listAllTags = []
    listAllWords = []
    for i in range(5):
        dGr, listTags, listWords = setEdges(dGr, _df, i)
        for tag in listTags:
            listAllTags.append(tag)
        for word in listWords:
            listAllWords.append(word)

    print("Liste des tags : ", set(listAllTags))
    print("\nListe des mots : ", set(listAllWords))
    nx.draw(dGr,with_labels=True)
    plt.show()

def heatmap(_df, _methode, _title, _annot):
    """Affichage de la heatmap de la matrice de corrélation des tags
    Arguments:
    _df -- dataframe contenant les donnée
    _methode -- méthode statistique pour la corrélation, Pearson, Spearman ou Kendall
    _title -- titre de la heatmap
    _annot -- choix de l'affichage des valeurs de corrélations
    """

    # suppression des figures
    plt.clf()
    # on calcule la matrice de corrélation pour le dataframe _df passé en paramètre
    corr = _df.corr(method=_methode)
    # masque d'affichage de la heatmap
    mask = np.zeros_like(corr, dtype=np.bool)
    mask[np.triu_indices_from(mask)] = True
    f, ax = plt.subplots(figsize=(20,12))
    cmap = sns.diverging_palette(220, 10, as_cmap=True)
    # traçage de la heatmap
    sns.heatmap(corr, cmap=cmap, cbar_kws={"shrink": .5}, mask=mask, annot=_annot)
    plt.title(_title)
    plt.show()

def createVocabFromTagsDataFrame(_df, _nbQ, _print):
    sentences = pd.Series(_df['Tags'].tolist())[:_nbQ]
    cv = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)")
    cv_fit=cv.fit_transform(sentences)
    vocab = cv.get_feature_names()
    if _print:
        print(vocab)
    matrixTr = np.transpose(cv_fit.toarray())
    dfVocab = pd.DataFrame()
    for index, c in enumerate(vocab):
        dfVocab[c] = pd.Series(matrixTr[index])
    return dfVocab

def corrBetweenTags(_df, _tag1, _tag2, _method):
    predict_tag1 = np.where(_df['Tags'].str.contains("<["+_tag1.replace('+', '\+')+"]*>")==True, 1, 0)
    predict_tag2 = np.where(_df['Tags'].str.contains("<["+_tag2.replace('+', '\+')+"]*>")==True, 1, 0)
    corr_tag1_tag2 = pd.Series(predict_tag1).corr(pd.Series(predict_tag2), _method)
    return corr_tag1_tag2

def createTagHierarchie(_df):
    listeTags = []
    def tagHierarchie(row):
        tags = row['Tags'].lower()
        listeRowTags = tags[1:-1].split("><")
        for tag in listeRowTags:
            listeTags.append(tag)
        return row

    _df.apply(tagHierarchie, axis = 1)
    return listeTags

def createMostCommonTagsDataFrame(_listeTags, _nbTags, _df, _print):
    compte = collections.Counter(_listeTags)
    mostCommonC = compte.most_common(_nbTags)
    mostCommon = []
    for c in mostCommonC:
        mostCommon.append(c[0])
    if _print:
        print(mostCommon)

    dfMostCommonTags = pd.DataFrame()
    for i in range(len(mostCommon)):
        predict_tag = np.where(_df['Tags'].str.contains("<["+mostCommon[i].replace('+', '\+')+"]*>")==True, 1, 0)
        dfMostCommonTags[mostCommon[i]] = pd.Series(predict_tag)
    return dfMostCommonTags

def deleteSpecialCharFromWord(_word):
    """ Méthode qui remplace certains caractères spéciaux (#, + etc.) car parfois non reconnus par certaines fonctions
    Arguments:
    _word -- le mot à transformer
    Retour:
    word -- le mot avec les caractères spéciaux transformés
    """
    word = _word.replace('+', '\+')
    return word
```

```python
def addSpecialCharFromWord(_dico):
    """ Méthode qui remplace dans les clefs d'un dictionnaire certains caractères spéciaux (#, + etc.) car parfois non reconn
us par certaines fonctions
    Arguments:
    _dico -- le dictionnaire à transformer
    Retour:
    newDico -- le dictionnaire avec les caractères spéciaux transformés ou le même si initialement vide
    """
    if len(_dico.keys()) > 0:
        oldKey = "".join(_dico.keys())
        values = _dico[oldKey]
        newKey = oldKey.replace("\+", "+")
        newDico = {}
        newDico[newKey] = values
        return newDico
    else:
        return _dico

def deleteSpecialCharFromTexts(_texts):
    """ Méthode qui remplace certains caractères spéciaux (#, + etc.) car parfois non reconnus par certaines fonctions
    Arguments:
    _texts -- les phrases à transformer
    Retour:
    texts -- les phrases avec les caractères spéciaux transformés
    """
    texts = []
    for text in _texts: # rajouter c en préfixe et suffixe pour éviter de prendre des termes existants
        texts.append(text.replace('#', 'cdiesec').replace('+', 'cplusc').replace('π', 'cpic'))
    return texts

def addSpecialCharFromTexts(_texts):
    """ Méthode qui remplace certains caractères spéciaux (#, + etc.) car parfois non reconnus par certaines fonctions
    Arguments:
    _texts -- les phrases à transformer
    Retour:
    texts -- les phrases avec les caractères spéciaux transformés
    """
    texts = []
    for text in _texts:
        texts.append(text.replace('cdiesec', '#').replace('cplusc', '+').replace('cpic', 'π'))
    return texts

def getGreatWordsFromSentences(_phrases):
    """ Méthode qui effectue un CountVectorizer sur différentes phrases afin de ne garder que les mots importants
    Arguments:
    _phrases -- les phrases à transformer
    Retour:
    texts_tr -- les phrases n'ayant plus que des mots importants (mots ayant potentiellement un sens important)
    """
    texts_tr = []
    for s in _phrases:
        vectorizer = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)", stop_words='english')
        # ne pas virer les nombres, car ils doivent rester dans les phrases si on veut les comparer, IE 7 ce n'est pas juste
IE
        try:
            vectorizer.fit_transform(s.split())
            feature_names = vectorizer.get_feature_names()
            texts_tr.append(" ".join(feature_names))
        except ValueError:
            print("erreur : ", s)
    return texts_tr

def getDicoTagsFromWord(_word, _df):
    """ Méthode qui récupère les enregistrements contenant un certain mot puis crée un dictionnaire des tags de ces enregistr
ements
    Arguments:
    _word -- le mot dont on veut les tags
    _df -- le dataset des données
    Retour:
    dico -- le dictionnaire des tags du dataset correspondant aux enregistrements contenant le mot _word
    """
    listWord = _df[_df.Title.str.contains(_word)].Tags
    dico = {}
    if len(listWord) > 0:
        dico[_word] = []
        for tagsent in listWord:
            wordTags = np.array(tagsent[1:-1].split('><'))
            for tag in wordTags:
                dico[_word].append(tag)
    return dico
```

```
def getAllTagsFromSentence(_sent, _df):
    """ Méthode qui crée un dictionnaire avec tous les tags correspondant aux mots de la phrase en entrée
    Arguments:
    _sent -- le phrase dont on veut les tags
    _df -- le dataset des données
    Retour:
    dicoall -- le dictionnaire des tags du dataset correspondant aux enregistrements contenant les mots de la phrase _sent
    """
    # passer tous les champs utiles en lower
    _df['Title'] = _df['Title'].str.lower()
    _df['Tags'] = _df['Tags'].str.lower()
    features = getGreatWordsFromSentences([_sent])[0].split(" ")
    dicos = []
    for feature in features:
        # on travaille les caractères quand ils sont en concurrence avec les regex (expressions régulières) des méthodes
        # et on récupère les dicos
        dico = addSpecialCharFromWord(getDicoTagsFromWord(deleteSpecialCharFromWord(feature), _df))
        dicos.append(dico)
    dicoall = {}
    for dico in dicos:
        for w in dico:
            if w in dicoall:
                dicoall[w] += dico[w]
            else:
                dicoall[w] = dico[w]
    dicoall = fillBlanck(dicoall)
    return dicoall

def fillBlanck(_dico):
    """ Méthode qui remplit les valeurs pour les feature qui n'ont pas de tags en rajoutant la feature afin de ne pas ramener
de dictionnaire vide.
    Cela permet au moins de proposer comme tags les mots initiaux du titre plutôt que rien du tout, les tags se trouvant parf
ois dans le titre
    _dico -- le dictionaire à éventuellement remplir
    Retour:
    _dico -- le dictionnaire éventuellement rempli
    """
    for feature in _dico.keys():
        values = _dico[feature]
        if len(values) == 0:
            _dico[feature].append(feature)
    return _dico

def orderDico(_dico):
    """ Méthode qui ordonne les fréquences des tags d'un dictionnaire
    _dico -- le dictionaire à ordonner
    Retour:
    dicoSorted -- le dictionnaire avec les fréquences ordonnées
    """
    dicoValues = []
    for word in _dico.keys():
        for tag in _dico[word]:
            dicoValues.append(tag)
    vectorizer = CountVectorizer(token_pattern=r"(c#|c\++|[\w, \-, ', \.]+)")
    dicoSorted = {}
    try:
        X = vectorizer.fit_transform(dicoValues)
        features = vectorizer.get_feature_names()
        frequences = (X.toarray().sum(axis=0))
        dicoFrequences = {}
        for i in range(len(frequences)):
            dicoFrequences[features[i]] = frequences[i]
        dicoSorted = sorted(dicoFrequences.items(), reverse=True, key=lambda t: t[1])
    except ValueError:
        pass
    return dicoSorted

def scoreLine(_dicoSorted, _tags, _len):
    """ Méthode qui calcule le score de présence des tags proposés dans la liste des tags existants pour les questions présen
tes dans la dataset
    _dicoSorted -- le dictionnaire des tags ordonnés par fréquence
    _tags -- les tags existants
    _len -- le nombre de tags qu'on veut proposer
    Retour:
    score -- le score de la proposition
    """
    tags = _tags[1:-1].split("><")
    listeMots = []
    score = 0
    if len(_dicoSorted) > 0: # si pour une nouvelle phrase, tous les mots sont en dehors du dictionnaire d'entraînement, alor
s le dico sera vide
        for mot in _dicoSorted[:_len]:
            listeMots.append(mot[0])
        for tag in tags:
            if tag in listeMots:
                score += 1
        score = 100.0*score/len(tags)
    return score
```

```
def scoreModel(_trainNbLine, _testNbLine, _nbTags, _df, _print):
    """ Méthode qui calcule le score pour un nombre déterminé de questions de test à partir des informations des données d'en
trafnement
    _trainNbLine -- nombre de questions d'entrafnement
    _testNbLine -- nombre de questions de test prise en dehors des questions d'entrafnement
    _nbTags -- le nombre de tags à proposer
    _df -- le dataframe des données
    _print -- imprime ou non les résultats pour chaque ligne
    Retour:
    scoreGlobal -- le score sur toutes les questions
    """
    _dfTrain = _df.iloc[:_trainNbLine, :] # questions d'entrafnement
    score = 0
    # on ne reprend pas les questions de l'entrafnement pour le test mais on utilise toujours celle du dataset afin de pouvoi
r établir un score
    # amélioration en prenant des questions au hasard
    for i in np.arange(_trainNbLine,_trainNbLine+_testNbLine):
        sentence = _df['Title'][i]
        tagsTest = _df['Tags'][i]
        dico = getAllTagsFromSentence(sentence, _dfTrain)
        dicoSorted = orderDico(dico)
        scoreTemp = scoreLine(dicoSorted, tagsTest, _nbTags)
        score += scoreTemp
        if _print:
            print("Score pour le n°{0} : {1}".format(i, scoreTemp))
    scoreGlobal = score/_testNbLine
    return scoreGlobal

def suggestGeneratedTagsFromSentence(_df, _sentence):
    """ Méthode qui suggère une liste de tags ordonnés en fonction de la question initiale
    _df -- le dataframe des données
    _sentence -- le titre de la question initiale
    Retour:
    dicoSorted -- les 15 tags les plus fréquents proposés
    """
    dico = getAllTagsFromSentence(_sentence, _df)
    dicoSorted = orderDico(dico)
    print(dicoSorted[:15])

def replaceSpecialCharInTexts(_texts):
    """ Méthode qui remplace certains caractères spéciaux qui entrent en concurrence avec les regex de certaines méthodes
    _texts -- la liste des phrases à formatter
    Retour:
    texts -- la liste des phrases formattées
    """
    texts = []
    for text in _texts:
        texts.append(text.replace('#', 'cdiesec').replace('+', 'cplusc').replace('π', 'cpic'))
    return texts

def reinjectSpecialChar(_text):
    """ Méthode qui remplace/réintroduit certains caractères spéciaux qui entrent en concurrence avec les regex de certaines
méthodes
    _texts -- la phrase à formatter
    Retour:
    text -- la phrase formattées
    """
    text = _text.replace('#', 'cdiesec').replace('+', 'cplusc').replace('π', 'cpic')
    return text

def createVocMatrixFromPhrases(_texts):
    """ Méthode qui crée la matrice des mots à partir d'un CountVectorizer
    _texts -- la liste des phrases dont on extrait le vocabulaire et dont on veut la matrice
    Retour:
    cv -- le CountVectorizer
    matrix -- la matrice
    """
    cv = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)")
    cv_fit=cv.fit_transform(_texts)
    matrix = cv_fit.toarray()
    return cv, matrix

def getNearestNeighbors(_n_neighbors, _algorithm, _matrix, _newSent):
    """ Méthode qui recherche les plus proches voisins à partir d'une matrice des mots (des questions servant de support pour
le vocabulaire)
    _n_neighbors -- le nombre de plus proches voisins que l'on désire
    _algorithm -- l'algorithme servant à la recherche des plus proches voisins
    _matrix -- la matrice des mots
    _newSent -- la nouvelle phrase dont on cherche les plus proches voisins
    Retour:
    indices -- la liste des indices des plus proches voisins
    """
    neanei = NearestNeighbors(n_neighbors=_n_neighbors, algorithm=_algorithm).fit(_matrix)
    indices = neanei.kneighbors(_newSent, return_distance=False)
    return indices
```

```python
def getListTagsToCompare(_df, _indices, _nbTagsAProposer):
    """ Méthode qui récupère la liste des tags les plus fréquents à partir des indices des plus proches voisins
    _df -- le dataframe des données
    _indices -- la liste des indices des plus proches voisins
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    Retour:
    listMoreFreqTags -- la liste des tags les plus fréquents à partir des indices des plus proches voisins
    """
    tags = _df['Tags']
    listeTags = []
    for i in _indices:
        ptags = tags[i][1:-1].split("><")
        for j in ptags:
            listeTags.append(j)
    c = Counter(listeTags)
    listMoreFreqTags = []
    for letter, count in c.most_common(_nbTagsAProposer):
        listMoreFreqTags.append(letter)
    return listMoreFreqTags

def formatFromBody(_body):
    """ Méthode qui formatte un champ Body en supprimant tous les tags HTML afin de ne garder que le texte important
    _body -- le champ que l'on veut formatter issu Posts.Body
    Retour:
    bodyFormatted -- le champ formatté en texte sans balises HTML
    """
    bodyFormatted = re.findall(r"(?![^<]*>)([\w|\.|\'|\-]+)", _body)
    bodyFormatted = " ".join(bodyFormatted)
    return bodyFormatted

def formatFromTitle(_title):
    """ Méthode qui formatte le titre
    _title -- le titre à formatter
    Retour:
    _title -- le titre formatté
    """
    # pour le champ titre, il s'agit déjà d'un texte sans balises, inutile de le traiter
    return _title

def formatFromTags(_tags):
    """ Méthode qui formatte un champ Tags en supprimant toutes les balises <> et joint ses contenus en une chaîne
    _tags -- les tags balisés concaténés
    Retour:
    listTagsToString -- la liste des tags joints en une phrase
    """
    listTagsToString = " ".join(_tags[1:-1].split('><'))
    return listTagsToString

def formatFields(row, listSupport, hasTitle, hasTags, hasBody):
    """ Méthode qui prend au choix les champs Title, Tags et Body et les formatte pour concaténer leurs contenus en une phras
e,
    méthode appelée dans un apply sur un dataframe
    row -- un enregistrement contenant éventuellement les 3 champs
    listSupport -- liste récupérant tous les enregistrements formattés
    hasTitle -- indicateur permettant de savoir si on veut formatter Title
    hasTags -- indicateur permettant de savoir si on veut formatter Tags
    hasBody -- indicateur permettant de savoir si on veut formatter Body
    """
    body = tags = title = ""
    if hasBody:
        body = formatFromBody(replaceSpecialCharInTexts([row['Body'].lower()])[0])
    if hasTitle:
        title = formatFromTitle(replaceSpecialCharInTexts([row['Title'].lower()])[0])
    if hasTags:
        tags = formatFromTags(replaceSpecialCharInTexts([row['Tags'].lower()])[0])
    ttb = title + " " + tags + " " + body
    ttb = addSpecialCharFromTexts([ttb])[0]
    listSupport.append(ttb)

def formText(_df, _index, _listCol):
    """ Méthode qui formatte un enregistrement donné d'un dataframe
    _df -- le dataframe des données
    _index -- l'index de l'enregistrement que l'on souhaite formatter
    _listCol -- liste des champs que l'on veut formatter
    Retour:
    ttb -- l'enregistrement formatté
    """
    body = tags = title = ""
    if 'Body' in _listCol:
        body = formatFromBody(replaceSpecialCharInTexts([_df['Body'][_index].lower()])[0])
    if 'Title' in _listCol:
        title = formatFromTitle(replaceSpecialCharInTexts([_df['Title'][_index].lower()])[0])
    if 'Tags' in _listCol:
        tags = formatFromTags(replaceSpecialCharInTexts([_df['Tags'][_index].lower()])[0])
    ttb = title + " " + tags + " " + body
    ttb = addSpecialCharFromTexts([ttb])[0]
    return ttb
```

```
def createVocMatrixFromTrainingPhrases(_df, _ntotal, _listCol):
    """ Méthode qui crée la matrice et le vocabulaire à partir des données d'entraînement
    _df -- dataframe des données
    _ntotal -- nombre de questions constituant l'ensemble de référence où trouver les plus proches voisins
    _listCol -- liste des champs pour lequel on veut le vocabulaire et la matrice
    Retour:
    cv -- le CountVectorizer
    matrix -- la matrice
    """
    listTitleBodyTags = []
    _df.loc[:_ntotal-1].apply(formatFields, axis = 1, args = (listTitleBodyTags, 'Title' in _listCol, 'Tags' in _listCol, 'Bo
dy' in _listCol))
    listSent = getGreatWordsFromSentences(listTitleBodyTags)
    cv, matrix = createVocMatrixFromPhrases(listSent)

    return cv, matrix

def generateTags(_newSent, _nbTagsAProposer, _df, _ntotal, _cv, _matrix, _print):
    """ Méthode qui génère la liste des tags à proposer
    _newSent -- nouvelle phrase dont on veut proposer les tags
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    _df -- dataframe des données
    _ntotal -- nombre de questions constituant l'ensemble de référence où trouver les plus proches voisins
    _cv -- le CountVectorizer
    _matrix -- la matrice
    _print -- si on souhaite ou non imprimer les résultats intermédiaires
    Retour:
    listMoreFreqTags -- la liste des tags proposés
    """
    indices = getNearestNeighbors(_nbTagsAProposer+1, "kd_tree", _matrix, _cv.transform(getGreatWordsFromSentences([reinjectS
pecialChar(_newSent)])).toarray())
    indices = indices[0][1:]
    if _print:
        print("indices = ", indices)
    listMoreFreqTags = getListTagsToCompare(_df, indices, _nbTagsAProposer)
    return listMoreFreqTags

def getScore(_index, _nbTagsAProposer, _df, _ntotal, _cv, _matrix, _listCol, _print):
    """ Méthode qui clacule le score d'une proposition de tags pour une question existante en la comparant avec la liste de s
es tags réels
    _index -- l'index de la question existante
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    _df -- dataframe des données
    _ntotal -- nombre total des questions existantes servant de support au vocabulaire (ce sont les données d'entraînement)
    _cv -- le CountVectorizer
    _matrix -- la matrice
    _listCol -- la liste des champs que l'on souhaite utiliser
    _print -- si on souhaite ou non imprimer les résultats intermédiaires
    Retour:
    score -- le score de la question existante
    """
    sent = formText(_df, _index, _listCol) # ici, puisqu'il s'agit de calculer un score, il faut une question déjà existante
    listTagsGenerated = generateTags(sent, _nbTagsAProposer, _df, _ntotal, _cv, _matrix, _print)
    listTagsExisting = _df['Tags'][_index][1:-1].split("><")
    if _print:
        print("existing : ", listTagsExisting)
        print("generated : ", listTagsGenerated)
    scoreTemp = 0
    for tag in listTagsGenerated:
        if tag in listTagsExisting:
            scoreTemp += 1
    score = scoreTemp * 100 / len(listTagsExisting)
    return score
def getScoreModel(_trainNbLine, _testNbLine, _nbTagsAProposer, _df, _listCol):
    """ Méthode qui clacule le score d'une proposition de tags pour une question existante en la comparant avec la liste de s
es tags réels
    _trainNbLine -- nombre de questions d'entraînement
    _testNbLine -- nombre de questions de test prise en dehors des questions d'entraînement
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    _df -- dataframe des données
    _listCol -- la liste des champs que l'on souhaite utiliser
    Retour:
    score -- le score du modèle pour le nombre de question demandé
    """
    cv, matrix = createVocMatrixFromTrainingPhrases(_df, _trainNbLine, _listCol) # on sort la création de la matrice afin de
ne pas la répéter à chaque nouvelle phrase

    scoreCumul = 0
    for index in np.arange(_trainNbLine, _trainNbLine+_testNbLine):
        scoreCumul += getScore(index, _nbTagsAProposer, _df, _trainNbLine, cv, matrix, _listCol, False)
    score = scoreCumul / _testNbLine
    return score
def getLenImportantWordList(_n, _df):
    """ Méthode qui calcule le nombre de mots d'un vocabulaire à partir d'un nombre donné de titres
    _n -- le nombre de titres
    _df -- le dataframe des données
    Retour:
    lenVocab -- la longueur du vocabulaire
    """
    sentences = pd.Series(_df['Title'].tolist())[:_n]
    features = getGreatWordsFromSentences(sentences)
    cv = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)")
    cv_fit=cv.fit_transform(features)
    vocab = cv.get_feature_names()
    lenVocab = len(vocab)
    return lenVocab
```

```
def getMatrixForTitles(_n, _df):
    """ Méthode qui calcule la matrice des mots pour le vocabulaire des titres
    _n -- le nombre de titres
    _df -- le dataframe des données
    Retour:
    matrix -- la matrice des mots des titres
    vocab -- le vocabulaire des mots des titres
    """
    titles = pd.Series(_df['Title'].tolist())[:_n]
    features = getGreatWordsFromSentences(titles)
    cv = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)")
    cv_fit=cv.fit_transform(features)
    vocab = cv.get_feature_names()
    matrix = cv_fit.toarray()
    return matrix, vocab

def fitGaussianNaiveBayes(_n, _tag, _matrix, _df, _print):
    """ Méthode qui entraîne le modèle Naive Bayses Gaussien avec les données de la matrice et le vecteur de présence du tag
    _n -- le nombre de titres
    _tag -- le tag dont on veut prédire la présence
    _matrix -- la matrice des mots des titres
    _df -- le dataframe des données
    _print -- choix d'affichage d'informations
    Retour:
    clf_GaussianNB -- le modèle
    """
    X = np.array(_matrix)
    tagsList = pd.Series(_df['Tags'].tolist())[:_n]
    diracTag = []
    for t in tagsList:
        tagsListList = t[1:-1].split('><')
        if _tag in tagsListList:
            diracTag.append(1)
        else:
            diracTag.append(0)
    Y = np.array(diracTag)
    if _print:
        print("Vecteur présence du tag '{0}' sur les {1} premières questions : \n{2}".format(_tag, _n, Y))
    clf_GaussianNB = GaussianNB()
    clf_GaussianNB.fit(X, Y)
    return clf_GaussianNB

def predictPresenceTagForNewSentence(_n, _df, _clf, _vocab):
    """ Méthode qui prédit la présence d'un tag dans les tags d'une nouvelle question
    _n -- le nombre de titres
    _df -- le dataframe des données
    _clf -- le modèle Naive Bayses Gaussien
    _vocab -- le vocabulaire
    Retour:
    prediction -- présence du tag
    """
    bestWords = getGreatWordsFromSentences([_df['Title'][_n]])[0].split()
    newVector = []
    for word in _vocab:
        if word in bestWords:
            newVector.append(1)
        else:
            newVector.append(0)
    prediction = _clf.predict([newVector])[0]
    return prediction

def realityTag(_n, _df, _tag):
    """ Méthode qui indique si un tag se trouve réellement dans la série de tags d'une question
    _n -- l'index de la question
    _df -- le dataframe des données
    _tag -- le tag dont on veut connaître la présence
    Retour:
     -- la présence du tag
    """
    return (int(_tag in _df['Tags'][_n][1:-1].split('><')))



def scoreModelBayesForOneWord(_tag, _deb, _fin, _df, _clf, _vocab):
    """ Méthode qui retourne le score de la méthode Naive Bayes Gaussienne pour un tag donné
    _tag -- le tag que l'on confronte aux tags réels
    _deb -- le début de la fenêtre de test, postérieur aux index d'entraînement
    _fin -- la fin de la fenêtre de test, postérieur aux index d'entraînement
    _df -- le dataframe des données
    _clf -- le modèle Naive Bayses Gaussien
    _vocab -- le vocabulaire
    Retour:
    score -- le score sur l'ensemble des cas (tag présent ou non présent)
    score1 -- le score uniquement sur les cas où le tag doit être présent (cf. explication ci-dessous)
    """
    score = 0
    score1 = 0 # ici on distingue bien le cas particulier où le tag recherché est bien présent dans la liste des tags car
               # prendre tous les cas peut fausser l'impression donnée par le score, il est plus intéressant de savoir si
               # un tag, réellement présent dans la liste des tags d'une question, est bien prédit que de savoir si une
               # multitude de tags non présents seront effectivement non prédits... ce qui compte, ce sont les tags réels
    countReality1 = 0
```

```
    for i in np.arange(_deb+1, _fin+1):
        prediction = predictPresenceTagForNewSentence(i, _df, _clf, _vocab)
        reality = realityTag(i, _df, _tag)
        #print("prediction = {0}, reality = {1}".format(prediction, reality))
        if reality == 1:
            countReality1 += 1   # nombre de tags réels
            if prediction == 1:
                score1 += 1      # le score pour les tags réels
        score += 1 - abs(prediction - reality) # le score pour la totalité des tags
    score = 100 * score / (_fin - _deb)
    score = np.around(score, decimals=2)
    if countReality1 > 0:
        score1 = 100 * score1 / countReality1
    else:
        score1 = -1
    return score, score1

def scoreModelBayesForSeveralQuestions(_deb, _fin, _df, _n):
    """ Méthode qui retourne le score de la méthode Naive Bayes Gaussienne pour tous les tags d'un certain nombre de question
s
    _deb -- le début de la fenêtre de test, postérieur aux index d'entraînement
    _fin -- la fin de la fenêtre de test, postérieur aux index d'entraînement
    _df -- le dataframe des données
    _n -- le nombre de titres d'entraînement
    Retour:
    scoreTotal -- le score sur l'ensemble des cas (tag présent ou non présent)
    scoreTotal1 -- le score uniquement sur les cas où les tags doivent être présents
    """
    countWord = 0
    countWord1 = 0 # le compte pour les mots qui ont au moins une donnée réelle 1
    score = 0
    score1 = 0
    matrix, vocab =  getMatrixForTitles(_n, _df)
    listAllTags = set() #set pour avoir des éléments uniques
    for i in np.arange(_deb+1, _fin+1):
        listTags = _df['Tags'][i][1:-1].split('><')
        for w in listTags:
            listAllTags.add(w)

    for w in listAllTags:
        countWord += 1
        countWord1 += 1
        clf_GaussianNB = fitGaussianNaiveBayes(_n, w, matrix, _df, False)
        scoreWord, scoreWord1 = scoreModelBayesForOneWord(w, _deb, _fin, _df, clf_GaussianNB, vocab)
        score += scoreWord
        if scoreWord1 >=0: # les mots avec au moins une donnée réelle 1
            score1 += scoreWord1
        else:
            countWord1 -= 1 # quand on n'a aucune donnée réelle 1, le mot

    scoreTotal = score / countWord
    if countWord1 > 0:
        scoreTotal1 = score1 / countWord1
    else:
        scoreTotal1 = -1
    return scoreTotal, scoreTotal1

def createListeIndicePlusProchesVoisins(_df, _trainingSentences, _testSentence, _nbTagsAProposer):
    """ Méthode qui crée la liste des indices des plus proches voisins
    _df -- le dataframe des données
    _trainingSentences -- la liste des questions dans lesquelles puiser les plus proches voisins
    _testSentence -- la question dont on veut les plus proches voisins
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    Retour:
    listeIndicePlusProchesVoisins -- la liste des indices des plus proches voisins
    """
    vectorizer = TfidfVectorizer(min_df=1, token_pattern=r"(c#|c\++|[\w\-\.\']+)", stop_words='english')
    allSentences = [_testSentence] + _trainingSentences
    myFeatures = vectorizer.fit_transform(allSentences)
    scores = (myFeatures[0, :] * myFeatures[1:, :].T).A[0] # la première ligne est la question qu'on veut confronter aux autr
es
    best_score = np.argmax(scores)
    mostSimilarQuestion = _trainingSentences[best_score]
    listeIndicePlusProchesVoisins = scores.argsort()[-_nbTagsAProposer:][::-1]
    return listeIndicePlusProchesVoisins

def generateTagsListToSuggest(_df, _trainingSentences, _testSentence, _nbTagsAProposer):
    """ Méthode qui crée la liste des tags suggérés
    _df -- le dataframe des données
    _trainingSentences -- la liste des questions dans lesquelles puiser les plus proches voisins
    _testSentence -- la question dont on veut les plus proches voisins
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    Retour:
    listTagsGenerated -- la liste des tags suggérés
    """
    listeIndicePlusProchesVoisins = createListeIndicePlusProchesVoisins(_df, _trainingSentences, _testSentence, _nbTagsAPropo
ser)
    listTagsGenerated = getListTagsToCompare(_df, listeIndicePlusProchesVoisins, _nbTagsAProposer)
    return listTagsGenerated
```

```
def getScoreTfid(_df, _trainNbLine, _testSentenceIndex, _nbTagsAProposer, _print):
    trainingSentences = _df['Title'].tolist()[:_trainNbLine]
    testSentence = _df['Title'][_testSentenceIndex]
    listTagsGenerated = generateTagsListToSuggest(_df, trainingSentences, testSentence, _nbTagsAProposer)
    listTagsExisting = _df['Tags'][_testSentenceIndex][1:-1].split("><")
    scoreTemp = 0
    for tag in listTagsGenerated:
        if tag in listTagsExisting:
            scoreTemp += 1
    score = scoreTemp * 100 / len(listTagsExisting)
    if _print:
        print("listTagsGenerated : ", listTagsGenerated)
        print("listTagsExisting : ", listTagsExisting)
        print("score : ", score)
    return score

def getScoreTfidModel(_df, _trainNbLine, _testNbLine, _nbTagsAProposer, _print):
    """ Méthode qui clacule le score d'une proposition de tags pour une question existante en la comparant avec la liste de s
es tags réels
    _df -- dataframe des données
    _trainNbLine -- nombre de questions d'entraînement
    _testNbLine -- nombre de questions de test prise en dehors des questions d'entraînement
    _nbTagsAProposer -- nombre de tags qu'on veut proposer
    _ntotal -- nombre total des questions existantes servant de support au vocabulaire
    Retour:
    score -- le score du modèle pour le nombre de question demandé
    """
    scoreCumul = 0
    for index in np.arange(_trainNbLine, _trainNbLine + _testNbLine):
        scoreCumul += getScoreTfid(_df, _trainNbLine, index, _nbTagsAProposer, _print)
    score = scoreCumul / _testNbLine
    return score

def getListTagsProposes(_dicoOrdered):
    """ Méthode qui fournit la liste des tags proposés
    _dicoOrdered -- le dictionnaire ordonné sur lequel se base la proposition de liste de tags
    Retour:
    listTagsProposes -- la liste des tags proposés
    """
    listTagsProposes = []
    for c in _dicoOrdered:
        listTagsProposes.append(c[0])
    return listTagsProposes

def getTagChoisi(_listTagsReels, _listTagsProposes, _listTagsChoisis):
    """ Méthode qui produit automatiquement le tag choisi
    _listTagsReels -- liste des tags réels
    _listTagsProposes -- liste des tags proposés
    _listTagsChoisis -- liste des tags choisis
    Retour:
    tagChoisi -- le tag choisi automatiquement
    """
    for tag in _listTagsReels:
        if tag in _listTagsProposes and tag not in _listTagsChoisis:
            tagChoisi = tag
            return tagChoisi
    return ""

def getScore(_listTagsToCheck, _listTagsReels):
    """ Méthode qui calcule le score d'une liste de tags vis-à-vis de la liste réelle
    _listTagsToCheck -- la liste des tags à vérifier
    _listTagsReels -- la liste réelle des tags
    Retour:
    score -- le score
    """
    score = 0
    for tag in _listTagsReels:
        if tag in _listTagsToCheck:
            score += 1
    score = 100.0*score/len(_listTagsReels)
    return score

def runIteration(_sent, _dfTrain, _listTagsReels, _listTagsChoisis, _len, _print):
    """ Méthode qui
    _sent -- titre sur lequel porte l'itération
    _dfTrain -- le dataframe des données d'entraînement
    _listTagsReels -- liste des tags réels
    _listTagsChoisis -- liste des tags choisis
    _len -- nombre de tags à proposer
    _print -- choix d'affichage d'information
    Retour:
    continueGranted -- permis de continuer l'itération
    _sent -- le titre auquel on ajoute les tags choisis pour l'itération suivante
    _listTagsChoisis -- liste des tags choisis pour l'itération suivante
    """
    allTagsFromSentence = getAllTagsFromSentence(_sent, _dfTrain)
    dicoOrdered = orderDico(allTagsFromSentence)
    if len(dicoOrdered) >= _len:
        dicoOrdered = dicoOrdered[:_len]
        if _print:
            print("dicoOrdered = ", dicoOrdered)
        listTagsProposes = getListTagsProposes(dicoOrdered)
        # si la liste des tags réels se trouve dans la liste des tags proposés, il est inutile de continuer
        continueGranted = (getScore(listTagsProposes, _listTagsReels) != 100.0)
```

```
    else:
        listTagsProposes = [] # il n'y a rien a proposer, le dictionnaire est vide
        continueGranted = False
    if continueGranted:
        tagChoisi = getTagChoisi(_listTagsReels, listTagsProposes, _listTagsChoisis)
        if tagChoisi == "": # on cherche le premier tag de listTagsProposes qui ne soit pas dans la liste des tags choisis en
espérant qu'on tombe sur un de la liste des tags réels
            for tagPropose in listTagsProposes:
                if tagPropose not in _listTagsChoisis:
                    tagChoisi = tagPropose
                    break
        # sinon, le tag est bien dans la liste des tags réels donc on l'ajoute à la liste des tags choisis
        if _print:
            print("tagChoisi = ", tagChoisi)

        _listTagsChoisis.append(tagChoisi)
        _sent = _sent + " " + tagChoisi

    return continueGranted, _sent, _listTagsChoisis

def automaticTagsGeneratedList(_df, _trainNbLine, _n, _len, _print):
    """ Méthode qui génère les tags après un choix automatique des différents tags successifs
    _df -- le dataframe des données
    _trainNbLine -- le nombre de'enregistrents d'entraînement
    _n -- index de la question
    _len -- nombre de tags à proposer
    _print -- choix d'affichage d'information
    Retour:
    listTagsChoisis -- liste des tags choisis
    """
    # initialisation de la récurrence
    # paramètrage
    sent = _df['Title'][_n]
    listTagsReels = _df['Tags'][_n][1:-1].split('><')
    listTagsChoisis = []

    if _print:
        print("Title = ", sent)
        print("\nListe des tags réels = ", listTagsReels)

    dfTrain = _df.iloc[:_trainNbLine, :]
    # itération, autant de fois qu'il y a de tags dans la réalité
    for tag in listTagsReels:
        if _print:
            print("\nItération avec le tag : ", tag)

        continueGranted, sent, listTagsChoisis = runIteration(sent, dfTrain, listTagsReels, listTagsChoisis, _len, _print)

        if not continueGranted: # si on arrête l'itération, c'est qu'on a dans la liste des tags proposés, tous les tags réel
s
            return listTagsReels

    return listTagsChoisis

def scoreAutomaticLine(_df, _trainNbLine, _n, _nbTags):
    """ Méthode qui calcule le score de présence des tags proposés dans la liste des tags existants pour les questions présen
tes dans la dataset
    _df -- le dataframe des données
    _trainNbLine -- nombre de questions d'entraînement
    _n -- l'index de la question existante
    _nbTags -- le nombre de tags qu'on veut proposer
    Retour:
    score -- le score de la question
    """
    listTagsChoisis = automaticTagsGeneratedList(_df, _trainNbLine, _n, _nbTags, False)
    listTagsReels = _df['Tags'][_n][1:-1].split("><")
    score = getScore(listTagsChoisis, listTagsReels)
    return score

def scoreAutomaticModel(_trainNbLine, _testNbLine, _nbTags, _df, _print):
    """ Méthode qui calcule le score pour un nombre déterminé de questions de test
    _trainNbLine -- nombre de questions d'entraînement
    _testNbLine -- nombre de questions de test prise en dehors des questions d'entraînement
    _nbTags -- le nombre de tags à proposer
    _df -- le dataframe des données
    _print -- imprime ou non les résultats pour chaque ligne
    Retour:
    scoreGlobal -- le score sur toutes les questions
    """
    score = 0
    for i in np.arange(_trainNbLine, _trainNbLine + _testNbLine):
        scoreTemp = scoreAutomaticLine(_df, _trainNbLine, i, _nbTags)
        score += scoreTemp
        if _print:
            print("Score pour le n°{0} : {1}".format(i, scoreTemp))
    scoreGlobal = score/_testNbLine
    return scoreGlobal
```

```
def make_autopct2(values):
    def my_autopct2(pct):
        """Customisation de l'affichage des valeurs (v:d)
        Arguments:
        values -- nombres de variables pour chaque section
        pct -- pourcentages de variables pour chaque section
        """
        total = sum(values)
        val = int(round(pct*total/100.0))
        return '{v:d}'.format(v=val)
    return my_autopct2

def getGreatWordsFromSentence(_phrase):
    """ Méthode qui effectue un CountVectorizer sur différentes phrases afin de ne garder que les mots importants
    Arguments:
    _phrases -- la phrase à transformer
    Retour:
    texts_tr -- la phrase n'ayant plus que des mots importants (mots ayant potentiellement un sens important)
    """
    vectorizer = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)", stop_words='english')
    # ne pas virer les nombres, car ils doivent rester dans les phrases si on veut les comparer, IE 7 ce n'est pas juste IE
    feature_names = []
    try:
        vectorizer.fit_transform([_phrase])
        feature_names = vectorizer.get_feature_names()
    except ValueError:
        pass
    return feature_names

def drawCamembertOfPcTagsInTitle(_pcTagsInTitle):
    values = [_pcTagsInTitle, 100 - _pcTagsInTitle]
    _, _, autotexts = plt.pie(values, autopct=make_autopct2(values))
    for i in range(2):
        autotexts[i].set_color('white')
        autotexts[i].set_fontsize(16)
    plt.title("Pourcentage de tags contenus / non contenus dans le titre", bbox={'facecolor':'0.8', 'pad':5})
    plt.legend(['In', 'Out'], loc='lower right')
    plt.show()

def drawCamembertOfPcTitlesWithoutTags(_pcTitleWithoutTags):
    values = [_pcTitleWithoutTags, 100 - _pcTitleWithoutTags]
    _, _, autotexts = plt.pie(values, autopct=make_autopct2(values))
    for i in range(2):
        autotexts[i].set_color('white')
        autotexts[i].set_fontsize(16)
    plt.title("Pourcentage de titres sans / avec tags", bbox={'facecolor':'0.8', 'pad':5})
    plt.legend(['Sans', 'Avec'], loc='lower right')
    plt.show()

def suggestWordsFromTitleHistoric(_newSentence, _df, _stemming):
    """ Méthode qui suggère une liste de mots déterminés comme étant les plus fréquents d'un dataframe d'entraînement via un
dictionnaire ordonnés des fréquences des mots
    _newSentence -- le titre de la nouvelle question dont on veut suggérer des tags
    _df -- le dataframe d'entraînement
    _stemming -- chois d'utiliser ou non l'élagage des données
    Retour:
    dicoSorted -- le dictionnaire ordonnés des fréquences des mots
    """
    # suppression des stop_words pour le nouveau titre
    lis = getGreatWordsFromSentences([_newSentence])[0].split()
    # récupération des mots des titres de l'historique ayant au moins un des mots du nouveau titre
    titlesWordsFromNewWords = []
    for w in lis:
        listWord = _df[_df.Title.str.contains(w.replace("+", "\+"))].Title
        for s in listWord:
            for wt in s.split():
                titlesWordsFromNewWords.append(wt)

    # élagage des mots afin de rassembler les termes dérivant d'une racine commune
    if _stemming:
        stemmer = SnowballStemmer("english")
        titlesWordsFromNewWords = [stemmer.stem(word) for word in titlesWordsFromNewWords]

    # comptage des fréquences
    vectorizerW = CountVectorizer(token_pattern=r"(c#|c\++|[\w\-\.\']+)", stop_words='english')
    if len(titlesWordsFromNewWords) > 0:
        X = vectorizerW.fit_transform(titlesWordsFromNewWords)
        features = vectorizerW.get_feature_names()
        frequences = (X.toarray().sum(axis=0))
        dicoFrequences = {}
        for i in range(len(frequences)):
            dicoFrequences[features[i]] = frequences[i]
        dicoSorted = sorted(dicoFrequences.items(), reverse=True, key=lambda t: t[1])
    else:
        dicoSorted = {}
    return dicoSorted
```

```python
def getScoreSheerTitle(_df, _trainingDataNumber, _testDataNumber, _print):
    """ Méthode qui calcule le score du modèle utilisant uniquement les titres d'un dataframe d'entraînement
    _df -- le dataframe des données
    _trainingDataNumber -- le nombre de questions du dataframe servant de d'entraînement
    _testDataNumber -- le nombre de questiond du dataframe servant de test
    _print -- choix d'affichage de données intermédiaires
    Retour:
    scoreGlobal -- score global du modèle
    """
    trainingData = tagsAll.iloc[:_trainingDataNumber, :]
    scoreGlobal = 0
    for n in np.arange(_trainingDataNumber, _trainingDataNumber + _testDataNumber):
        newSentence = _df['Title'][n]
        newSentenceWords = suggestWordsFromTitleHistoric(newSentence, trainingData, True)
        if len(newSentenceWords) > 0:
            newSentenceWords = [c[0] for c in newSentenceWords[:15]]
            tagsList = _df['Tags'][n][1:-1].split('><')
            score = 0
            for w in tagsList:
                if w in newSentenceWords:
                    score += 1
            score = 100 / len(tagsList) * score
        else:
            score = 0
        if _print:
            print("Exemple : ")
            print("Titre : {0}".format(newSentence))
            print("Liste des tags : {0}".format(tagsList))
            print("Tags proposés : {0}".format(newSentenceWords))
            print("Score : {0} %".format(score))
        scoreGlobal += score
    scoreGlobal = scoreGlobal / _testDataNumber
    return scoreGlobal

def runLdaModel(_sentences, _num_topics, _num_words):
    """ Méthode qui lance le modèle LDA sur un ensemble de documents
    _sentences -- les titres des questions sur lequel est utilisé LDA
    _num_topics -- nombre de thèmes qu'on veut extraire de l'ensemble des titres
    _num_words -- nombre de mots que doivent comproter les thèmes
    Retour:
    vect -- le CountVectorizer des documents
    ldamodel -- le modèle LDA
    topics -- le descriptif des thèmes
    """
    vect = CountVectorizer(stop_words='english', token_pattern=r"(c#|c\++|[\w\-\.\']+)")
    # Fit and transform
    X = vect.fit_transform(_sentences)
    # Convert sparse matrix to gensim corpus.
    corpus = gensim.matutils.Sparse2Corpus(X, documents_columns=False)
    # Mapping from word IDs to words (To be used in LdaModel's id2word parameter)
    id_map = dict((v, k) for k, v in vect.vocabulary_.items())

    # Use the gensim.models.ldamodel.LdaModel constructor to estimate
    # LDA model parameters on the corpus, and save to the variable `ldamodel`
    ldamodel = LdaModel(corpus, num_topics=_num_topics, id2word=id_map, passes=25, random_state=34)

    # get the topics
    topics = ldamodel.print_topics(num_topics=_num_topics, num_words=_num_words)

    return vect, ldamodel, topics

def extractWordsFromTopicsDescription(_texts):
    """ Méthode qui formatte le descriptif des thèmes générés par LDA en une liste de mots décrivant les thèmes
    _texts -- la description des thèmes ([(0, '0.131*"value" + 0.131*"fastest" + 0.131*"way" + 0.131*"n"...], ...)
    Retour:
    listWords -- liste des mots des thèmes ( [ ['value', 'fastest', 'way', 'n', ...], ... ] )
    """
    listWords = []
    for i in range(len(_texts)):
        text = [w[7:-1] for w in [w.strip() for w in _texts[i][1].split('+')]]
        listWords.append(text)
    return listWords

def predictLdaNewSentences(_vect, _ldamodel, _newSentences):
    """ Méthode qui donne la liste des probabilités de correspondance entre les nouveaux titres est les titres d'entraînement
    _vect -- le CountVectorizer des documents
    _ldamodel -- le modèle LDA
    _newSentences -- le ou les nouveaux titres
    Retour:
    fittingList -- la liste des probabilités de correspondance entre les nouveaux titres est les titres d'entraînement
    """
    # Fit and transform
    X = _vect.transform(_newSentences)
    # Convert sparse matrix to gensim corpus.
    corpus = gensim.matutils.Sparse2Corpus(X, documents_columns=False)
    fittingList = []
    if len(corpus) > 0:
        fittingList = list(_ldamodel[corpus])[0]
    return fittingList
```

```python
def getWordsFromMostProbableTopic(_listProbabilities, _listTopicsWords):
    """ Méthode qui renvoie la liste des mots du thème dont le nouveu titre a le plus de chance de relever
    _listProbabilities -- la liste des probabilités de correspondance
    _listTopicsWords -- liste des mots des thèmes
    Retour:
    listBestTopicsWords -- liste des mots du thème le plus probable pour le nouveau titre
    """
    listBestTopicsWords = []
    if len(_listProbabilities) > 0:
        best = sorted(_listProbabilities, key=lambda x: x[1], reverse=True)[0][0]
        listBestTopicsWords = _listTopicsWords[best]
    return listBestTopicsWords

def getScoreModelLDA(_df, _nbTopics, _trainingQuestionNumber, _testQuestionNumber):
    """ Méthode qui calcule le score global pour tous les titres de test pour le modèle LDA
    _df -- le dataframe des données
    _nbTopics -- le nombre de thèmes
    _trainingQuestionNumber -- le nombre de questions (titres) d'entraînement
    _testQuestionNumber -- le nombre de questions (titres) de test
    Retour:
    scoreGlobal -- le score global du modèle LDA pour les données d'entraînement et de test
    """
    # liste des titres des questions d'entraînement
    sentences = pd.Series(_df['Title'].tolist())[:_trainingQuestionNumber]
    # génération des thèmes
    vect, ldamodel, topics = runLdaModel(sentences, _nbTopics, 15)
    # formattage de la description des thèmes
    listTopicsWords = extractWordsFromTopicsDescription(topics)

    scoreGlobal = 0
    for n in np.arange(_trainingQuestionNumber, _trainingQuestionNumber + _testQuestionNumber):
        # liste des correspondances (question / ressemblance)
        fittingList = predictLdaNewSentences(vect, ldamodel, [_df['Title'][n]])
        # liste des mots du thème le plus probable pour le nouveau titre
        listTagsToCheck = getWordsFromMostProbableTopic(fittingList, listTopicsWords)
        # liste des tags réels
        listRealTags = _df['Tags'][n][1:-1].split("><")
        # comparaison des tags réels et des mots du thème le plus probable pour le nouveau titre
        score = getScore(listTagsToCheck, listRealTags)
        scoreGlobal += score
    # score global pour tous les titres de test
    scoreGlobal = scoreGlobal / _testQuestionNumber

    return scoreGlobal
```