Damien ALOUGES
Cécile AMSALLEM IBO1

Rapport du projet 2
Advanced Machine Learning

Lien vers le notebook :

https://colab.research.google.com/drive/1Ea7fW2BMqEuV7nBicgrjBIg5xXX6oqcT#scrollTo=UIwTIfUSIPjo

IMPORTANT :

Le rapport reprend en partie le rapport du premier projet, puisque nous sommes partis de cette base de code pour avancer sur la nouvelle problématique. L'importation des données,une partie du pré-processing, la génération de keywords et le clustering ont été fait dans le premier projet.

I) Importation des données

Après avoir téléchargé le fichier xml nous avons commencé par enlever la partie renseignements du corpus, ou figure la structure du xml et la licence. Pour parser nous avons utilisé la librairie xml.etree, puis nous allons chercher "à la main" les éléments que nous cherchons dans les articles du corpus.

```
tree1 = etree.parse("corpus_taln_v2.tei.xml")
```

Cette ligne permet de récupérer le fichier xml sous forme d'un arbre, ce qui permet d'accéder au contenu des articles assez simplement, en commençant par boucler sur les articles de l'arbre :

```
root = tree1.getroot()
for article in root:
```

Ensuite il suffit de chercher les données voulues en étudiant la structure des articles du xml, par exemple pour prendre les abstracts en français nous avons procédés comme ceci :

```
abstract_fr = article[1][0][0][0].text
```

Ici chaque [ ] correspond à un niveau de l'arbre, ce qui correspond à une bannière dans le xml, par exemple article[1][0][0][0] correspond à <text> <front> <p> en xml, ce qui est l'emplacement de l'abstract en français dans les articles.

Nous avons choisi de d'importer les données dans un dataframe qui a 5 colonnes :

```
df_cols = ["abstract_fr", "keywords_fr", "intro", "titre", "date"]
```

La fonction text_cleaner sert à retirer le "\t\n" présent dans les données récupérées, ainsi que transformer les données type None en string "None" afin de récupérer que des données type string.

```
def text_cleaner(string):
  if string != None :
    string = string.translate(str.maketrans("\n\t", "  "))
  elif string == None:
    string = "None"
  return (string)
```

Quand la boucle sur les articles se termine on met toutes les données dans le dataframe ci-dessous :

```
1 df = pd.DataFrame(rows, columns = df_cols)
2 df
```

	abstract_fr	keywords_fr	intro	titre	date
0	Nous considérons dans notre travail la tâche ...	None	Le modèle de la Grammaire Applicative et Cogn...	Éléments de conception d'un système d'interpr...	1997
1	Nous donnons ici un aperçu du logiciel DECID ...	None	Dans le domaine de l'ingénierie linguistique ...	Informatisation du dictionnaire explicatif et...	1997
2	Diverses méthodes ont été proposées pour cons...	None		Construction d'une représentation sémantique ...	1997
3	Le terme de lambda-DRT désigne un ensemble de...	None	La « Théorie des Représentations Discursives ...	Systèmes de types pour la (lambda-)DRT ascend...	1998
4	Dans cet article, nous comparons deux modèles...	None	TAG est un formalisme initialement développé ...	Une grammaire TAG vue comme une grammaire Sen...	1998
...	...	...	...	...	...
1597	Dans cet article, nous présentons une approch...	Curriculum d'apprentissage, transfert d'appre...	L'apprentissage humain est réalisé par étapes...	Curriculum d'apprentissage : reconnaissance d...	2019
1598	Cet article présente une méthodologie de déte...	ellipse, anglais, corpus, sous-titres, détect...	L'ellipse renvoie à une incomplétude syntaxiq...	Détection des ellipses dans des corpus de sou...	2019
1599	La génération automatique de poésie est une t...	génération de poésie, réseaux de neurones, fa...	La génération automatique de poésie est une t...	La génération automatique de poésie en français	2019
1600	Nous proposons une architecture neuronale ave...	Réseaux neuronaux, modélisation de séquences,...	L'étiquetage de séquences est un problème imp...	Modèles neuronaux hybrides pour la modélisati...	2019
1601	Nous présentons la base PolylexFLE, contenant...	expressions polylexicales vebales, niveau CEC...	Les expressions polylexicales (EP) constituen...	PolylexFLE : une base de données d'expression...	2019

1602 rows × 5 columns

Comme on peut le voir sur cet affichage certain articles n'ont pas de keywords ou d'introduction ou d'abstract en français

II) Pré-processing

1) Pré-processing projet 1

Par la suite, nous avons utilisé un replacer pour retirer les abréviations et la ponctuation : les d' deviennent de, les l' deviennent des, ce qui facilitent le traitement des données. Les paternes de remplacement ont été défini par nous selon nos besoins. Nous avons également défini un tokenizer.

```
tokenizer=RegexpTokenizer("[\w]+")

replacement_patterns = [
(r'd\'', 'de '),
(r'l\'', 'le '),
(r'qu\'', 'que '),
(r',', ''),
(r'-', ' '),
(r'\.', ''),
(r'\;', '')
]

class RegexpReplacer(object):
    def __init__(self, patterns=replacement_patterns):
        self.patterns = [(re.compile(regex), repl) for (regex, repl) in patterns]

    def replace(self, text):
        s = text
        for (pattern, repl) in self.patterns:
            s = re.sub(pattern, repl, s)
        return s

replacer = RegexpReplacer()
```

Par la suite, nous créons deux nouvelles colonnes dans notre dataframe : tokenized_abstract et cleaned_abstract. Le premier sert à contenir l'abstract tokenizé et le second sert à contenir l'abstract sans ponctuation et tout en minuscule.

```
tokenized_abstract = []
cleaned_abstract = []
for abstract in df['abstract_fr'] :
  tokenized_abstract.append((tokenizer.tokenize(replacer.replace(abstract.lower()))))
  cleaned_abstract.append(replacer.replace(abstract.lower()))

df['tokenized_abstract'] = tokenized_abstract
df['cleaned_abstract'] = cleaned_abstract
```

Par la suite, nous définissons la liste des mots les plus communs de notre jeu de données, ceux-là n'ayant pas de valeur s'ajouteront à nos stopwords.

```
common_word = pd.Series(' '.join(df['cleaned_abstract']).split()).value_counts()[:50]
common_word
```

Nous retirons ensuite les stopwords et les mots communs établis ci-dessus des mots contenus dans tokenized_abstract et nous lemmatizons tous les mots restants, comme l'indique la capture d'écran ci-dessous.

```
from string import digits
nltk.download('wordnet')
nltk.download('stopwords')

stopwords = nltk.corpus.stopwords.words('french')
lemmatizer_output=WordNetLemmatizer()
for index in range(len(df['tokenized_abstract'])) :
  df['tokenized_abstract'][index] =
    [lemmatizer_output.lemmatize(word.lower(), pos='v') for word in df['tokenized_abstract'][index] if word not in common_word]
  df['tokenized_abstract'][index] =
    [lemmatizer_output.lemmatize(word.lower(), pos='v') for word in df['tokenized_abstract'][index] if word not in stopwords]
```

Enfin, nous supprimons toutes les données n'ayant ni keywords ni abstract de notre dataframe, puis nous remettons les indexes à jour pour pouvoir manipuler plus simplement les données.

```
# Ici nous supprimons les lignes n'ayant ni abstract ni keyword

num_line = []
for index in range(0, len(df)):
  if (df['abstract_fr'][index] == 'None' or df['abstract_fr'][index] == ' ') and df['keywords_fr'][index] == 'None':
    num_line.append(index)

print(num_line)
num_line.append(1545)
df = df.drop(num_line)
df.reset_index(inplace = True)
len(df)
```

Par la suite, nous avons définis une liste de mot courant, moins que ceux définis dans les mots communs, mais non représentatifs du contenu de l'article : les nombres, les verbes conjugués, et les mots de logique. Une fois tous les mots inutiles supprimés, nous utilisons un counter pour repérer les mots importants de chaque abstract.

Dans le cas où nous n'avions pas de keywords, nous incrustons cette liste de mots dans la colonne keywords_fr.

Puis, nous avons créé une nouvelle colonne new_keywords, qui concatènent les keywords d'origine et ceux obtenus par notre algorithme afin de comparer les résultats.

```
for index in range(0, len(df)):
  keywords_list = []
  if df['keywords_fr'][index] != 'None' :
    keywords_list = str(df['keywords_fr'][index]).split(',')
  new_keywords[index] = new_keywords[index] + keywords_list


df['new_keywords'] = new_keywords
```

Voici à quoi ressemble notre dataframe une fois modifié :

	index	abstract_fr	keywords_fr	intro	titre	date	tokenized_abstract	cleaned_abstract	new_keywords
0	0	Nous considérons dans notre travail la tâche ...	None	Le modèle de la Grammaire Applicative et Cogn...	Éléments de conception d'un système d'interpr...	1997	[considérons, travail, tâche, traitement, visa...	nous considérons dans notre travail la tâche ...	[relative, grande, vitesse]
1	1	Nous donnons ici un aperçu du logiciel DECID ...	None	Dans le domaine de l'ingénierie linguistique ...	Informatisation du dictionnaire explicatif et...	1997	[donnons, ici, aperçu, logiciel, decid, dévelo...	nous donnons ici un aperçu du logiciel decid ...	[logiciel, développé, geta, informatiser, réda...
2	2	Diverses méthodes ont été proposées pour cons...	None		Construction d'une représentation sémantique ...	1997	[diverses, méthodes, proposées, construire, gr...	diverses méthodes ont été proposées pour cons...	[syntaxique, construction, représentation, sém...
3	3	Le terme de lambda-DRT désigne un ensemble de...	None	La « Théorie des Représentations Discursives ...	Systèmes de types pour la (lambda-)DRT ascend...	1998	[terme, lambda, drt, désigne, ensemble, méthod...	le terme de lambda drt désigne un ensemble de...	[terme, lambda, ensemble, construire, mise, oe...
4	4	Dans cet article, nous comparons deux modèles...	None	TAG est un formalisme initialement développé ...	Une grammaire TAG vue comme une grammaire Sen...	1998	[comparons, modèles, linguistiques, utilisés, ...	dans cet article nous comparons deux modèles ...	[tag]
...	...	...	...	...	...	...	...	...	...
1539	1597	Dans cet article, nous présentons une approch...	Curriculum d'apprentissage, transfert d'appre...	L'apprentissage humain est réalisé par étapes...	Curriculum d'apprentissage : reconnaissance d...	2019	[bout, bout, extraction, concepts, sémantiques...	dans cet article nous présentons une approche...	[extraction, particulier, pilotée, curriculum,...

2) Pré-processing projet 2

Keywords :

Nous avons dû transformer nos données afin de leur appliquer un algorithme de Machine Learning. Pour se faire, nous avons appliqué un one hot encoding à nos keywords, afin de pouvoir appliquer deux algorithmes dessus : un Random Forest Classifier et un Support Vector Classifier.

Tout d'abord nous stockons toutes nos dates dans y :

```
y = df['date']
y
```

```
0       1997
1       1997
2       1997
3       1998
4       1998
        ...
1539    2019
1540    2019
1541    2019
1542    2019
1543    2019
```

Puis nous stockons tous nos mots clefs de manière « unique » (une seule occurrence de chaque mot clef) dans une liste appelée data :

```
keywords = df['new_keywords']
data = []
for i in range (0, len(keywords)):
  for j in range (0, len(keywords[i])):
    if keywords[i][j] not in data:
      data.append(keywords[i][j])
values = np.array(data)
print(values)
```

```
['relative' 'grande' 'vitesse' ... ' expressions polylexicales vebales'
 ' niveau CECR' ' TAL pour la didactique du FLE. ']
```

Nous stockons dans x au format Dataframe nos keywords (ceux générés précédemment et ceux déjà existants) et nous créons pour chaque mot clef présent dans data une colonne dans x qui lui sera associée. Ces colonnes ne contiennent d'abord que des 0.

En suite, nous appliquons le principe du one-hot encoding : nous mettons dans les colonnes créées ci-dessus un 1 si le mot clef associé à la colonne est dans la liste des mots clefs de l'article. Enfin, nous suppromons la colonne « new_keywords » qui étaient la liste de nos mots clefs par article. Nous avons donc un dataframe contenant nos mots clefs d'origine et ceux généré par les abstracts sous format one hot encoding. Nous obtenons donc ceci :

```
x = x.drop("new_keywords",axis = 1)
x.head()
```

	keyword0	keyword1	keyword2	keyword3	keyword4	keyword5	keyword6	keyword7	keyword8
0	1	1	1	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

Enfin, nous séparons notre x et notre y en train et test avec une proportion de 80% - 20%, puis nous divisons le x_test et le y_test en deux avec une proportion de 50% - 50%. Nous avons donc 80% de nos données dans le train, 10% dans le test et 10% dans le jeu de validation. Nous utilisons la fonction de sklearn pour cela :

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20, random_state = 0)

print("Dimension x_train :", x_train.shape)
print("Dimension of y_train :", y_train.shape)
print("Dimension of x_test :", x_test.shape)
print("Dimension of y_test :", y_test.shape)
```

```
Dimension x_train : (1235, 5859)
Dimension of y_train : (1235,)
Dimension of x_test : (309, 5859)
Dimension of y_test : (309,)
```

```
x_test, x_valid, y_test, y_valid = train_test_split(x_test, y_test, test_size = 0.50, random_state = 0)

print("Dimension of x_test :", x_test.shape)
print("Dimension of y_test :", y_test.shape)
print("Dimension of x_valid :", x_valid.shape)
print("Dimension of y_valid :", y_valid.shape)
```

```
Dimension of x_test : (154, 5859)
Dimension of y_test : (154,)
Dimension of x_valid : (155, 5859)
Dimension of y_valid : (155,)
```

Suite à cette séparation nous avons voulu vérifiés que la répartition des dates est égale, nous avons donc affichés les dates présente dans les 3 set de données

```
['1997', '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018', '2019']
['1997', '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018', '2019']
['1998', '1999', '2000', '2001', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2018', '2019']
```

La répartition des dates est plutôt bonne entre les 3 set, il y a cependant certaines dates qui ne sont pas présente dans le test_set , cela est dû au fait que certaines dates ne sont représentées que quelques fois comme 1997, ou il n'y a que 3 articles qui sont sortis en 1997 dans notre dataset final.

Titres :

Nous avons voulu tester nos modèles avec plusieurs parties du dataset, la première partie sont les keywords des articles, la deuxieme partie du dataset que nous avons voulu exploiter sont les titres des articles.

Le préprocessing des titres est semblable à celui des keywords, c'est à dire faire un one hot encoding d'une liste de mots, nous avons donc refait les mêmes actions que sur la liste des keywords pour obtenir un dataframe sur lequel nos modèles pourront s'entrainer à prédire les daates des articles.

III) Clustering

Nous avons réutilisé le clusturing que nous avions fait pour le projet 1 dans le but d'utiliser ses résultats afin de créer un nouveau dataframe sur lequel entrainer nos modèles. La partie importante des résultats est la distance aux centroïdes des clusters de chaque article, que nous avons mis dans un dataframe pour ensuite les séparer en train set test set et validation set.

```python
from sklearn.cluster import KMeans

#clustering
num_clusters = 9
km = KMeans(n_clusters=num_clusters)
km.fit(X)
clusters = km.labels_.tolist()
```

Une fois le clustering fait voici le dataframe des distances aux centroïdes des clusters obtenus.

	0	1	2	3	4	5	6	7	8
0	0.496517	0.265010	0.392645	0.125162	0.367798	0.333952	0.322488	0.419676	0.480029
1	0.496517	0.265010	0.392645	0.125162	0.367798	0.333952	0.322488	0.419676	0.480029
2	0.496517	0.265010	0.392645	0.125162	0.367798	0.333952	0.322488	0.419676	0.480029
3	0.496517	0.265010	0.392645	0.125162	0.367798	0.333952	0.322488	0.419676	0.480029
4	1.116481	1.034519	1.060427	1.004378	1.030890	1.054288	1.050713	1.084494	1.106728
...	...	...	...	...	...	...	...	...	...
1539	0.931407	1.024251	1.064635	0.995747	1.058736	1.034301	1.041023	1.023409	1.100938
1540	1.096402	1.017976	1.057467	0.983889	1.057579	1.037703	1.039635	1.068548	1.062135
1541	1.107663	1.034519	1.070758	1.004441	1.048113	0.783429	1.042980	1.078282	1.097451
1542	1.111727	1.032199	1.072989	1.004298	1.059423	0.967605	1.047688	1.076926	1.102432
1543	1.116481	1.031506	1.074323	0.995770	1.063477	1.053084	1.050191	1.078563	1.107605

IV) Prédiction

1) Sur les keywords :

Tout d'abord nous nous intéressons à l'application de modèles sur le one-hot encoding de nos keywords. Nous appliquons alors un premier modèle classique, celui du Random Forest, comme ce qui suit :

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_squared_error

model1 = RandomForestClassifier(n_estimators=100, max_depth=None, min_samples_split=2, min_samples_leaf= 1, max_leaf_nodes=None)
model1.fit(x_train, y_train)

y_pred1 = model1.predict(x_valid)


print("Training Accuracy :", model1.score(x_train, y_train))
print("Validating Accuracy :", model1.score(x_valid, y_valid))
print("MSE :", mean_squared_error(y_validd, y_pred1))
```

```
Training Accuracy : 0.9959514170040485
Validating Accuracy : 0.0967741935483871
MSE : 83.49677419354839
```

Il est important de noter que l'accuracy n'est pas une métrique représentative dans notre cas, en effet, une erreur d'un an est tout à fait acceptable (un article de décembre 2009 est plus proche d'un article de janvier 2010 qu'un article de décembre 2010). Nous regardons donc aussi la MSE comme valeur comme suggéré par le sujet, et nous avons créé une nouvelle métrique qui nous est propre. On obtient alors la proportion de date bien prédite à un seuil de tolérance près. Elle prend en argument le seuil de tolérance (nombre d'année en valeur absolue acceptable) ainsi que les prédictions et les valeurs réelles des dates comme ce qui suit :

```
def accuracy_adaptee(n, y_pred, y_test_list):
  cmpt = 0
  for i in range (0, len(y_pred)):
    if abs(int(y_pred[i])-int(y_test_list[i])) < (n+1) :
      cmpt +=1
  return cmpt/len(y_pred)
```

Pour le Random Forest Classifier nous avons donc le résultat suivant :

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred1, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred1, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred1, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred1, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred1, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.2645161290322581
Taux de réponse juste à trois ans près : 0.3419354838709677
Taux de réponse juste à cinq ans près : 0.45806451612903226
Taux de réponse juste à sept ans près : 0.5741935483870968
Taux de réponse juste à dix ans près : 0.7354838709677419
```

Ce résultat n'est pas vraiment un bon résultat. Nous avons donc décidé d'essayer d'améliorer nos paramètres. Nous avons donc fait varier différents paramètres afin de voir quand est-ce que la MSE est minimale. Nous commençons par faire varier le nombre d'arbre de la forêt ainsi que le nombre minimal d'échantillon pour séparer un noeud. Nous obtenons donc ce résultat comme étant le meilleur, qui est déjà celui des paramètres précédents :

```
n : 100
m : 2
MSE : 78.15483870967742
```

Ce résultat n'est toujours pas précis. Nous avons donc décidé de tester un autre algorithme.

Nous avons donc appliqué le Support Vector Classifier à nos données, de la même manière. Nous l'avons choisi car bien que souvent cité, nous ne l'avions pas encore appliqué.

```
from sklearn.svm import SVC

model2 = SVC(C= 1.0, tol =0.0001)
model2.fit(x_train, y_train)

y_pred2 = model2.predict(x_valid)

print("Training Accuracy :", model2.score(x_train, y_train))
print("Testing Accuracy :", model2.score(x_valid, y_valid))
print("MSE :", mean_squared_error(y_validd, y_pred2))
```

```
Training Accuracy : 0.9060728744939271
Testing Accuracy : 0.04516129032258064
MSE : 33.26451612903226
```

Nous pouvons déjà constater que la MSE est très inférieure à ce que nous obtenions avec le RandomForest, ce qui est signe que ce modèle est nettement plus précis. Nous pouvons alors regarder les résultats avec la métrique que nous avons adaptée :

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred2, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred2, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred2, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred2, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred2, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.32903225806451614
Taux de réponse juste à trois ans près : 0.4258064516129032
Taux de réponse juste à cinq ans près : 0.6451612903225806
Taux de réponse juste à sept ans près : 0.7870967741935484
Taux de réponse juste à dix ans près : 0.9419354838709677
```

Nous pouvons donc voir qu'il y a bien moins d'erreurs de plus de 7 ans que précédemment par exemple.

De la même manière nous avons regardé si faire varier la tolérance de ce modèle avait une influence sur notre MSE. Nos tests nous suggèrent que les meilleurs résultats sont pour une tolérance de 0.001, voici les résultats que nous obtenons alors :

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred2, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred2, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred2, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred2, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred2, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.32903225806451614
Taux de réponse juste à trois ans près : 0.4258064516129032
Taux de réponse juste à cinq ans près : 0.6451612903225806
Taux de réponse juste à sept ans près : 0.7870967741935484
Taux de réponse juste à dix ans près : 0.9419354838709677
```

Il s'agit exactement des mêmes résultats que dans le teste pour t = 0.0001. Nous avons donc décidé d'essayer de nouvelles approches.

2) Sur les titres :

Nous avons envisagé de ne travailler que sur les titres pour voir si nous obtenions des différences significatives par rapport aux abstracts car avec les abstracts nous avions plus de 5000 mots

clefs différents. Ce grand nombre de mots clefs ne semble pas être profitable à nos algorithmes de prédiction. Nous appliquons nos deux modèles avec les variations de paramètres comme vu ci-dessus à nos titres déjà traités par le pré-processing décrit précédemment. Voici nos résultats pour le RandomForest avant optimisation de paramètres :

```
print("Training Accuracy :", model5.score(x_train2, y_train2))
print("Validating Accuracy :", model5.score(x_valid2, y_valid2))
print("MSE :", mean_squared_error(y_validd, y_pred5))
```

```
Training Accuracy : 0.9983805668016195
Validating Accuracy : 0.07096774193548387
MSE : 52.83870967741935
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred5, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred5, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred5, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred5, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred5, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.2064516129032258
Taux de réponse juste à trois ans près : 0.3096774193548387
Taux de réponse juste à cinq ans près : 0.5548387096774193
Taux de réponse juste à sept ans près : 0.7032258064516129
Taux de réponse juste à dix ans près : 0.8516129032258064
```

Les résultats obtenus montre que l'algorithme a plus d'erreur de l'ordre de cinq ans, mais moins de l'ordre de dix ans. Ainsi, il n'est pas plus précis dans le sens où il y a moins de réponses justes à deux ou trois ans près, mais les erreurs semblent plus « rapprochées ». Et voici le résultat obtenu après l'optimisation de nos paramètres :

```
print("n :", meilleur_n)
print("m :", meilleur_m)
print("MSE :", meilleure_MSE)
```

```
n : 50
m : 2
MSE : 46.92903225806452
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred5, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred5, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred5, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred5, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred5, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.25806451612903225
Taux de réponse juste à trois ans près : 0.36774193548387096
Taux de réponse juste à cinq ans près : 0.567741935483871
Taux de réponse juste à sept ans près : 0.6967741935483871
Taux de réponse juste à dix ans près : 0.8258064516129032
```

De la même manière que précédemment nous avons choisi d'appliquer l'algorithme SVC, afin de voir si notre modèle est plus précis. Voici nos résultats avant l'optimisation de nos paramètres :

```
print("Training Accuracy :", model6.score(x_train2, y_train2))
print("Validating Accuracy :", model6.score(x_valid2, y_valid2))
print("MSE :", mean_squared_error(y_validd, y_pred6))
```

```
Training Accuracy : 0.2242914979757085
Validating Accuracy : 0.04516129032258064
MSE : 42.07741935483871
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred6, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred6, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred6, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred6, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred6, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.3032258064516129
Taux de réponse juste à trois ans près : 0.3741935483870968
Taux de réponse juste à cinq ans près : 0.6064516129032258
Taux de réponse juste à sept ans près : 0.7354838709677419
Taux de réponse juste à dix ans près : 0.8838709677419355
```

Sur ce modèle, nous constatons que les résultats sont moins bons que ceux obtenus pour les keywords, alors que cette approche était meilleure sur le RandomForest. Lorsque l'on cherche à améliorer la performance, on retombe sur le même seuil de tolérance.

3) Sur les distances aux centroïdes des clusters :

Nous avons décidé de regarder la distance de chaque article aux centroïdes de nos clusters. En effet cela nous permet d'avoir une distance à un groupe de mots clefs, et donc de réduire le nombre de mots clefs à traiter. Nous passons de plus de 5000 mots clefs uniques à une distance à 9 groupes de mots clefs. Nous appliquons donc nos deux modèles avec les variations de paramètres comme vu ci-dessus à nos distances aux centroïdes des clusters obtenus précédemment. Voici nos résultats pour le RandomForest avant optimisation de paramètres :

```
print("Training Accuracy :", model3.score(x_train3, y_train3))
print("Validating Accuracy :", model3.score(x_valid3, y_valid3))
print("MSE :", mean_squared_error(y_validd, y_pred3))
```

```
Training Accuracy : 0.977327935222672
Validating Accuracy : 0.09032258064516129
MSE : 35.27741935483871
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred3, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred3, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred3, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred3, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred3, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.36774193548387096
Taux de réponse juste à trois ans près : 0.5225806451612903
Taux de réponse juste à cinq ans près : 0.6709677419354839
Taux de réponse juste à sept ans près : 0.7741935483870968
Taux de réponse juste à dix ans près : 0.9161290322580645
```

Lorsque nous souhaitons améliorer nos performances, nous trouvons des paramètres plus optimisés, avec 175 arbres dans la forêt et 5 données minimum pour séparer un nœud. Voici les résultats que nous obtenons :

```
print("n :", meilleur_n)
print("m :", meilleur_m)
print("MSE :", meilleure_MSE)
```

```
n : 175
m : 5
MSE : 29.81290322580645
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred3, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred3, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred3, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred3, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred3, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.3741935483870968
Taux de réponse juste à trois ans près : 0.4967741935483871
Taux de réponse juste à cinq ans près : 0.6645161290322581
Taux de réponse juste à sept ans près : 0.7677419354838709
Taux de réponse juste à dix ans près : 0.9161290322580645
```

Ce modèle est plus précis à deux ans près, mais il est moins précis de trois à sept ans près. Enfin, il redevient aussi précis que le modèle précédemment entrainé à dix ans près.

Appliquons maintenant le SVC. En voici les résultats :

```
print("Training Accuracy :", model4.score(x_train3, y_train3))
print("Validating Accuracy :", model4.score(x_valid3, y_valid3))
print("MSE :", mean_squared_error(y_validd, y_pred4))
```

```
Training Accuracy : 0.09473684210526316
Validating Accuracy : 0.03870967741935484
MSE : 26.767741935483873
```

```
print("Taux de réponse juste à deux ans près : " + str(accuracy_adaptee(2, y_pred4, y_valid_list)))
print("Taux de réponse juste à trois ans près : " + str(accuracy_adaptee(3, y_pred4, y_valid_list)))
print("Taux de réponse juste à cinq ans près : " + str(accuracy_adaptee(5, y_pred4, y_valid_list)))
print("Taux de réponse juste à sept ans près : " + str(accuracy_adaptee(7, y_pred4, y_valid_list)))
print("Taux de réponse juste à dix ans près : " + str(accuracy_adaptee(10, y_pred4, y_valid_list)))
```

```
Taux de réponse juste à deux ans près : 0.32903225806451614
Taux de réponse juste à trois ans près : 0.45161290322580644
Taux de réponse juste à cinq ans près : 0.7161290322580646
Taux de réponse juste à sept ans près : 0.8258064516129032
Taux de réponse juste à dix ans près : 0.9741935483870968
```

Notre MSE est inférieure à toutes les MSE obtenues jusqu'ici. Notre modèle n'est pas le plus juste à deux et trois ans près, mais dans 71.6% des cas, il prédit la bonne date à cinq ans près, ce qui est mieux que tous nos autres modèles. Nous avons regardé pour améliorer les performances en changeant les paramètres, mais nous n'obtenons rien de mieux.

Damien ALOUGES
Cécile AMSALLEM

Conclusion

Nous avons entrainé plusieurs modèles sur plusieurs données différentes et nous avons obtenus des résultats différents. Cependant, le modèle qui fait le moins d'erreur à deux ans près est le RandomForest appliqué aux distances aux centroïdes des clusters obtenus sur les mots-clefs, et ce n'est pas le même que celui qui est le plus juste à cinq en près. Ainsi nous ne pouvons déterminer quel modèle est dans l'absolu « le meilleur », cela dépend de l'objectif fixé : une erreur d'un an, de deux ou de quatre ?

Nos modèles sont les plus pertinents restent ceux qui ont été optimisé sur les distances aux centroïdes des clusters, les features sont moins nombreuses, mais elles comportent plus d'information que nos keywords en one hot encoding par exemple.

Un approfondissement de notre approche aurait été de tester ces modèles sur différentes étapes de clustering en faisant notamment varier le nombre de cluster par exemple.