Repositionnement des médicaments

Cédric MARTINEZ, Roman RIEUNIER

Janvier 2020

Introduction

Le repositionnement de médicaments est une problématique importante dans le corps médical. En effet une fois l'étude préclinique d'un nouveau médicament réalisée, il peut se passer beaucoup de temps avant qu'il soit mis sur le marché car il est nécessaire d'analyser tous les effets secondaires qu'il est susceptible de provoquer.

Le repositionnement de médicament permet alors de gagner du temps en permettant de réutiliser un médicament déjà mis sur le marché pour une nouvelle utilisation. Cette méthode s'appuie alors sur l'ensemble des bases de connaissances qu'il est possible d'avoir sur les médicaments.

Cependant cette mise en commun des bases de connaissance est limitée par la diversité des format dans lesquels sont enregistrées ces informations. Le cadre de notre projet était alors de réussir, grâce à des méthodes de Machine Learning, d'automatiser la mise en commun d'informations provenant de bases de données différentes puis de créer un algorithme capable de prédire les repositionnements possibles des médicaments.

1 Mise en commun des bases de connaissances

1.1 Traitement des données

La première étape de notre projet est de recenser l'ensemble des bases de données que nous avons a disposition ainsi que de prévoir les format qui pourront être attendues par de nouvelles bases de données.

Par conséquent il est important de prévoir un algorithme qui permet à la fois d'analyser des données *xml* comme dans la base de données de DrugBank ou des données *sdf* comme dans DrugCentral.

1.1.1 Conversion *sdf* en *csv*

Nous avons donc décidé pour commencer de faire une recherche des différentes bibliothèques *Python* permettant de traiter les données qui nous intéresse. Le format *Structure Data File* ou *sdf* developpé par Elsevier Molecular Design Limited est utilisé pour décrire des structures moléculaires en 3 dimensions. La bibliothèque rdkit permet de traiter facilement des données *sdf* en les stockant dans un *DataFrame* et en les convertissant éventuellement en *csv* comme dans le code ci-dessous.

Le format *csv* est intéressant car il peut être facilement adapté en *rdf* comme on le verra dans la suite de ce rapport.

```
import pandas as pd
from rdkit import Chem
from rdkit.Chem import PandasTools


my_sdf_file = './structures.molV2.sdf'

df = PandasTools.LoadSDF(my_sdf_file,
                         smilesName='SMILES',
                         molColName='Molecule',
                         includeFingerprints=False)

df.to_csv('./structures.molV2.csv')
```

1.1.2 Conversion *xml* en *csv*

La bibliothèque ElementTree permet de charger et traiter des fichiers au format *xml*. On peut ainsi en extraire de l'information et l'exporter sous un autre format tel que le *csv*, importable dans des triplestores populaires comme Neo4J. Cette librairie est essentielle dans notre contexte puisque la grande majorité des données mises à disposition publiquement le sont en *xml*. Le jeu de données sur lequel nous travaillons principalement disponible sur DrugBank est justement au format *xml* et recense des informations sur plus de 6800 médicaments dont

1500 approuvés par la FDA.

TODO

```python
import xml.etree.ElementTree as ET
import pandas as pd

root = ET.parse('./full database.xml').getroot()
ns = '{http://www.drugbank.ca}'

drugs = []
for drug in root.findall(f"{ns}drug"):
    for product in drug.find(f'{ns}products').findall(f'{ns}product'):
        drugs.append({"Name": drug.find(f"{ns}name").text, "Labeller": drug.find(f"{ns}description").text, "Countries": product.find(f"{ns}country").text, "Drug": product.find(f"{ns}name").text})

pd.DataFrame(drugs).to_csv("drugbank.csv", sep=";")
```

1.2 Stockage des données

Une fois les données au format *csv* nous pouvons les stocker sous forme de graphe au format *rdf* grâce au logiciel Neo4J.

En effet Neo4J, permet de charger des fichiers *csv* et de stocker les informations de ces fichiers dans un graphe de connaissances grâce à des commandes en langage Cypher Query développé exclusivement pour le logiciel. Pour charger un fichier *csv* il suffit alors d'utiliser la commande :

```
LOAD CSV FROM "file:///E:/pfe_data/drugs.csv"

// "file:///" va cherche le fichier dans le répertoire
// import/ du dossier de la base de données associée à notre
// projet Neo4J

// Sur mon ordinateur le fichier drugs.csv est placé dans le
// répertoire "  E:\
//              neo4jDatabases\
//              database-fce6df9e-3f23-4c35-bc7b-5609a7078b86\
//              installation-3.5.14\import\
//              pfe_data"

// LOAD CSV FROM n'est pas utilisable seule
```

Dans notre version du logiciel nous récupérons les données de DrugBank sous la forme de 2 fichiers *csv*.

drugs.csv qui contient les champs :

Name	*Direct_parent*	*Kingdom*	*Superclass*
Lepirudin	Peptides	Organic Compounds	Organic Acids
...	...	...	...
Class	*Subclass*	*Synonyms*	*Description*
Carboxylic Acids and Derivatives	Amino Acids, Peptides, and Analogues	Hirudin variant-1, Lepirudin recombinant	Lepirudin is identical ...
...	...	...	...

products.csv qui contient les champs :

Name	*Labellers*	*Drug*
Refludan	Celgene Europe Limited EU, Bayer US,Bayer Canada	Lepirudin
...	...	...

Pour cela on utilise un premier script qui permet de récupérer les N premières molécules de la base de données de *DrugBank*.

```
TOTAL_DRUG = 30 # Nombre de médicaments qu'on veut
                # récupérer depuis la base de données complète

file = open('./drugbank_all_full_database.xml/full database.xml',
            'r', encoding="utf8")
count = 0
drug_count = 0
drug_nb = 0

with open('./drugbank_all_full_database.xml/partial database.xml',
            'w', encoding="utf8") as sub_file:
    while drug_nb < TOTAL_DRUG:
        count += 1

        line = file.readline()

        if not line:
            break
        sub_file.write(line)

        if "<drug>" in line or "<drug " in line:
            drug_count += 1
        if "</drug>" in line:
            drug_count -= 1
            if drug_count == 0:
                drug_nb += 1
    sub_file.write('</drugbank>')
    sub_file.close()

file.close()
```

Puis on met ces données au format *csv* de manière à ce qu'elles puissent être facilement mise ensuite sous forme de graphe dans *Neo4J* et utilisées par un

algorithme de *Graph embedding* (dont on verra le fonctionnement dans la suite du rapport).

```
import xml.etree.ElementTree as ET
import pandas as pd

ns = '{http://www.drugbank.ca}'

root = ET.parse(
    './drugbank_all_full_database.xml/\partial database.xml'
    ).getroot()
drugs = []
products = []

# Transforme un tableau en string
# Utilisable facilement dans Neo4J
def a2str(array):
    str = ""
    for e in array:
            str = str + e + ","
    return str[:-1]

for drug in root.findall(f"{ns}drug"):
    # Drug.csv

    # Liste des attributs de classification
    d_parent = "No parent"
    kingdom = "No kingdom"
    superclass = "No super c"
    _class = "No class"
    subclass = "No sub c"
    if drug.find(f"{ns}classification") is not None:

        if drug.find(f"{ns}classification").find(
                    f"{ns}direct-parent") is not None:
            d_parent = drug.find(
                    f"{ns}classification").find(
                    f"{ns}direct-parent").text

        if drug.find(f"{ns}classification").find(
                    f"{ns}kingdom") is not None:
            kingdom = drug.find(
                    f"{ns}classification").find(
                    f"{ns}kingdom").text

        if drug.find(f"{ns}classification").find(
                    f"{ns}superclass") is not None:
            superclass = drug.find(
                    f"{ns}classification").find(
                    f"{ns}superclass").text

        if drug.find(f"{ns}classification").find(
                    f"{ns}class") is not None:
            _class = drug.find(
                    f"{ns}classification").find(
                    f"{ns}class").text
```

```
54
55          if drug.find(f"{ns}classification").find(
56                      f"{ns}subclass") is not None:
57              subclass = drug.find(
58                      f"{ns}classification").find(
59                      f"{ns}subclass").text
60
61      synonyms = []
62      for syn in drug.find(
63                      f'{ns}synonyms').findall(f'{ns}synonym'):
64          synonyms.append(syn.text)
65
66      drugs.append({"Name": drug.find(f"{ns}name").text,
67                  "Direct_parent": d_parent,
68                  "Kingdom": kingdom,
69                  "Superclass": superclass,
70                  "Class": _class,
71                  "Subclass": subclass,
72                  "Synonyms": a2str(synonyms),
73                  "Description": drug.find(
74                                  f"{ns}description").text})
75
76      # Products.csv
77      labellers = {}
78      for product in drug.find(
79                      f'{ns}products').findall(f'{ns}product'):
80          name = product.find(f"{ns}name").text
81
82          # On évite les doublons si les produits
83          # ont le même nom
84          if name not in labellers.keys():
85              labellers[name] = []
86
87          # On associe le labeller à son pays
88          # Si il y a plusieurs pays il y aura
89          # plusieurs noeuds ex : Bayer US / Bayer Canada
90          lab_str = product.find(f"{ns}labeller").text \
91                  + " " \
92                  + product.find(f"{ns}country").text
93          # On évite les doublons de labeller
94          if lab_str not in labellers[name]:
95              labellers[name].append(lab_str)
96
97      # On écris dans product.csv après avoir recensé tous
98      # les produits associés à une molécule pour éviter les
99      # doublons
100     for k in labellers: # k : nom du produit
101         products.append({"Name": k,
102                     "Labellers": a2str(labellers[k]),
103                     "Drug": drug.find(f"{ns}name").text})
104
105 pd.DataFrame(drugs).to_csv("E:/neo4jDatabases/database-fce6df9e-3
        f23-4c35-bc7b-5609a7078b86/installation-3.5.14/import/pfe_data/
        drugs.csv", sep=";")
106 pd.DataFrame(products).to_csv("E:/neo4jDatabases/database-fce6df9e
        -3f23-4c35-bc7b-5609a7078b86/installation-3.5.14/import/
        pfe_data/products.csv", sep=";")
```

Une fois qu'on a chargé le fichier *csv* on crée les entités de notre base de donnée. Etant donné que Neo4J ne stocke pas les valeurs *null* il faut également s'assurer que cela ne posera pas de problèmes :

```
LOAD CSV WITH HEADERS
FROM "file:///E:/pfe_data/drugs.csv" AS row
FIELDTERMINATOR ";"
WITH row WHERE row.Name IS NOT NULL
MERGE (m:Molecule {name: row.Name,
        description: coalesce(row.Description, "No description")
        })
MERGE (p:Molecule {name: coalesce(row.Direct_parent, "No parent")})
MERGE (m)-[:CHILD_OF]->(p)
MERGE (k:Kingdom {name: coalesce(row.Kingdom, "No kingdom")})
MERGE (m)-[:IN_KINGDOM]->(k)
MERGE (supc:Class {name: coalesce(row.Superclass, "No super c")})
MERGE (m)-[:IS_SUB_C]->(supc)
MERGE (c:Class {name: coalesce(row.Class, "No class")})
MERGE (m)-[:IS_CLASS]->(c)
MERGE (subc:Class {name: coalesce(row.Subclass, "No sub c")})
MERGE (m)-[:IS_SUPER_C]->(subc);

// Supprime les valeurs où le nom est null et
// met une description null à "No description"
```

Comme dans une base de données classique on peut également ajouter des contraintes pour éviter de future erreurs.

```
CREATE CONSTRAINT ON (d:Drug)
ASSERT d.name IS UNIQUE
```

```
CREATE CONSTRAINT ON (p:Product)
ASSERT p.name IS UNIQUE
```

```
CREATE CONSTRAINT ON (l:Labeller)
ASSERT l.name IS UNIQUE
```

Neo4J permet d'utiliser 2 commandes pour créer un nouveau noeud, **MERGE** ou **CREATE**, ici on préfère utiliser **MERGE** qui regarde d'abord si le noeud n'existe pas, au quel cas, il le crée.

Ensuite il faut préciser au logiciel les champs composés d'une liste d'items, par exemple ici les champs *Labellers* de *products.csv* et *Synonyms* de *drugs.csv*. On utilise alors le script ci-dessous.

```
LOAD CSV WITH HEADERS
FROM "file:///E:/pfe_data/products.csv" AS row
FIELDTERMINATOR ";"
WITH row WHERE row.Name IS NOT NULL
WITH row, split(row.Labellers, ',') AS labs
```

```
UNWIND labs AS lab
MERGE (p:Product {name: row.Name})
MERGE (l:Labeller {name: lab})
MERGE (p)-[:LABELLED_BY]->(l);
```

```
LOAD CSV WITH HEADERS
FROM "file:///E:/pfe_data/drugs.csv" AS row
FIELDTERMINATOR ";"
WITH row WHERE row.Name IS NOT NULL
WITH row, split(row.Synonyms, ',') AS syns
UNWIND syns AS syn
MERGE (m:Molecule {name: row.Name})
MERGE (s:Molecule {name: syn})
MERGE (m)-[:IS_SYNONYM]->(s)
MERGE (s)-[:IS_SYNONYM]->(m);
```

Enfin on relie les produits au médicaments grâce au champ *Drug* de *products.csv* :

```
LOAD CSV WITH HEADERS
FROM "file:///E:/pfe_data/products.csv" AS row
FIELDTERMINATOR ";"
MATCH (p:Product {name: row.Name})
MATCH (m:Molecule {name: row.Drug})
MERGE (p)-[:CONTAINS]->(m);
```

Enfin, pour visualiser notre graphe, on peut utiliser la commande **MATCH** qui sélectionne des éléments du graphe.

```
MATCH (e) RETURN e
// Affiche la totalité du graphe
```

```
MATCH e=()-[r:CONTAINS]->(m:Molecule)
WHERE m.name="Lepirudin"
RETURN e
// Affiche uniquement les éléments dans une relation CONTAINS
// qui cible un noeud Lepirudin
```

Une fois ces scripts exécutés notre graphe de connaissance contient suffisamment de données pour pouvoir effectuer des prédictions de repositionnemt. Le but de notre projet est donc maintenant de trouver un moyen, en utilisant des méthodes d'intelligence artificielle, de faire ces prédictions de manière automatique.

2 Prédictions de repositionnements

Maintenant que nous avons un graphe de connaissance contenant toutes les données dont nous avons besoin nous pouvons y appliquer un algorithme qui détecte automatiquement les repositionnements possibles en utilisant des méthodes de *Machine Learning*.

2.1 Plongement de mots

La méthode d'IA principale que nous avons utilisée dans notre projet est le Graph embedding.

Cette méthode repose sur une autre méthode, le plongement de mots ("Word embedding" en anglais), qui permet d'analyser des textes et de trouver les mots avec le plus de similitudes. Pour comprendre son fonctionnement nous avons regardé la bibliothèque Word2vec utilisée pour faire du plongement de mots.

L'idée principale de Word2Vec est de représenter les mots par des vecteurs qui sont similaires si les mots on la même utilisation dans le texte utilisé pour l'entraînement du réseau de neurones, c'est à dire si leur entourage est similaire. Dans le code suivant :

```
data = [
    ["paracétamol", "molécule", "antalgique",
    "antipyrétique", "médicament"],

    ["aspirine", "molécule", "antalgique",
    "antipyrétique", "médicament"],

    ["estazolam", "molécule", "anxiolytique",
    "hypnotique", "sédatif"],
]

from gensim.test.utils import common_texts, get_tmpfile
from gensim.models import Word2Vec

model = Word2Vec(data,
            min_count=1,# Nombre minimale d'occurrences
                        # du mot pour qu'il soit pris en compte
            size= 10,   # Nombre de dimension dans le
                        # vecteur de similarités
            workers=3,  # Nombre de partitions lors de
                        # l'entraînement du modèle
            window =5,  # Nombre de mots avant et après pour
                        # établir l'entourage
            sg = 0)     # Algorithme d'entraînement
                        # CBOW(0) ou Skip-Gram(1)
```

On donne en entrée au modèle *Word2Vec* des descriptions des molécules de paracétamol, aspirine et estazolam par mots clés. Le modèle va attribuer un

vecteur de dimension 10 à chaque mot et les mots ayant un sens proche auront également un vecteur proche. La bibliothèque permet ensuite d'observer les similitudes entre les mots entre 100% et -100% et on obtient après entraînement les résultats suivants :

```
model.similarity('paracétamol', 'aspirine')
# 24%

model.similarity('paracétamol', 'estazolam')
# -19%

model.similarity('estazolam', 'aspirine')
# -42%
```

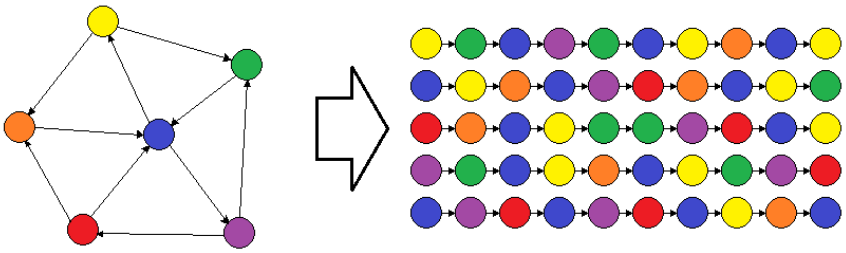
Par conséquent le modèle a bien compris que le paracétamol et l'aspirine sont plus proches entre eux que de l'estazolam qui est un somnifère.

2.2 Plongement sur des graphes (Graph embedding)

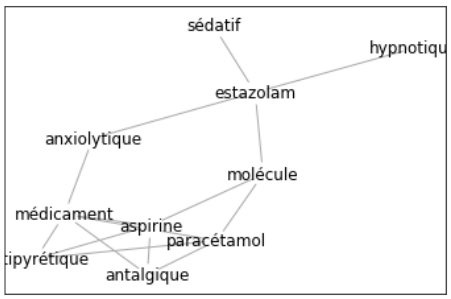
2.2.1 Graph Neural Network (GNN)

Ce même principe peut alors être appliqué sur des données sous forme de graphe comme le fait notamment la bibliothèque Node2vec.

Pour cela il faut considérer les noeuds comme des mots et les connections comme l'entourage du noeud. Pour obtenir les vecteurs attendus par l'algorithe *Skip-Gram* il suffit alors de parcourir le graphe et d'enregistrer les positions successives comme dans la figure ci-dessous :



Sous forme de graphe, les données que nous avons utilisées précédemment prendraient environ la même forme que sur le graphe de droite.



A partir de ce format de données il est possible d'utiliser la bibliothèque *Node2vec* et ainsi retrouver les similitudes entre les mots.

Pour cela on exécute le script suivant :

```
from node2vec import Node2Vec

node2vec = Node2Vec(graph, dimensions=10, workers=3,
                    num_walks=100,  # Nombre de parcours du graphe
                    walk_length=50) # Taille des parcours

model = node2vec.fit(window=5, min_count=1, batch_words=4)
```

Comme avec *Word2vec* on peut regarder la similitude entre paracétamol, aspirine et estazolam, et vérifier que le paracétamol est plus proche de l'aspirine que de l'estazolam pour être sûr que le modèle marche :

```
# Avec walk_length=10
model.wv.most_similar('paracétamol')
# Aspirine :    99%
# Estazolam :   96%

# Avec walk_length=20
model.wv.most_similar('paracétamol')
# Aspirine :    100%
# Estazolam :   43%

# Avec walk_length=50
model.wv.most_similar('paracétamol')
# Aspirine :    99%
# Estazolam :   -7%
```

Comme on peut le voir le modèle fonctionne très bien et on peut également remarquer qu'augmenter la longueur des parcours augmente significativement la précision du modèle.

Cette caractéristique est importante car elle veut dire que même avec un très petit jeu de données on peut avoir un modèle performant et simulant une quantité importante de parcours comme données d'entraînement ce qu'il n'était pas possible de faire avec les données sous forme de tableaux.

2.2.2 Visualisation t-SNE

Enfin, pour mieux visualiser les similitudes entre les mots nous pouvons utiliser l'algorithme t-SNE.

Premièrement on peut afficher l'intégralité des mots sur un repère 2 dimensions tout en conservant les similitudes. Pour cela le code suivant récupère respectivement dans els tableaux *word_cluster* et *embedding_cluster* les différents mots du vocabulaire ainsi que leurs plongements.

```python
word_cluster = []
embedding_cluster = []

for word in model.wv.vocab:
    word_cluster.append(word)
    embedding_cluster.append(model[word])
```

Ces tableaux sont ensuite passés à un modèle de *t-SNE* qui va fournir une représentation en 2 dimensions des similitudes entre les mots :

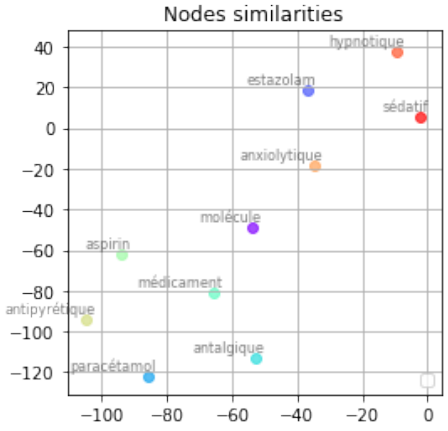
```python
from sklearn.manifold import TSNE
import numpy as np

embedding_cluster = np.array(embedding_cluster)
n, m = embedding_cluster.shape
tsne_model_en_2d = TSNE(
    perplexity=6,    # Valeur relié à la taille des données
                     # d'entrées, normalement entre 5 et 50,
                     # ici 6 car le graphe utilisé comme
                     # exemple est très petit
    n_components=2, # Dimension attendue en résultat
    init='pca',      # PCA ou aléatoire, PCA est plus stable
    n_iter=3500,     # Nombre d'itérations pour
                     # l'optimisation du résultat
    random_state=32)# Graine pour le facteur aléatoire

embeddings_en_2d = np.array(tsne_model_en_2d.fit_transform(
    embedding_cluster.reshape(n, m))).reshape(n, 2)
```

Enfin le code ci-dessous permet d'afficher le résultat obtenu en utilisant la bibliothèque *matplotlib* et on obtient alors la figure de droite.

On peut voir que les mots sont bien globalement placés en fonction de leur similitudes avec, d'un coté, le groupe lié à l'estazolam et de l'autre le groupe lié au paracétamol et à l'aspirine.

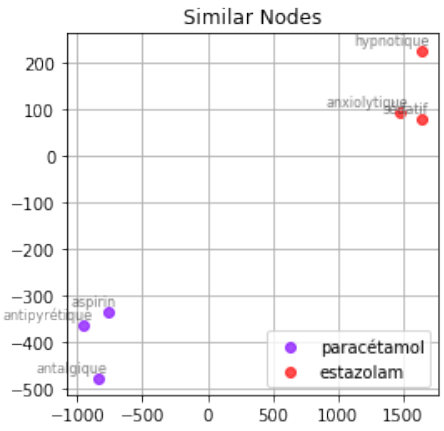



```python
import matplotlib.pyplot as plt
import matplotlib.cm as cm

def tsne_plot_words(title, embedding_cluster, word_cluster,
                    a, filename=None):
    plt.figure(figsize=(4, 4))
    colors = cm.rainbow(np.linspace(0, 1, len(word_cluster)))
```

```
8
9       for i in range(len(word_cluster)):
10          x = embedding_cluster[i][0]
11          y = embedding_cluster[i][1]
12          plt.scatter(x, y, c=colors[i], alpha=a)
13          plt.annotate(word_cluster[i], alpha=0.5, xy=(x, y),
14                       xytext=(5, 2), textcoords='offset points',
15                       ha='right', va='bottom', size=8)
16      plt.legend(loc=4)
17      plt.title(title)
18      plt.grid(True)
19      if filename:
20          plt.savefig(filename, format='png', dpi=150,
21                      bbox_inches='tight')
22      plt.show()
23
24
25  tsne_plot_words('Nodes similarities', embeddings_en_2d,
        word_cluster, 0.7,
26                      'nodes_similarities.png')
```

En modifiant un peu le code précédent on peut également obtenir une représentation qui montre pour les mots qu'on a choisi, ici paracétamol et estazolam, un nuage de point correspondant aux mots avec le sens le plus proche. Cette méthode est pratique lorsqu'on a énormement de mots et qu'on veut en sélectionner une partie.



L'argument "topn=3" à la ligne 8 permet ici de dire combient de mot similaires doivent être affichés sur le graphique pour chaque mot clé.

```
1   keys = ['paracétamol', 'estazolam']
2
3   embedding_clusters = []
4   word_clusters = []
5   for word in keys:
6       embeddings = []
7       words = []
8       for similar_word, _ in model.most_similar(word, topn=3):
9           words.append(similar_word)
10          embeddings.append(model[similar_word])
11      embedding_clusters.append(embeddings)
12      word_clusters.append(words)
13
14  from sklearn.manifold import TSNE
15  import numpy as np
16
```

```
embedding_clusters = np.array(embedding_clusters)
n, m, k = embedding_clusters.shape
tsne_model_en_2d = TSNE(
    perplexity=2, n_components=2, init='pca',
    n_iter=3500, random_state=32)

embeddings_en_2d = np.array(tsne_model_en_2d.fit_transform(
                        embedding_clusters.reshape(n * m, k))
                    ).reshape(n, m, 2)

import matplotlib.pyplot as plt
import matplotlib.cm as cm

def tsne_plot_similar_words(title, labels, embedding_clusters,
                            word_clusters, a, filename=None):
    plt.figure(figsize=(4, 4))
    colors = cm.rainbow(np.linspace(0, 1, len(labels)))
    for label, embeddings, words, color
        in zip(labels, embedding_clusters, word_clusters, colors):

        x = embeddings[:, 0]
        y = embeddings[:, 1]
        plt.scatter(x, y, c=color, alpha=a, label=label)
        for i, word in enumerate(words):
            plt.annotate(word, alpha=0.5, xy=(x[i], y[i]),
                         xytext=(5, 2), textcoords='offset points',
                         ha='right', va='bottom', size=8)
    plt.legend(loc=4)
    plt.title(title)
    plt.grid(True)
    if filename:
        plt.savefig(filename, format='png',
                    dpi=150, bbox_inches='tight')
    plt.show()


tsne_plot_similar_words('Similar Nodes', keys, embeddings_en_2d,
                        word_clusters, 0.7, 'similar_nodes.png')
```

2.3 Application à notre base de données

Nous savons maintenant comment calculer les similitudes entre les noeuds de notre graphe de connaissance *Neo4J* vu au début de ce rapport. Le code ci-dessous permet d'exporter notre graphe au format *graphml* et ainsi d'utiliser la bibliothèque *Node2Vec* dessus.

```
CALL apoc.export.graphml.all("drug.graphml", {})

// Cette commande nécessite le plugin apoc qui peut
// être installer facilement depuis l'onglet Plugins
// de l'interface Neo4J Desktop
```

Améliorations

L'important maintenant est de mettre dans la base de données des informations qui permettrons ensuite à *Node2Vec* de déterminer les nouvelles applications des médicaments.
Actuellement la base de données ne permet pas vraiment de faire cela mais on observe déjà que le modèle reconnaît les molécules synonymes.

Une piste intéressante serait d'ajouter dans la base de données les applications des médicaments ainsi que les effets secondaires et toute autres propriétés qui pourraient permettre d'effectuer le repositionnement en observant directement les liens entre les médicaments, les maladies et les effets.

On peut également envisager de relier un médicament à des mots clés eux-même reliés entre eux. Pour cela on peut par exemple, à partir du champ *description* ne garder que les mots les plus rares dans la langue anglaise, ce qui enlèvera les mots comme "the", "a", "an" et conservera les mots comme "antalgesic". Ces mots seraient donc relié au médicament décrit puis à l'aide des modèle de plongements de mots on pourrait également relier ces mots clés entre eux. De plus le fait de relier les mots clés entre eux ne nécessite pas forcement de partir de données médicales, n'importe quel texte peut être donné à un modèle de plongement de mot pour reconnaître les mots similaires.
Le fait d'avoir un nuage de mot clé reliés entre eux et aux médicaments associés serait très bénéfique pour *Node2Vec* qui pourra alors facilement retrouver les médicaments similaires.

Conclusion

Ce projet nous a permis de mettre en pratique les méthodes de plongement de mots que nous avons vu en cours. Cependant étant donné que le sujet global du projet était difficile à appréhender et pas instinctivement corrélé à l'intelligence artificielle nous n'avons pas encore obtenu de résultats montrant qu'il est possible de faire du repositionnement de médicaments de façon automatisé.
Malgré tout, les résultats obtenus à la fin du projet vis à vis du traitement des données de *Drugbank* et de leur mise en place dans un graphe de connaissance sont encourageants et laisse à penser qu'il est effectivement possible d'utiliser les algorithmes de *Graph embedding* afin de trouver les repositionnements.