LAB 8 - Natural Language Processing

Auteur : Badr TAJINI - Introduction à l'IA - ESIEE-IT - 2024/2025

Dans ce chapitre, nous allons explorer le sujet passionnant du **traitement du langage naturel** (**NLP**). Comme nous l'avons discuté dans les chapitres précédents, disposer d'ordinateurs capables de comprendre le langage humain est l'une des avancées qui rendront véritablement les ordinateurs encore plus utiles. Le NLP fournit les bases pour commencer à comprendre comment cela pourrait être possible.

Nous discuterons et utiliserons divers concepts tels que la tokenisation, la racinisation (stemming) et la lemmatisation pour traiter le texte. Nous aborderons ensuite le modèle du **Sac de Mots** (**Bag of Words**) et comment l'utiliser pour classer le texte. Nous verrons comment utiliser l'apprentissage automatique pour analyser le sentiment d'une phrase. Nous discuterons ensuite de la modélisation thématique et implémenterons un système pour identifier les sujets dans un document donné.

À la fin de ce chapitre, vous serez familiarisé avec les sujets suivants :

- Installation des packages NLP pertinents
- Tokenisation des données textuelles
- Conversion des mots en leurs formes de base à l'aide de la racinisation (stemming)
- Conversion des mots en leurs formes de base à l'aide de la lemmatisation
- Division des données textuelles en segments
- Extraction d'une matrice document-terme en utilisant le modèle du Sac de Mots
- Construction d'un prédicteur de catégorie
- Construction d'un identificateur de genre
- Construction d'un analyseur de sentiments
- Modélisation thématique à l'aide de l'Allocation Dirichlet Latente

Introduction et installation des packages

Le **traitement du langage naturel** (**NLP**) est devenu une partie importante des systèmes modernes. Il est largement utilisé dans les moteurs de recherche, les interfaces conversationnelles, les processeurs de documents, etc. Les machines peuvent bien gérer les données structurées, mais lorsqu'il s'agit de travailler avec du texte libre, elles rencontrent des difficultés. L'objectif du NLP est de développer des algorithmes qui permettent aux ordinateurs de comprendre le texte libre et de les aider à comprendre le langage.

L'un des aspects les plus difficiles du traitement du langage naturel libre est la quantité de variation. Le contexte joue un rôle très important dans la manière dont une phrase est comprise. Les humains sont naturellement excellents pour comprendre le langage. Il n'est pas encore clair comment les humains comprennent le langage aussi facilement et intuitivement. Nous utilisons nos connaissances et expériences passées pour comprendre les conversations et nous pouvons rapidement saisir l'essentiel de ce dont les autres parlent, même avec peu de contexte explicite.

Pour résoudre ce problème, les chercheurs en NLP ont commencé à développer diverses applications en utilisant des approches d'apprentissage automatique. Pour construire de telles applications, un grand corpus de texte est obtenu, puis des algorithmes sont entraînés sur ces données pour effectuer diverses tâches telles que la catégorisation de texte, l'analyse des sentiments et la modélisation thématique. Les algorithmes sont entraînés à détecter des motifs dans les données textuelles d'entrée et à en tirer des informations pertinentes.

Dans ce chapitre, nous discuterons de divers concepts sous-jacents utilisés pour analyser le texte et construire des applications NLP. Cela nous permettra de comprendre comment extraire des informations significatives à partir des données textuelles fournies. Nous utiliserons un package Python appelé **Natural Language Toolkit** (**NLTK**) pour construire ces applications. Vous pouvez l'installer en exécutant la commande suivante :

```
pip install nltk
```

Vous pouvez en savoir plus sur NLTK à http://www.nltk.org.

Afin d'accéder à tous les ensembles de données fournis par NLTK, nous devons les télécharger. Ouvrez une console Python en tapant la commande suivante :

```
python
```

Nous sommes maintenant dans la console Python. Tapez ce qui suit pour télécharger les données :

```
import nltk
nltk.download()
```

Nous utiliserons également un package appelé `gensim` dans ce chapitre. `gensim` est une bibliothèque robuste de modélisation sémantique utile pour de nombreuses applications. Vous pouvez l'installer en exécutant la commande suivante :

```
pip install gensim
```

Vous pourriez avoir besoin d'un autre package, appelé `pattern`, pour que `gensim` fonctionne correctement. Vous pouvez l'installer en exécutant la commande suivante :

```
pip install pattern
```

Vous pouvez en savoir plus sur `gensim` à https://radimrehurek.com/gensim. Maintenant que vous avez installé NLTK et `gensim`, poursuivons la discussion.

Tokenisation des données textuelles

Lorsque nous travaillons avec du texte, nous devons le décomposer en morceaux plus petits pour l'analyse. Pour ce faire, nous pouvons appliquer la **tokenisation**. La tokenisation est le processus de division du texte en un ensemble de morceaux, tels que des mots ou des phrases. Ces morceaux sont appelés **tokens**. Selon ce que nous voulons faire, nous pouvons définir nos propres méthodes pour diviser le texte en de nombreux tokens. Voyons comment tokeniser le texte d'entrée en utilisant NLTK.

Créez un nouveau fichier Python et importez les packages suivants :

```
from nltk.tokenize import sent_tokenize, word_tokenize, WordPunctTokenizer
```

Définissez le texte d'entrée qui sera utilisé pour la tokenisation :

```
# Définir le texte d'entrée
input_text = "Do you know how tokenization works? It's actually quite interesting!
Let's analyze a couple of sentences and figure it out."
```

Divisez le texte d'entrée en tokens de phrases :

```
# Tokeniseur de phrases
print("\nTokeniseur de phrases :")
print(sent_tokenize(input_text))
```

Divisez le texte d'entrée en tokens de mots :

```
# Tokeniseur de mots
print("\nTokeniseur de mots :")
print(word_tokenize(input_text))
```

Divisez le texte d'entrée en tokens de mots en utilisant le tokeniseur `WordPunct` :

```
# Tokeniseur WordPunct
print("\nTokeniseur WordPunct :")
print(WordPunctTokenizer().tokenize(input_text))
```

Le code complet est disponible dans le fichier `tokenizer.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

```
Sentence tokenizer:
['Do you know how tokenization works?', "It's actually quite interesting!", "Let's analyze a couple of se
ntences and figure it out."]

Word tokenizer:
['Do', 'you', 'know', 'how', 'tokenization', 'works', '?', 'It', "'s", 'actually', 'quite', 'interesting'
, '!', 'Let', "'s", 'analyze', 'a', 'couple', 'of', 'sentences', 'and', 'figure', 'it', 'out', '.']

Word punct tokenizer:
['Do', 'you', 'know', 'how', 'tokenization', 'works', '?', 'It', "'", 's', 'actually', 'quite', 'interest
ing', '!', 'Let', "'", 's', 'analyze', 'a', 'couple', 'of', 'sentences', 'and', 'figure', 'it', 'out', '.
']
```

Figure 1 : Sortie des tokeniseurs

Le tokeniseur de phrases divise le texte d'entrée en phrases. Les deux tokeniseurs de mots se comportent différemment lorsqu'il s'agit de ponctuation. Par exemple, le mot "It's" est divisé différemment par le tokeniseur ponctuation que par le tokeniseur régulier.

Conversion des mots en leurs formes de base à l'aide de la racinisation (stemming)

Travailler avec du texte signifie gérer beaucoup de variations. Nous devons traiter les différentes formes du même mot et permettre à l'ordinateur de comprendre que ces mots différents ont la même forme de base. Par exemple, le mot `sing` peut apparaître sous de nombreuses formes, telles que *singer*, *singing*, *song*, *sung*, etc. Cet ensemble de mots partage des significations similaires. Ce processus est connu sous le nom de **stemming**. Le stemming est une méthode de production de variantes morphologiques d'un mot racine/de base. Les humains peuvent facilement identifier ces formes de base et en tirer un contexte.

Lors de l'analyse du texte, il est utile d'extraire ces formes de base. Cela permet d'extraire des statistiques utiles à partir du texte d'entrée. Le stemming est une façon d'y parvenir. L'objectif d'un stemmer est de réduire les mots de leurs différentes formes en une forme de base commune. C'est essentiellement un processus heuristique qui coupe les terminaisons des mots pour extraire leurs formes de base. Voyons comment le faire en utilisant NLTK.

Créez un nouveau fichier Python et importez les packages suivants :

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer
```

Définissez quelques mots d'entrée :

```
input_words = ['writing', 'calves', 'be', 'branded', 'horse', 'randomize',
              'possibly', 'provision', 'hospital', 'kept', 'scratchy', 'code']
```

Créez des objets pour les stemmers **Porter**, **Lancaster** et **Snowball** :

```
# Créer divers objets stemmers
porter = PorterStemmer()
lancaster = LancasterStemmer()
snowball = SnowballStemmer('english')
```

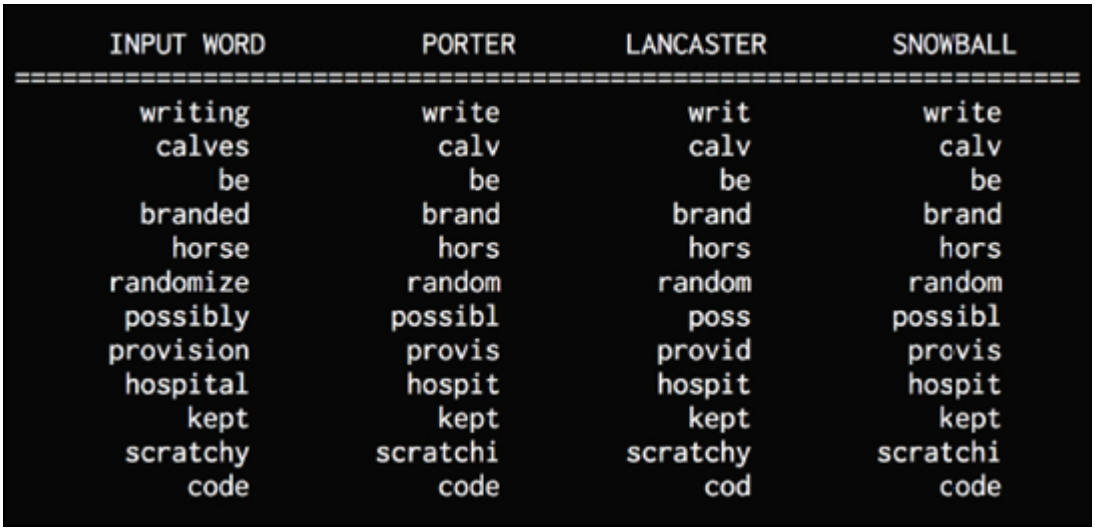
Créez une liste de noms pour l'affichage en tableau et formatez le texte de sortie en conséquence :

```
# Créer une liste de noms de stemmers pour l'affichage
stemmer_names = ['PORTER', 'LANCASTER', 'SNOWBALL']
formatted_text = '{:>16}' * (len(stemmer_names) + 1)
print('\n', formatted_text.format('MOT D\'ENTRÉE', *stemmer_names), '\n', '='*68)
```

Itérez à travers les mots et stemmez-les en utilisant les trois stemmers :

```
# Stem chaque mot et afficher la sortie
for word in input_words:
    output = [word, porter.stem(word), lancaster.stem(word), snowball.stem(word)]
    print(formatted_text.format(*output))
```

Le code complet est disponible dans le fichier `stemmer.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :



INPUT WORD	PORTER	LANCASTER	SNOWBALL
writing	write	writ	write
calves	calv	calv	calv
be	be	be	be
branded	brand	brand	brand
horse	hors	hors	hors
randomize	random	random	random
possibly	possibl	poss	possibl
provision	provis	provid	provis
hospital	hospit	hospit	hospit
kept	kept	kept	kept
scratchy	scratchi	scratchy	scratchi
code	code	cod	code

Figure 2 : Sortie des stemmers

Parlons des trois algorithmes de racinisation utilisés ici. Tous visent essentiellement le même objectif. La différence entre eux réside dans le niveau de rigueur utilisé pour arriver à la forme de base.

Le stemmer de Porter est le moins strict, et Lancaster est le plus strict. Si vous observez attentivement les sorties, vous remarquerez les différences. Les stemmers se comportent différemment avec des mots tels que `possibly` ou `provision`. Les sorties racinisées obtenues avec le stemmer Lancaster sont un peu obscurcies car il réduit beaucoup les mots. En même temps, l'algorithme est rapide. Une bonne règle empirique est d'utiliser le stemmer Snowball car il offre un bon compromis entre rapidité et rigueur.

Conversion des mots en leurs formes de base à l'aide de la lemmatisation

La **lemmatisation** est une autre méthode de réduction des mots à leurs formes de base. Dans la section précédente, nous avons vu que certaines des formes de base obtenues à partir des stemmers n'avaient pas de sens. La lemmatisation est le processus de regroupement des différentes formes fléchies d'un mot afin qu'elles puissent être analysées comme un seul élément. La lemmatisation est similaire au stemming, mais elle apporte du contexte aux mots. Ainsi, elle lie des mots ayant des significations similaires à un seul mot. Par exemple, tous les trois stemmers ont dit que la forme de base de *calves* est *calv*, ce qui n'est pas un vrai mot. La lemmatisation prend une approche plus structurée pour résoudre ce problème. Voici quelques exemples supplémentaires de lemmatisation :

- rocks : rock
- corpora : corpus

- worse : bad

Le processus de lemmatisation utilise l'analyse lexicale et morphologique des mots. Il obtient les formes de base en supprimant les terminaisons fléchies des mots telles que *ing* ou *ed*. Cette forme de base d'un mot est connue sous le nom de **lemme**. Si vous lemmatisez le mot *calves*, vous devriez obtenir *calf* en sortie. Une chose à noter est que la sortie dépend de savoir si le mot est un verbe ou un nom. Voyons comment faire cela avec NLTK.

Créez un nouveau fichier Python et importez les packages suivants :

```
from nltk.stem import WordNetLemmatizer
```

Définissez quelques mots d'entrée. Nous utiliserons le même ensemble de mots que dans la section précédente afin de pouvoir comparer les sorties :

```
input_words = ['writing', 'calves', 'be', 'branded', 'horse', 'randomize',
               'possibly', 'provision', 'hospital', 'kept', 'scratchy', 'code']
```

Créez un objet `lemmatizer` :

```
# Créer l'objet lemmatizer
lemmatizer = WordNetLemmatizer()
```

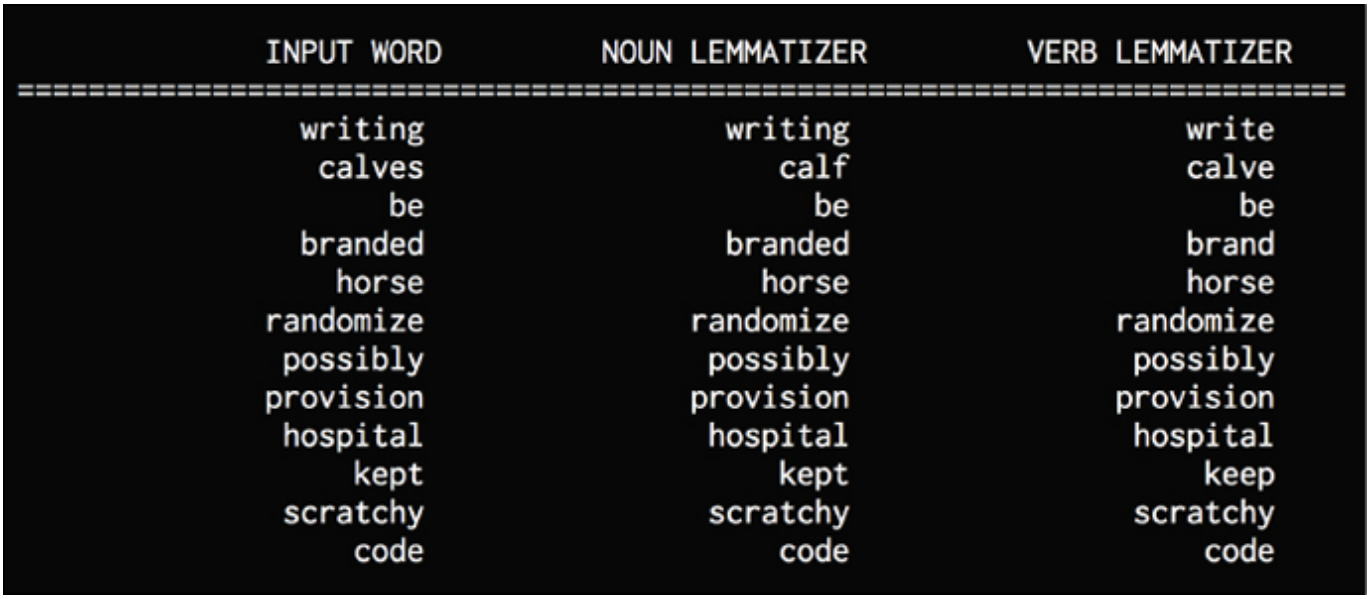
Créez une liste de noms de lemmatizers pour l'affichage en tableau et formatez le texte en conséquence :

```
# Créer une liste de noms de lemmatizers pour l'affichage
lemmatizer_names = ['LEMMATIZER NOMINAL', 'LEMMATIZER VERBAL']
formatted_text = '{:>24}' * (len(lemmatizer_names) + 1)
print('\n', formatted_text.format('MOT D\'ENTRÉE', *lemmatizer_names), '\n', '='*75)
```

Itérez à travers les mots et lemmatisez-les en utilisant les lemmatizers nominaux et verbaux :

```
# Lemmatizer chaque mot et afficher la sortie
for word in input_words:
    output = [word, lemmatizer.lemmatize(word, pos='n'),
              lemmatizer.lemmatize(word, pos='v')]
    print(formatted_text.format(*output))
```

Le code complet est disponible dans le fichier `lemmatizer.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :



INPUT WORD	NOUN LEMMATIZER	VERB LEMMATIZER
writing	writing	write
calves	calf	calve
be	be	be
branded	branded	brand
horse	horse	horse
randomize	randomize	randomize
possibly	possibly	possibly
provision	provision	provision
hospital	hospital	hospital
kept	kept	keep
scratchy	scratchy	scratchy
code	code	code

Figure 3 : Sortie des lemmatizers

Nous pouvons voir que le lemmatizer nominal fonctionne différemment du lemmatizer verbal lorsqu'il s'agit de mots tels que `writing` ou `calves`. Si vous comparez ces sorties aux sorties des stemmers, vous remarquerez également des différences. Les sorties des lemmatizers sont toutes significatives, tandis que les sorties des stemmers peuvent ou non être significatives.

Division des données textuelles en segments

Les données textuelles doivent généralement être divisées en morceaux pour une analyse ultérieure. Ce processus est connu sous le nom de **chunking**. Il est utilisé fréquemment dans l'analyse de texte. Les conditions utilisées pour diviser le texte en segments peuvent varier en fonction du problème à résoudre. Ce n'est pas la même chose que la tokenisation, où le texte est également divisé en morceaux. Lors du chunking, nous ne respectons aucune contrainte, sauf le fait que les segments de sortie doivent être significatifs.

Lorsque nous traitons de grands documents textuels, il devient important de diviser le texte en segments pour extraire des informations significatives. Dans cette section, nous verrons comment diviser le texte d'entrée en plusieurs morceaux.

Créez un nouveau fichier Python et importez les packages suivants :

```
import numpy as np
from nltk.corpus import brown
```

Définissez une fonction pour diviser le texte d'entrée en segments. Le premier paramètre est le texte, et le second paramètre est le nombre de mots dans chaque segment :

```
# Diviser le texte d'entrée en segments, où
# chaque segment contient N mots
def chunker(input_data, N):
    input_words = input_data.split(' ')
    output = []

    cur_chunk = []
    count = 0
    for word in input_words:
        cur_chunk.append(word)
        count += 1
        if count == N:
            output.append(' '.join(cur_chunk))
            count, cur_chunk = 0, []
    output.append(' '.join(cur_chunk))
    return output
```

Définissez la fonction `main` et lisez les données d'entrée en utilisant le corpus Brown. Nous lirons `12 000` mots dans ce cas. Vous êtes libre de lire autant de mots que vous le souhaitez :

```
if __name__=='__main__':
    # Lire les 12 000 premiers mots du corpus Brown
    input_data = ' '.join(brown.words()[:12000])

    # Définir le nombre de mots dans chaque segment
    chunk_size = 700

    # Diviser le texte d'entrée en segments
    chunks = chunker(input_data, chunk_size)

    # Afficher les résultats
    print('\nNombre de segments de texte :', len(chunks), '\n')
    for i, chunk in enumerate(chunks):
        print('Segment', i+1, '==>', chunk[:50])
```

Le code complet est disponible dans le fichier `text_chunker.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

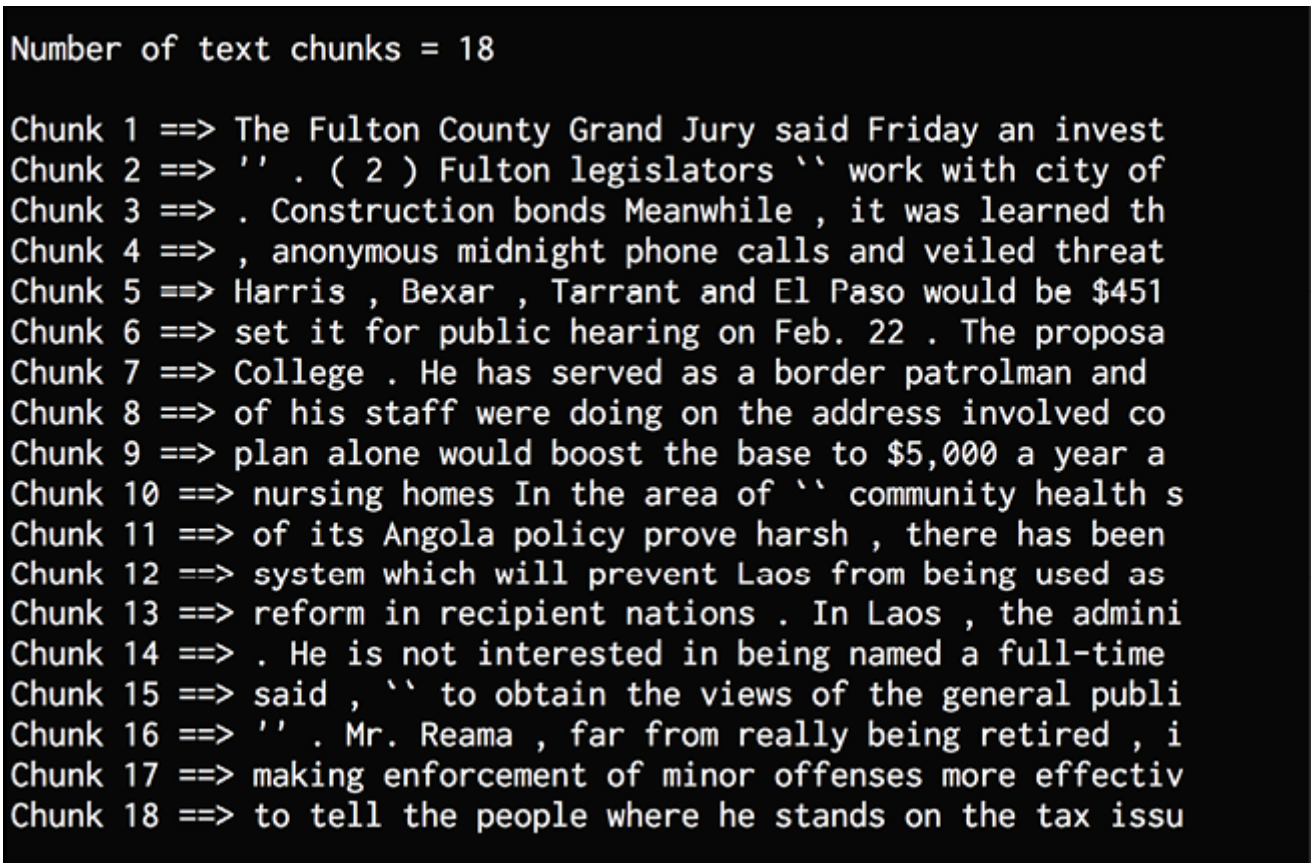


Figure 4 : Sortie du text chunker

La capture d'écran précédente montre les 50 premiers caractères de chaque segment.

Maintenant que nous avons exploré les techniques de division et de segmentation du texte, commençons à examiner les méthodes pour effectuer une analyse de texte.

Extraction de la fréquence des termes en utilisant le modèle du Sac de Mots

L'un des principaux objectifs de l'analyse de texte avec le modèle du Sac de Mots est de convertir le texte en une forme numérique afin que nous puissions utiliser l'apprentissage automatique dessus. Considérons des documents textuels contenant des millions de mots. Pour analyser ces documents, nous devons extraire le texte et le convertir en une forme de représentation numérique.

Les algorithmes d'apprentissage automatique ont besoin de données numériques pour fonctionner afin de pouvoir analyser les données et extraire des informations pertinentes. C'est là qu'intervient le modèle du Sac de Mots. Ce modèle extrait le vocabulaire de tous les mots dans les documents et construit un modèle en utilisant une matrice document-terme. Cela nous permet de représenter chaque document comme un *sac de mots*. Nous ne gardons que le comptage des mots et ignorons les détails grammaticaux et l'ordre des mots.

Considérons les phrases suivantes :

- Phrase 1 : The children are playing in the hall
- Phrase 2 : The hall has a lot of space
- Phrase 3 : Lots of children like playing in an open space

Si vous considérez les trois phrases, nous avons les 14 mots uniques suivants :

- the
- children
- are
- playing
- in
- hall
- has
- a
- lot
- of
- space
- like
- an
- open

Construisons un histogramme pour chaque phrase en utilisant le comptage des mots dans chaque phrase. Chaque vecteur de caractéristiques sera de 14 dimensions car nous avons 14 mots uniques :

- Phrase 1 : [2, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
- Phrase 2 : [1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0]
- Phrase 3 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1]

Maintenant que nous avons extrait ces caractéristiques avec le modèle du Sac de Mots, nous pouvons utiliser des algorithmes d'apprentissage automatique pour analyser ces données.

Voyons comment construire un modèle du Sac de Mots dans NLTK. Créez un nouveau fichier Python et importez les packages suivants :

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import brown
from text_chunker import chunker
```

Lisez les données d'entrée depuis le corpus Brown. Nous utiliserons `5 400` mots. N'hésitez pas à essayer avec autant de mots que vous le souhaitez :

```
# Lire les données depuis le corpus Brown
input_data = ' '.join(brown.words()[:5400])
```

Définissez le nombre de mots dans chaque segment :

```
# Nombre de mots dans chaque segment
chunk_size = 800
```

Divisez le texte d'entrée en segments :

```
text_chunks = chunker(input_data, chunk_size)
```

Convertissez les segments en éléments de dictionnaire :

```
# Convertir en éléments de dictionnaire
chunks = []
for count, chunk in enumerate(text_chunks):
    d = {'index': count, 'text': chunk}
    chunks.append(d)
```

Extrayez la matrice document-terme où nous obtenons le comptage de chaque mot. Nous y parviendrons en utilisant l'objet `CountVectorizer`, qui prend deux paramètres d'entrée. Le premier paramètre est la fréquence minimale des documents, et le second paramètre est la fréquence maximale des documents. La fréquence fait référence au nombre d'occurrences d'un mot dans le texte :

```
# Extraire la matrice document-terme
count_vectorizer = CountVectorizer(min_df=7, max_df=20)
document_term_matrix = count_vectorizer.fit_transform([chunk['text'] for chunk in
chunks])
```

Extrayez le vocabulaire avec le modèle du Sac de Mots et affichez-le. Le vocabulaire fait référence à la liste des mots distincts qui ont été extraits à l'étape précédente :

```
# Extraire le vocabulaire et l'afficher
vocabulary = np.array(count_vectorizer.get_feature_names())
print("\nVocabulaire :\n", vocabulary)
```

Générez les noms pour l'affichage :

```
# Générer les noms des segments
chunk_names = []
for i in range(len(text_chunks)):
    chunk_names.append('Segment-' + str(i+1))
```

Affichez la matrice document-terme :

```
# Afficher la matrice document-terme
print("\nMatrice document-terme :")
formatted_text = '{:>12}' * (len(chunk_names) + 1)
print('\n', formatted_text.format('Mot', *chunk_names), '\n')
for word, item in zip(vocabulary, document_term_matrix.T):
    # 'item' est une structure de données 'csr_matrix'
    output = [word] + [str(freq) for freq in item.data]
    print(formatted_text.format(*output))
```

Le code complet est disponible dans le fichier `bag_of_words.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

Document term matrix:

Word	Chunk-1	Chunk-2	Chunk-3	Chunk-4	Chunk-5	Chunk-6	Chunk-7
and	23	9	9	11	9	17	10
are	2	2	1	1	2	2	1
be	6	8	7	7	6	2	1
by	3	4	4	5	14	3	6
county	6	2	7	3	1	2	2
for	7	13	4	10	7	6	4
in	15	11	15	11	13	14	17
is	2	7	3	4	5	5	2
it	8	6	8	9	3	1	2
of	31	20	20	30	29	35	26
on	4	3	5	10	6	5	2
one	1	3	1	2	2	1	1
said	12	5	7	7	4	3	7
state	3	7	2	6	3	4	1
that	13	8	9	2	7	1	7
the	71	51	43	51	43	52	49
to	11	26	20	26	21	15	11
two	2	1	1	1	1	2	2
was	5	6	7	7	4	7	3
which	7	4	5	4	3	1	1
with	2	2	3	1	2	2	3

Figure 5 : Sortie de la matrice document-terme du modèle du Sac de Mots

Tous les mots peuvent être vus dans la matrice document-terme du modèle du Sac de Mots avec les comptages correspondants dans chaque segment.

Maintenant que nous avons effectué un comptage des mots, nous pouvons nous appuyer sur cela et commencer à faire des prédictions basées sur la fréquence des mots.

Construction d'un prédicteur de catégorie

Un **prédicteur de catégorie** est utilisé pour prédire la catégorie à laquelle appartient un texte donné. Cela est fréquemment utilisé dans la classification de texte pour catégoriser des documents textuels. Les moteurs de recherche utilisent souvent cet outil pour ordonner les résultats de recherche par pertinence. Par exemple, supposons que nous voulons prédire si une phrase donnée appartient au sport, à la politique ou à la science. Pour ce faire, nous construisons un corpus de données et entraînons un algorithme. Cet algorithme peut ensuite être utilisé pour l'inférence sur des données inconnues.

Pour construire ce prédicteur, nous utiliserons une métrique appelée **Fréquence de Terme – Fréquence Inverse de Document** (**tf-idf**). Dans un ensemble de documents, nous devons comprendre l'importance de chaque mot. La métrique tf-idf nous aide à comprendre à quel point un mot donné est important pour un document dans un ensemble de documents.

Considérons la première partie de cette métrique. La **Fréquence de Terme** (**tf**) est essentiellement une mesure de la fréquence à laquelle chaque mot apparaît dans un document

donné. Comme différents documents ont un nombre différent de mots, les nombres exacts dans l'histogramme varieront. Afin de disposer d'un terrain de jeu équitable, nous devons normaliser les histogrammes. Ainsi, nous divisons le comptage de chaque mot par le nombre total de mots dans un document donné pour obtenir la fréquence de terme.

La seconde partie de la métrique est la **Fréquence Inverse de Document** (**idf**), qui est une mesure de la rareté d'un mot dans un ensemble de documents. Lorsque nous calculons la fréquence de terme, l'hypothèse est que tous les mots sont également importants. Mais nous ne pouvons pas simplement nous fier à la fréquence de chaque mot car des mots tels que *and*, *or* et *the* apparaissent beaucoup. Pour équilibrer les fréquences de ces mots couramment utilisés, nous devons réduire leurs poids et augmenter les poids des mots rares. Cela nous aide à identifier les mots uniques à chaque document également, ce qui nous aide à formuler un vecteur de caractéristiques distinctif.

Pour calculer cette statistique, nous devons calculer le ratio du nombre de documents contenant le mot donné et le diviser par le nombre total de documents. Ce ratio est essentiellement la fraction des documents contenant le mot donné. La fréquence inverse de document est ensuite calculée en prenant le logarithme négatif de ce ratio.

Nous combinons ensuite la fréquence de terme et la fréquence inverse de document pour formuler un vecteur de caractéristiques afin de catégoriser les documents. Ce travail est la base pour une analyse plus approfondie du texte afin d'obtenir une signification plus profonde, telle que l'analyse des sentiments, le contexte du texte ou la modélisation thématique. Voyons comment construire un prédicteur de catégorie.

Créez un nouveau fichier Python et importez les packages suivants :

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer
```

Définissez la carte des catégories qui sera utilisée pour l'entraînement. Nous utiliserons cinq catégories dans ce cas. Les clés dans ce dictionnaire font référence aux noms dans l'ensemble de données de scikit-learn :

```
# Définir la carte des catégories
category_map = {
    'talk.politics.misc': 'Politique',
    'rec.autos': 'Autos',
    'rec.sport.hockey': 'Hockey',
    'sci.electronics': 'Électronique',
    'sci.med': 'Médecine'
}
```

Obtenez l'ensemble de données d'entraînement en utilisant `fetch_20newsgroups` :

```
# Obtenir l'ensemble de données d'entraînement
training_data = fetch_20newsgroups(
    subset='train',
    categories=category_map.keys(),
    shuffle=True,
    random_state=5
)
```

Extrayez les comptages de termes en utilisant l'objet `CountVectorizer` :

```
# Construire un CountVectorizer et extraire les comptages de termes
count_vectorizer = CountVectorizer()
train_tc = count_vectorizer.fit_transform(training_data.data)
print("\nDimensions des données d'entraînement :", train_tc.shape)
```

Créez le transformateur **tf-idf** et entraînez-le en utilisant les données :

```
# Créer le transformateur tf-idf
tfidf = TfidfTransformer()
train_tfidf = tfidf.fit_transform(train_tc)
```

Définissez quelques phrases d'entrée qui seront utilisées pour les tests :

```
# Définir des phrases d'entrée pour les tests
input_data = [
    'You need to be careful with cars when you are driving on slippery roads',
    'A lot of devices can be operated wirelessly',
    'Players need to be careful when they are close to goal posts',
    'Political debates help us understand the perspectives of both sides'
]
```

Entraînez un classificateur bayésien multinomial en utilisant les données d'entraînement :

```
# Entraîner un classificateur Naive Bayes multinomial
classifier = MultinomialNB().fit(train_tfidf, training_data.target)
```

Transformez les données d'entrée en utilisant le vectoriseur de comptage :

```
# Transformer les données d'entrée en utilisant le vectoriseur de comptage
input_tc = count_vectorizer.transform(input_data)
```

Transformez les données vectorisées en utilisant le transformateur **tf-idf** afin qu'elles puissent être passées à travers le modèle d'inférence :

```
# Transformer les données vectorisées en utilisant le transformateur tf-idf
input_tfidf = tfidf.transform(input_tc)
```

Prédisez la sortie en utilisant le vecteur transformé **tf-idf** :

```
# Prédire les catégories de sortie
predictions = classifier.predict(input_tfidf)
```

Affichez la catégorie de sortie pour chaque échantillon dans les données de test d'entrée :

```
# Afficher la catégorie prédite pour chaque phrase d'entrée
for sent, category in zip(input_data, predictions):
    print('\nEntrée :', sent)
    print('Catégorie prédite :', category_map[training_data.target_names[category]])
```

Le code complet est disponible dans le fichier `category_predictor.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

```
Dimensions of training data: (2844, 40321)

Input: You need to be careful with cars when you are driving on slippery roads
Predicted category: Autos

Input: A lot of devices can be operated wirelessly
Predicted category: Electronics

Input: Players need to be careful when they are close to goal posts
Predicted category: Hockey

Input: Political debates help us understand the perspectives of both sides
Predicted category: Politics
```

Figure 6 : Sortie du prédicteur de catégorie

Nous pouvons voir intuitivement que les catégories prédites sont correctes. Passons maintenant à une autre forme d'analyse de texte – l'identification du genre.

Construction d'un identificateur de genre

L'identification du genre est un problème intéressant et loin d'être une science exacte. Nous pouvons rapidement penser à des noms qui peuvent être utilisés à la fois pour les hommes et les femmes :

- Dana
- Angel
- Lindsey
- Morgan
- Jessie
- Chris
- Payton
- Tracy
- Stacy
- Jordan
- Robin
- Sydney

De plus, dans une société hétérogène telle que les États-Unis, il y aura de nombreux noms ethniques qui ne suivront pas les règles anglaises. En général, nous pouvons faire une supposition éclairée pour une large gamme de noms. Dans cet exemple simple, nous utiliserons une heuristique pour construire un vecteur de caractéristiques et l'utiliser pour entraîner un

classificateur. L'heuristique qui sera utilisée ici est les dernières *N* lettres d'un nom donné. Par exemple, si le nom se termine par *ia*, c'est probablement un nom féminin, comme *Amelia* ou *Genelia*. D'autre part, si le nom se termine par *rk*, c'est probablement un nom masculin, comme *Mark* ou *Clark*. Comme nous ne sommes pas sûrs du nombre exact de lettres à utiliser, nous allons jouer avec ce paramètre et découvrir quelle est la meilleure réponse. Voyons comment faire.

Créez un nouveau fichier Python et importez les packages suivants :

```
import random
from nltk import NaiveBayesClassifier
from nltk.classify import accuracy as nltk_accuracy
from nltk.corpus import names
```

Définissez une fonction pour extraire les dernières *N* lettres du mot d'entrée :

```
# Extraire les dernières N lettres du mot d'entrée
# et cela agira comme notre "caractéristique"
def extract_features(word, N=2):
    last_n_letters = word[-N:]
    return {'feature': last_n_letters.lower()}
```

Définissez la fonction `main` et extrayez les données d'entraînement depuis le package `nltk`. Ces données contiennent des noms masculins et féminins étiquetés :

```
if __name__=='__main__':
    # Créer les données d'entraînement en utilisant des noms étiquetés disponibles
dans NLTK
    male_list = [(name, 'male') for name in names.words('male.txt')]
    female_list = [(name, 'female') for name in names.words('female.txt')]
    data = male_list + female_list
```

Semez le générateur de nombres aléatoires et mélangez les données :

```
    # Semez le générateur de nombres aléatoires
    random.seed(5)
    # Mélanger les données
    random.shuffle(data)
```

Créez des noms d'échantillons qui seront utilisés pour les tests :

```
    # Créer des données de test
    input_names = ['Alexander', 'Danielle', 'David', 'Cheryl']
```

Définissez le pourcentage des données qui sera utilisé pour l'entraînement et le test :

```
    # Définir le pourcentage des données utilisé pour l'entraînement et le test
    num_train = int(0.8 * len(data))
```

La dernière *N* lettres sera utilisée comme vecteur de caractéristiques pour prédire le genre. Ce paramètre sera modifié pour voir comment les performances varient. Dans ce cas, nous allons de `1` à `5` :

```
    # Itérer à travers différentes longueurs pour comparer la précision
    for i in range(1, 6):
        print('\nNombre de lettres finales :', i)
        features = [(extract_features(n, i), gender) for (n, gender) in data]

        train_data, test_data = features[:num_train], features[num_train:]

        classifier = NaiveBayesClassifier.train(train_data)

        # Calculer la précision du classificateur
        accuracy = round(100 * nltk_accuracy(classifier, test_data), 2)
        print('Précision = ' + str(accuracy) + '%')

        # Prédire les sorties pour les noms d'entrée en utilisant
        # le modèle classificateur entraîné
        for name in input_names:
            print("\nNom :", name, "==>", classifier.classify(extract_features(name,
i)))
```

Le code complet est disponible dans le fichier `gender_identifier.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

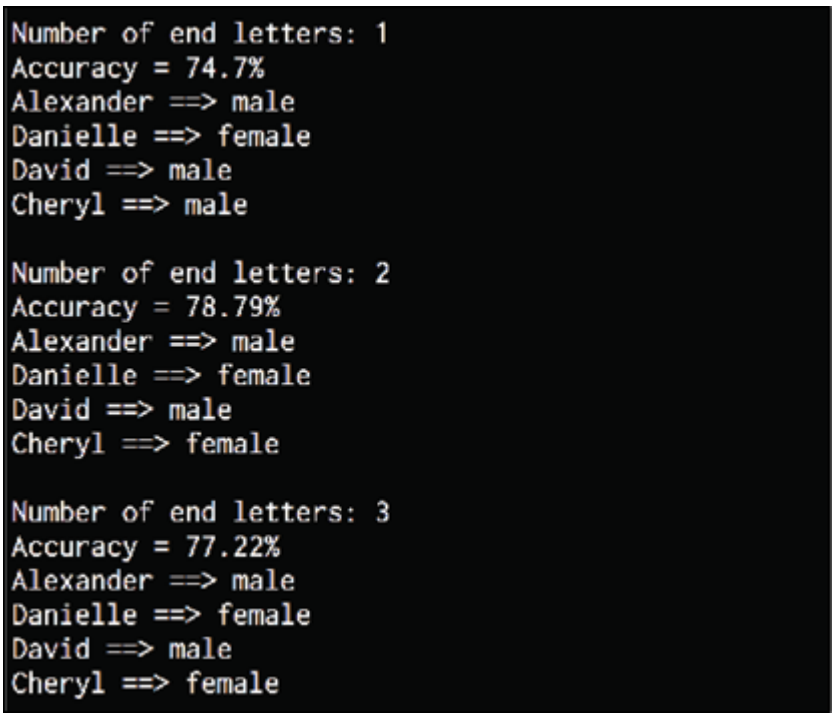


Figure 7 : Sortie de l'identificateur de genre

Passons plus loin et voyons ce qui se passe :

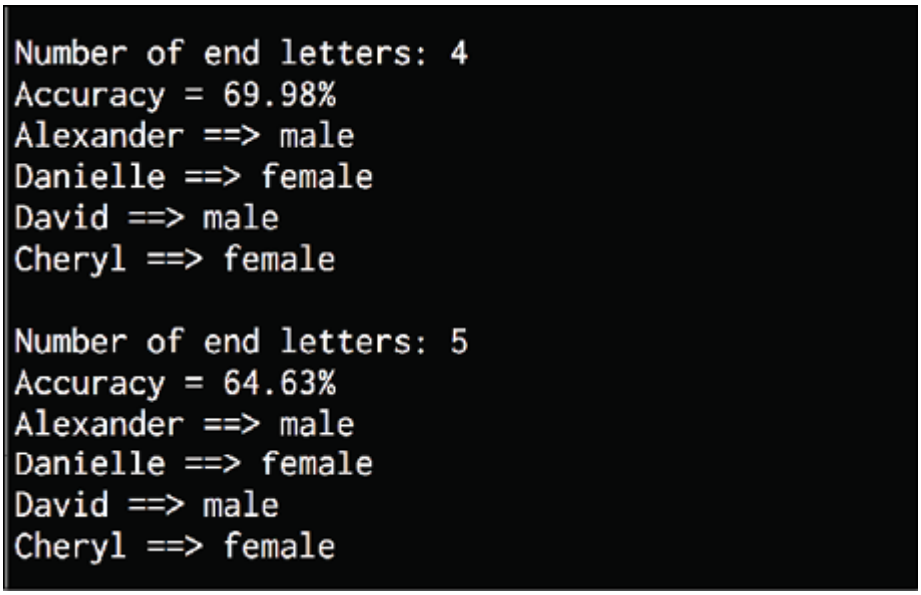


Figure 8 : Sortie de l'identificateur de genre

Nous pouvons voir que la précision a culminé à deux lettres et a commencé à diminuer après cela. Passons maintenant à un autre problème intéressant – l'analyse du sentiment d'un texte.

Construction d'un analyseur de sentiments

L'**analyse de sentiments** est le processus de détermination du sentiment d'un morceau de texte. Par exemple, il peut être utilisé pour déterminer si une critique de film est positive ou négative. C'est l'une des applications les plus populaires du traitement du langage naturel. Nous pouvons ajouter plus de catégories également, en fonction du problème à résoudre. Cette technique peut être utilisée pour avoir une idée de la façon dont les gens se sentent à propos d'un produit, d'une marque ou d'un sujet. Elle est fréquemment utilisée pour analyser les campagnes marketing, les sondages d'opinion, la présence sur les réseaux sociaux, les avis de produits sur les sites de commerce électronique, etc. Voyons comment déterminer le sentiment d'une critique de film.

Nous utiliserons un classificateur bayésien Naive Bayes pour construire cet analyseur de sentiments. Tout d'abord, extrayez tous les mots uniques du texte. Le classificateur NLTK a besoin que ces données soient organisées sous forme de dictionnaire afin qu'il puisse les ingérer. Une fois que les données textuelles sont divisées en ensembles d'entraînement et de test, le classificateur bayésien Naive Bayes sera entraîné pour classer les critiques en sentiments positifs et négatifs. Ensuite, les mots les plus informatifs pour indiquer les critiques positives et négatives peuvent être calculés et affichés. Ces informations sont intéressantes car elles montrent quels mots sont utilisés pour dénoter diverses réactions.

Voyons comment cela peut être réalisé. Tout d'abord, créez un nouveau fichier Python et importez les packages suivants :

```
from nltk.corpus import movie_reviews
from nltk.classify import NaiveBayesClassifier
from nltk.classify.util import accuracy as nltk_accuracy
```

Définissez une fonction pour construire un objet dictionnaire basé sur les mots d'entrée et le retourner :

```
# Extraire les caractéristiques de la liste de mots d'entrée
def extract_features(words):
    return dict([(word, True) for word in words])
```

Définissez la fonction `main` et chargez les critiques de films étiquetées depuis le corpus :

```
if __name__=='__main__':
    # Charger les critiques depuis le corpus
    fileids_pos = movie_reviews.fileids('pos')
    fileids_neg = movie_reviews.fileids('neg')
```

Extrayez les caractéristiques des critiques de films et étiquetez-les en conséquence :

```
    # Extraire les caractéristiques des critiques
    features_pos = [(extract_features(movie_reviews.words(fileids=[f])), 'Positive')
for f in fileids_pos]
    features_neg = [(extract_features(movie_reviews.words(fileids=[f])), 'Negative')
for f in fileids_neg]
```

Définissez la séparation entre l'entraînement et le test. Dans ce cas, nous allouerons 80% pour l'entraînement et 20% pour le test :

```
    # Définir la séparation pour l'entraînement et le test (80% et 20%)
    threshold = 0.8
    num_pos = int(threshold * len(features_pos))
    num_neg = int(threshold * len(features_neg))
```

Séparez les vecteurs de caractéristiques pour l'entraînement et le test :

```
    # Créer les ensembles d'entraînement et de test
    features_train = features_pos[:num_pos] + features_neg[:num_neg]
    features_test = features_pos[num_pos:] + features_neg[num_neg:]
```

Entraînez un classificateur `NaiveBayesClassifier` en utilisant les données d'entraînement et calculez la précision en utilisant la méthode d'exactitude intégrée disponible dans NLTK :

```
    # Entraîner un classificateur Naive Bayes
    classifier = NaiveBayesClassifier.train(features_train)
    print('\nExactitude du classificateur :', nltk_accuracy(classifier,
features_test))
```

Affichez les mots les plus informatifs :

```
    # Définir le nombre de mots informatifs à afficher
    N = 15
    print('\nTop ' + str(N) + ' mots les plus informatifs :')
    for i, item in enumerate(classifier.most_informative_features()):
        print(str(i+1) + '. ' + item[0])
        if i == N - 1:
            break
```

Définissez des phrases d'échantillons qui seront utilisées pour les tests :

```
# Critiques de films d'entrée pour les tests
input_reviews = [
    'The costumes in this movie were great',
    'I think the story was terrible and the characters were very weak',
    'People say that the director of the movie is amazing',
    'This is such an idiotic movie. I will not recommend it to anyone.'
]
```

Itérez à travers les données d'échantillons et prédisez la sortie :

```
print("\nPrédictions des sentiments des critiques de films :")
for review in input_reviews:
    print("\nCritique :", review)

    # Calculer les probabilités
    probabilities = classifier.prob_classify(extract_features(review.split()))

    # Choisir la valeur maximale
    predicted_sentiment = probabilities.max()

    # Afficher la classe prédite (sentiment positif ou négatif)
    print("Sentiment prédit :", predicted_sentiment)
    print("Probabilité :", round(probabilities.prob(predicted_sentiment), 2))
```

Le code complet est disponible dans le fichier `sentiment_analyzer.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

```
Number of training datapoints: 1600
Number of test datapoints: 400

Accuracy of the classifier: 0.735

Top 15 most informative words:
1. outstanding
2. insulting
3. vulnerable
4. ludicrous
5. uninvolving
6. astounding
7. avoids
8. fascination
9. symbol
10. seagal
11. affecting
12. anna
13. darker
14. animators
15. idiotic
```

Figure 9 : Sortie de l'analyseur de sentiments

La capture d'écran précédente affiche les 15 mots les plus informatifs. Si vous faites défiler vers le bas, vous verrez ceci :

```
Movie review predictions:

Review: The costumes in this movie were great
Predicted sentiment: Positive
Probability: 0.59

Review: I think the story was terrible and the characters were very weak
Predicted sentiment: Negative
Probability: 0.8

Review: People say that the director of the movie is amazing
Predicted sentiment: Positive
Probability: 0.6

Review: This is such an idiotic movie. I will not recommend it to anyone.
Predicted sentiment: Negative
Probability: 0.87
```

Figure 10 : Sortie des sentiments des critiques de films

Nous pouvons voir et vérifier intuitivement que les prédictions sont correctes.

Dans cette section, nous avons construit un analyseur de sentiments sophistiqué. Nous continuerons notre parcours dans l'espace NLP et apprendrons les bases de l'Allocation Dirichlet Latente.

Modélisation thématique à l'aide de l'Allocation Dirichlet Latente

La **modélisation thématique** est le processus d'identification des motifs dans les données textuelles qui correspondent à un sujet. Si le texte contient plusieurs sujets, cette technique peut être utilisée pour identifier et séparer ces thèmes au sein du texte d'entrée. Cette technique peut être utilisée pour découvrir la structure thématique cachée dans un ensemble donné de documents.

La modélisation thématique nous aide à organiser les documents de manière optimale, ce qui peut ensuite être utilisé pour l'analyse. Une chose à noter sur les algorithmes de modélisation thématique est qu'ils n'ont pas besoin de données étiquetées. C'est comme l'apprentissage non supervisé en ce qu'il identifiera les motifs par lui-même. Étant donné les volumes énormes de données textuelles générées sur Internet, la modélisation thématique est importante car elle permet la synthèse de vastes quantités de données, ce qui ne serait autrement pas possible.

L'Allocation Dirichlet Latente (**Latent Dirichlet Allocation** ou **LDA**) est une technique de modélisation thématique dont le concept sous-jacent est qu'un morceau de texte donné est une combinaison de plusieurs sujets. Considérons la phrase suivante : La visualisation des données est un outil important dans l'analyse financière. Cette phrase comporte plusieurs sujets, tels que les données, la visualisation et la finance. Cette combinaison aide à identifier le texte dans un grand document. C'est un modèle statistique qui essaie de capturer des concepts et de créer un modèle basé sur eux. Le modèle suppose que les documents sont générés par un processus aléatoire basé sur ces sujets. Un sujet est une distribution sur un vocabulaire fixe de mots. Voyons comment faire de la modélisation thématique en Python.

La bibliothèque `gensim` sera utilisée dans cette section. Cette bibliothèque a déjà été installée dans la première section de ce chapitre. Assurez-vous de l'avoir avant de poursuivre. Créez un nouveau fichier Python et importez les packages suivants :

```
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
from gensim import models, corpora
```

Définissez une fonction pour charger les données d'entrée. Le fichier d'entrée contient 10 phrases séparées par des lignes :

```
# Charger les données d'entrée
def load_data(input_file):
    data = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data.append(line.strip())
    return data
```

Définissez une fonction pour traiter le texte d'entrée. La première étape consiste à le tokeniser :

```
# Fonction de traitement pour la tokenisation, suppression des stop
# words et racinisation
def process(input_text):
    # Créer un tokeniseur à expression régulière
    tokenizer = RegexpTokenizer(r'\w+')

    # Créer un stemmer Snowball
    stemmer = SnowballStemmer('english')

    # Obtenir la liste des stop words
    stop_words = stopwords.words('english')

    # Tokeniser la chaîne d'entrée
    tokens = tokenizer.tokenize(input_text.lower())

    # Supprimer les stop words
    tokens = [x for x in tokens if x not in stop_words]

    # Effectuer la racinisation sur les mots tokenisés
    tokens_stemmed = [stemmer.stem(x) for x in tokens]
    return tokens_stemmed
```

Définissez la fonction `main` et chargez les données d'entrée depuis le fichier `data.txt` fourni :

```
if __name__=='__main__':
    # Charger les données d'entrée
    data = load_data('data.txt')

    # Créer une liste de tokens de phrases
    tokens = [process(x) for x in data]
```

Créez un dictionnaire basé sur les tokens de phrases :

```
# Créer un dictionnaire basé sur les tokens des phrases
dict_tokens = corpora.Dictionary(tokens)
```

Créez une matrice document-terme en utilisant les tokens de phrases :

```
# Créer une matrice document-terme
doc_term_mat = [dict_tokens.doc2bow(token) for token in tokens]
```

Nous devons fournir le nombre de sujets comme paramètre d'entrée. Dans ce cas, nous savons que le texte d'entrée a deux sujets distincts. Spécifions cela :

```
# Définir le nombre de sujets pour le modèle LDA
num_topics = 2
```

Générez le modèle `LatentDirichlet` :

```
# Générer le modèle LDA
ldamodel = models.ldamodel.LdaModel(
    doc_term_mat,
    num_topics=num_topics,
    id2word=dict_tokens,
    passes=25
)
```

Affichez les cinq mots les plus contributeurs pour chaque sujet :

```
num_words = 5
print('\nTop ' + str(num_words) + ' mots contributeurs pour chaque sujet :')
for item in ldamodel.print_topics(num_topics=num_topics, num_words=num_words):
    print('\nSujet', item[0])
    # Afficher les mots contributeurs avec leurs contributions relatives
    list_of_strings = item[1].split(' + ')
    for text in list_of_strings:
        weight = text.split('*')[0]
        word = text.split('*')[1].strip('"')
        print(word, '==>', str(round(float(weight) * 100, 2)) + '%')
```

Le code complet est disponible dans le fichier `topic_modeler.py`. Si vous exécutez le code, vous obtiendrez la sortie suivante :

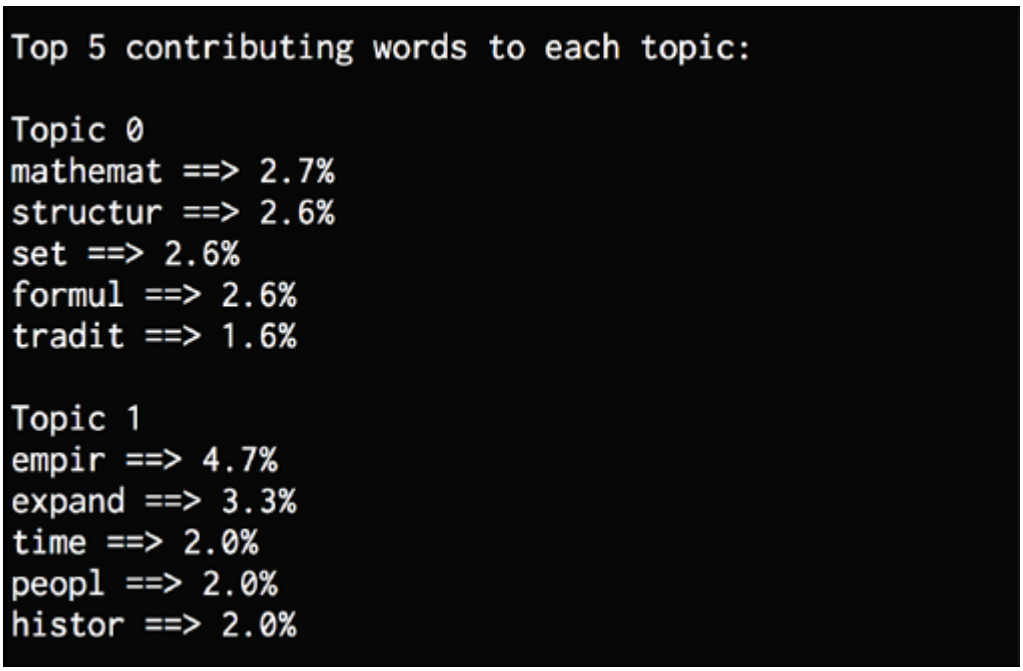


Figure 11 : Sortie du modèle thématique

Nous pouvons voir qu'il fait un travail raisonnable de séparation des deux sujets – mathématiques et histoire. Si vous regardez le texte, vous pouvez vérifier que chaque phrase est soit sur les mathématiques, soit sur l'histoire.

Résumé

Dans ce chapitre, nous avons appris divers concepts sous-jacents au **traitement du langage naturel**. Nous avons discuté de la tokenisation et de la manière de séparer le texte d'entrée en plusieurs tokens. Nous avons appris à réduire les mots à leurs formes de base à l'aide de la racinisation et de la lemmatisation. Nous avons implémenté un chunker de texte pour diviser le texte d'entrée en segments basés sur des conditions prédéfinies.

Nous avons discuté du modèle du **Sac de Mots** et construit une matrice document-terme pour le texte d'entrée. Nous avons ensuite appris à catégoriser le texte en utilisant l'apprentissage automatique. Nous avons construit un identificateur de genre en utilisant une heuristique. Nous avons également utilisé l'apprentissage automatique pour analyser les sentiments des critiques de films. Enfin, nous avons discuté de la modélisation thématique et implémenté un système pour identifier les sujets dans un document donné.

END