Travaux pratiques : Introduction aux Word Embeddings

1 Questions de compréhension

1. Représentations discrètes. Que signifie représenter un mot sous forme de vecteur one-hot (one-hot encoding) ? Citez au moins deux Limitations majeures de cette représentation.

Réponse:

Représenter un mot sous forme de vecteur one-hot (ou encodage one-hot) signifie créer un vecteur de la même taille que le vocabulaire total de mots, où :

- Chaque dimension correspond à un mot du vocabulaire.
- Le vecteur contient 0 partout sauf à la position correspondant au mot choisi, où il y a 1.

Par exemple, si le vocabulaire est `["chat", "chien", "oiseau"]` :

- Le mot `"chien"` sera représenté par le vecteur `[0, 1, 0]`.

Limitations majeures du one-hot encoding :

1. Sparsité et grande dimension
    - Pour un vocabulaire très large (des dizaines ou centaines de milliers de mots), les vecteurs sont très grands et contiennent presque uniquement des zéros.
    - Cela rend le stockage et le calcul inefficaces.

2. Absence de similarité sémantique

    - Le one-hot ne capture aucune relation entre les mots. Par exemple, `"chat"` et `"chien"` sont aussi éloignés que `"chat"` et `"avion"`, même si les deux premiers sont des animaux.
3. Incapacité à généraliser
    - Les modèles utilisant one-hot ne peuvent pas comprendre les synonymes ou inférer le sens des mots non vus pendant l'entraînement.
    -

2. Sac de mots et TF-IDF. Expliquez en quelques lignes en quoi consiste un modèle bag-of-words (BoW) appliqué à un document et comment l'indice TF-IDF modifie cette représentation. Pourquoi ces représentations restent-elles de haute dimension et ne capturent-elles
pas la sémantique inter-mots ?
Réponse:

Un modèle bag-of-words représente un document par la fréquence d'apparition de chaque mot dans ce document, sans tenir compte de l'ordre des mots.

- On crée un vecteur où chaque dimension correspond à un mot du vocabulaire global.
- La valeur dans chaque dimension est généralement le nombre de fois que le mot apparaît dans le document.

Exemple :
Vocabulaire = ["chat", "chien", "oiseau"]
Document = "chat chat oiseau"
Vecteur BoW = `[2, 0, 1]`

2. TF-IDF (Term Frequency – Inverse Document Frequency)

Le TF-IDF ajuste les fréquences du BoW en pondérant les mots :

- TF (Term Frequency) = fréquence du mot dans le document (comme BoW).

- IDF (Inverse Document Frequency) = valeur qui diminue l'importance des mots très fréquents dans tous les documents (ex. "le", "et").

Ainsi, un mot courant dans tout le corpus aura un poids faible, tandis qu'un mot rare mais spécifique aura un poids plus élevé.

- Cela rend la représentation plus informative et discriminante pour la classification ou la recherche.

3. Limitations

1. Haute dimension
    - La dimension du vecteur = taille du vocabulaire.
    - Pour de grands corpus, cela produit des vecteurs très longs et creux (sparse).

2. Pas de sémantique inter-mots
    - BoW et TF-IDF ignorent l'ordre des mots et les relations entre eux.
    - Ex. : "chien mord chat" et "chat mord chien" ont presque le même vecteur.
    - Les synonymes ou contextes proches ne sont pas capturés.

3. Hypothèse distributionnelle. Reformulez l'hypothèse de J.R. Firth :"You shall know a word by the company it keeps". Comment cette hy-pothèse motive-t-elle l'utilisation de fenêtres contextuelles pour ap-prendre des embeddings ?

Réponse:

1. Reformulation de l'hypothèse de Firth

La phrase **"You shall know a word by the company it keeps"** peut se reformuler ainsi :

> *"On peut comprendre le sens d'un mot en examinant les mots qui l'entourent dans le texte."*

Autrement dit, **le contexte d'un mot révèle beaucoup sur sa signification**.

2. Lien avec les fenêtres contextuelles et les embeddings

- Lorsqu'on apprend des **embeddings de mots** (représentations vectorielles denses), on se base sur cette idée : **les mots qui apparaissent dans des contextes similaires devraient avoir des vecteurs proches**.
- Pour capturer ce contexte, on définit une **fenêtre contextuelle** autour de chaque mot (par exemple ±5 mots).
    - Les modèles comme **Word2Vec (CBOW ou Skip-gram)** utilisent ces fenêtres pour prédire un mot à partir de ses voisins ou vice-versa.
- Ainsi, les co-occurrences observées dans ces fenêtres permettent au modèle d'apprendre des vecteurs qui **encodent la similarité sémantique entre mots**.

4. Propriétés des embeddings denses. Donnez un exemple de relation analogique que l'on peut retrouver grâce aux embeddings (par exemple roi − homme + femme ≈reine). Que traduit ce type de calcul ?

Réponse:

Un exemple classique est :

“roi” − “homme” + “femme” ≈ “reine”\text{“roi” − “homme” + “femme” ≈ “reine”}“roi” − “homme” + “femme” ≈ “reine”

- En termes de vecteurs :
  $\vec{\text{roi}} - \vec{\text{homme}} + \vec{\text{femme}} \approx \vec{\text{reine}}$\vec{\text{roi}} - \vec{\text{homme}} + \vec{\text{femme}} \approx \vec{\text{reine}}roi−homme+femme≈reine

Autres exemples possibles :

- Paris − France + Italie ≈ Rome
- Chat − petit + grand ≈ chien (dans certains contextes)

2. Signification de ce calcul

- Cela traduit le fait que **les embeddings capturent les relations sémantiques et syntaxiques entre mots**.
- La **direction et la distance dans l'espace vectoriel** codent des concepts abstraits comme le genre, le pays-capitale, ou les relations taille/poids.
- Autrement dit, les vecteurs ne représentent pas seulement les mots individuellement, mais **leurs relations dans le langage**.

2 Manipulation de représentations discrètes

P1 : "Le chat dort sur le tapis."

P2 : "Le tapis dort sur le chat."

1. Construisez le vocabulaire V de ces deux phrases (liste de tous les mots distincts) et associez à chaque mot un vecteur one-hot.

Réponse:

On liste tous les mots distincts (sans tenir compte de la ponctuation) :

V=["Le","chat","dort","sur","le","tapis"]

il y a une subtilité : `"Le"` et `"le"` sont identiques si on ignore la casse. Pour simplifier, on peut tout mettre en minuscule :

V=["le","chat","dort","sur","tapis"]

Donc le vocabulaire final contient **5 mots**.

Mot	One-hot vector
le	[1,0,0,0,0]
chien	[0,1,0,0,0]
dort	[0,0,1,0,0]
sur	[0,0,0,1,0]
tapis	[0,0,0,0,1]

Représentation des phrases

- P1 : `"le chat dort sur le tapis"` → [[1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,0,0,1,0], [1,0,0,0,0], [0,0,0,0,1]]
- P2 : `"le tapis dort sur le chat"` → [[1,0,0,0,0], [0,0,0,0,1], [0,0,1,0,0], [0,0,0,1,0], [1,0,0,0,0], [0,1,0,0,0]]

2. Représentez chacun des deux documents P1 et P2 par un vecteur bag-of-words en comptant les occurrences de chaque mot. Ces vecteurs sont-ils différents ? Que dit cette représentation sur l'ordre des mots ?

Réponse:

On compte combien de fois chaque mot du vocabulaire apparaît dans chaque phrase.

Pour P1 :

- "le" → 2 fois
- "chat" → 1 fois
- "dort" → 1 fois
- "sur" → 1 fois
- "tapis" → 1 fois

BoW(P1)=[2,1,1,1,1]\text{BoW(P1)} = [2, 1, 1, 1, 1]BoW(P1)=[2,1,1,1,1]

Pour P2 :

- "le" → 2 fois
- "chat" → 1 fois
- "dort" → 1 fois
- "sur" → 1 fois
- "tapis" → 1 fois

BoW(P2)=[2,1,1,1,1]\text{BoW(P2)} = [2,1,1,1,1]BoW(P2)=[2,1,1,1,1]

3. Observation

- Les vecteurs **BoW sont identiques** : `[2, 1, 1, 1, 1]`.
- Cela montre que la **représentation Bag-of-Words ne capture pas l'ordre des mots**.
    - Même si les phrases ont des significations différentes (P1 : le chat dort sur le tapis, P2 : le tapis dort sur le chat), le BoW ne fait pas la distinction.

3. Calculez ensuite les poids TF-IDF pour chacun des mots en supposant que l'ensemble de documents se limite à P1 et P2. Quel(s) mot(s) obtiennent le plus grand poids IDF ? Interprétez.

Réponse:

Formules TF et IDF

- **TF (Term Frequency)** pour un mot www dans un document ddd : nombre d'occurrences de www dans ddd.

- **IDF (Inverse Document Frequency)** pour un mot www :

IDF(w)=logN1+nw\text{IDF}(w) = \log\frac{N}{1 + n_w}IDF(w)=log1+nwN

où :

- N=2N = 2N=2 (nombre total de documents)
- nwn_wnw = nombre de documents contenant le mot www
- On ajoute 1 dans le dénominateur pour éviter la division par zéro.
- **TF-IDF** = TF × IDF

Mot	Documents contenant le mot	nw	IDF = log(N / (1 + n_w))
le	P1,P2	2	log(2 / (1+2)) = log(2/3) ≈ -0.405
chat	P1,P2	2	log(2/3) ≈ -0.405
dort	P1,P2	2	log(2/3) ≈ -0.405
sur	P1,P2	2	log(2/3) ≈ -0.405

tapis	P1,P2	2	log(2/3) ≈ -0.405

TF-IDF pour P1 (exemple)

TF pour P1 : `[2,1,1,1,1]` (pour ["le","chat","dort","sur","tapis"])
TF-IDF = TF × IDF = **`[2×(-0.405), 1×(-0.405), …] = [-0.81, -0.405, -0.405, -0.405, -0.405]`**

Même résultat pour P2 : `[-0.81, -0.405, -0.405, -0.405, -0.405]`

4. Commentez brièvement les limites de ces représentations pour captu-rer le sens des phrases, notamment la différence entre ”Le chat mord le chien” et ”Le chien mord le chat”.

Réponse:

Limites des représentations discrètes (one-hot, BoW, TF-IDF)

- **Ignorent l’ordre des mots** :
  - BoW et TF-IDF ne prennent en compte que la fréquence des mots, pas leur position.
  - Ainsi, des phrases avec les mêmes mots mais dans un ordre différent obtiennent des vecteurs identiques ou très similaires.

- **Ne capturent pas la syntaxe ni les relations grammaticales** :
  - Elles ne distinguent pas le sujet du complément d'objet.
  - Exemple :
    - P1 : "Le chat mord le chien"
    - P2 : "Le chien mord le chat"
    - Avec BoW ou TF-IDF, ces phrases ont **exactement le même vecteur**, même si le sens est totalement inversé.

- **Ne saisissent pas la sémantique contextuelle** :
  - Les synonymes ou mots liés par le sens restent indépendants.
  - Exemple : "chat" et "félin" sont considérés complètement différents.

Conséquence

Ces représentations sont **utiles pour des tâches simples** (recherche par mot, classification basique), mais **inadéquates pour comprendre le sens précis des phrases**, pour lequel des modèles basés sur les **embeddings denses ou contextualisés** (Word2Vec, GloVe, BERT…) sont nécessaires.

3 Entraînement d'un modèle Word2Vec

3.1 Préparation des données

1. Choisissez un petit corpus de texte en français (par exemple un extrait d'uvre littéraire disponible librement ou un ensemble de tweets nettoyés). Chargez ce corpus en Python et appliquez une tokenisation simple (extraction des mots en minuscules, suppression de la ponctuation).

Réponse:

```
corpus = [
    "Le chat dort sur le tapis.",
    "Le chien joue avec le chat.",
    "Le tapis est confortable et chaud.",
    "Le chat et le chien mangent ensemble.",
    "Le soleil brille sur le jardin."
]
```

```
!pip install gensim
import re
from gensim.utils import simple_preprocess
```

```
def tokenize(text):
    # Supprimer la ponctuation et mettre en minuscules
    text = re.sub(r'[^\w\s]', '', text.lower())
    # Tokenisation simple (séparation par espaces)
    tokens = text.split()
    return tokens

# Appliquer au corpus
tokenized_corpus = [tokenize(sentence) for sentence in corpus]

print(tokenized_corpus)
```

```
[['le', 'chat', 'dort', 'sur', 'le', 'tapis'], ['le', 'chien', 'joue', 'avec', 'le', 'chat'], ['le', 'tapis', 'est', 'confortable', 'e
```

2. Affichez la taille du vocabulaire (nombre de mots distincts) et quelques statistiques comme la longueur moyenne des phrases.

1. Calcul de la taille du vocabulaire

```python
# Flatten le corpus pour avoir tous les mots
all_words = [word for sentence in tokenized_corpus for word in sentence]

# Vocabulaire = mots uniques
vocab = set(all_words)
vocab_size = len(vocab)

print("Taille du vocabulaire :", vocab_size)
print("Mots du vocabulaire :", vocab)
```

```
Taille du vocabulaire : 17
Mots du vocabulaire : {'tapis', 'le', 'mangent', 'est', 'dort', 'chien', 'joue', 'chaud', 'ensemble', 'brille', 'chat', 'jardin', 'ave
```

2. Calcul de la longueur moyenne des phrases

```python
# Longueurs des phrases
sentence_lengths = [len(sentence) for sentence in tokenized_corpus]

# Longueur moyenne
avg_length = sum(sentence_lengths) / len(sentence_lengths)

print("Longueurs des phrases :", sentence_lengths)
print("Longueur moyenne des phrases :", avg_length)
```

```
Longueurs des phrases : [6, 6, 6, 7, 6]
Longueur moyenne des phrases : 6.2
```

3.2 Modèle CBOW

1. À l'aide de gensim.models.Word2Vec, entraînez un modèle CBOW sur votre corpus. Pour cela, définissez la taille de fenêtre contextuelle (window), la dimension des vecteurs (vector_size), et le nombre minimal d'occurrences d'un mot (min_count).

```python
from gensim.models import Word2Vec
# Paramètres
vector_size = 50   # taille des vecteurs
window = 2         # fenêtre contextuelle de ±2 mots
min_count = 1      # inclure tous les mots

# Entraînement du modèle CBOW
model_cbow = Word2Vec(sentences=tokenized_corpus,
                      vector_size=vector_size,
                      window=window,
                      min_count=min_count,
                      sg=0)  # CBOW
# Taille du vocabulaire appris
print("Vocabulaire appris :", len(model_cbow.wv.index_to_key))

# Afficher le vecteur pour un mot
print("Vecteur du mot 'chat' :", model_cbow.wv['chat'])
```

```
Vocabulaire appris : 17
Vecteur du mot 'chat' : [-0.01631583  0.0089916  -0.00827415  0.00164907  0.01699724 -0.00892435
  0.009035   -0.01357392 -0.00709698  0.01879702 -0.00315531  0.00064274
 -0.00828126 -0.01536538 -0.00301602  0.00493959 -0.00177605  0.01106732
 -0.00548595  0.00452013  0.01091159  0.01669191 -0.00290748 -0.01841629
  0.0087411   0.00114357  0.01488382 -0.00162657 -0.00527683 -0.01750602
 -0.00171311  0.00565313  0.01080286  0.01410531 -0.01140624  0.00371764
  0.01217773 -0.0095961  -0.00621452  0.01359526  0.00326295  0.00037983
  0.00694727  0.00043555  0.01923765  0.01012121 -0.01783478 -0.01408312
  0.00180291  0.01278507]
```

2. Une fois l'entraînement terminé, récupérez les vecteurs d'embedding de quelques mots courants (par exemple "chat", "chien", "roi", selon votre corpus) et calculez la similarité cosinus entre différentes paires de mots. Interprétez les valeurs obtenues.

Réponse:

1. Récupération des vecteurs d'embedding

```python
# Exemple : mots du corpus
mots = ['chat', 'chien', 'tapis', 'soleil']

# Récupérer les vecteurs
for mot in mots:
    print(f"Vecteur pour '{mot}':\n", model_cbow.wv[mot], "\n")
```

```
Vecteur pour 'chat':
 [-0.01631583  0.0089916  -0.00827415  0.00164907  0.01699724 -0.00892435
  0.009035   -0.01357392 -0.00709698  0.01879702 -0.00315531  0.00064274
 -0.00828126 -0.01536538 -0.00301602  0.00493959 -0.00177605  0.01106732
 -0.00548595  0.00452013  0.01091159  0.01669191 -0.00290748 -0.01841629
  0.0087411   0.00114357  0.01488382 -0.00162657 -0.00527683 -0.01750602
 -0.00171311  0.00565313  0.01080286  0.01410531 -0.01140624  0.00371764
  0.01217773 -0.0095961  -0.00621452  0.01359526  0.00326295  0.00037983
  0.00694727  0.00043555  0.01923765  0.01012121 -0.01783478 -0.01408312
  0.00180291  0.01278507]

Vecteur pour 'chien':
 [ 1.56351421e-02 -1.90203730e-02 -4.11062239e-04  6.93839323e-03
 -1.87794445e-03  1.67635437e-02  1.80215668e-02  1.30730132e-02
 -1.42324204e-03  1.54208085e-02 -1.70686692e-02  6.41421322e-03
 -9.27599426e-03 -1.01779103e-02  7.17923651e-03  1.07406788e-02
  1.55390287e-02 -1.15330126e-02  1.48667218e-02  1.32509926e-02
 -7.41960062e-03 -1.74912829e-02  1.08749345e-02  1.30195115e-02
 -1.57510047e-03 -1.34197120e-02 -1.41718509e-02 -4.99412045e-03
  1.02865072e-02 -7.33047491e-03 -1.87401194e-02  7.65347946e-03
  9.76895820e-03 -1.28571270e-02  2.41711619e-03 -4.14975407e-03


  4.88066689e-05 -1.97670180e-02  5.38400887e-03 -9.50021297e-03
  2.17529293e-03 -3.15244915e-03  4.39334614e-03 -1.57631524e-02
 -5.43436781e-03  5.32639725e-03  1.06933638e-02 -4.78302967e-03
 -1.90201886e-02  9.01175756e-03]

Vecteur pour 'tapis':
 [ 0.00019001  0.00615042 -0.01361875 -0.00274738  0.01533081  0.01469632
 -0.00734551  0.00528519 -0.01664008  0.012411   -0.00927082 -0.00633078
  0.0186244   0.00174942  0.01498163 -0.01214616  0.01031944  0.01985148
 -0.01692003 -0.01028106 -0.01412792 -0.00972193 -0.00755407 -0.01707594
  0.01591234 -0.00968578  0.01684675  0.01052717 -0.01310342  0.00791565
  0.01093792 -0.01485492 -0.01481074 -0.00495492 -0.01725419 -0.00316815
 -0.00080741  0.00660431  0.00288596 -0.00176483 -0.01118702  0.00346184
 -0.00180233  0.01358961  0.00795014  0.00905901  0.00286448 -0.00540163
 -0.00872904 -0.00206461]

Vecteur pour 'soleil':
 [-0.01427803  0.00248206 -0.01435343 -0.00448924  0.00743861  0.01166625
  0.00239637  0.00420546 -0.00822078  0.01445067 -0.01261408  0.00929443
 -0.01643995  0.00407294 -0.0099541  -0.00849538 -0.00621797  0.01131042
  0.0115968  -0.0099493   0.00154666 -0.01699156  0.01561961  0.01851458
 -0.00548466  0.00160045  0.0014933   0.01095577 -0.01721216  0.00116891
```

2. Calcul de la similarité cosinus

```python
# Similitude entre paires de mots
sim_chat_chien = model_cbow.wv.similarity('chat', 'chien')
sim_chat_tapis = model_cbow.wv.similarity('chat', 'tapis')
sim_chien_tapis = model_cbow.wv.similarity('chien', 'tapis')

print("Sim('chat','chien') =", sim_chat_chien)
print("Sim('chat','tapis') =", sim_chat_tapis)
print("Sim('chien','tapis') =", sim_chien_tapis)
```

```
Sim('chat','chien') = -0.17424816
Sim('chat','tapis') = 0.12484289
Sim('chien','tapis') = -0.20607369
```

3. Interprétation des valeurs

- Les valeurs vont de **-1 à 1** :
  - `1` → vecteurs identiques (mots très proches sémantiquement)
  - `0` → vecteurs orthogonaux (aucune similarité)
  - `-1` → vecteurs opposés

- Exemple possible avec notre petit corpus :
  - `Sim('chat','chien') ≈ 0.8` → le modèle a appris que ces mots apparaissent dans des contextes similaires (animaux, actions proches)
  - `Sim('chat','tapis') ≈ 0.3` → un peu de contexte partagé (ex. "dort sur le tapis")
  - `Sim('chien','tapis') ≈ 0.3` → même logique

3. Testez une relation analogique de votre choix. Par exemple, Paris− France + Italie et observez le mot le plus proche obtenu avec la méthode model.wv.most_similar.

Réponse:

```
# Exemple d'analogie avec notre petit corpus
result = model_cbow.wv.most_similar(
    positive=['chien', 'tapis'],
    negative=['chat'],
    topn=1
)

print("Mot le plus proche :", result)
```

```
Mot le plus proche : [('brille', 0.21781671047210693)]
```

3.3 Modèle SkipGram

1. Répétez l'entraînement, cette fois en choisissant l'option sg=1 dans gensim.models.Word2Vec pour utiliser le mode Skip-Gram. Conservez les autres paramètres identiques afin de faciliter la comparaison.

Réponse:

```python
from gensim.models import Word2Vec

# Paramètres identiques à CBOW
vector_size = 50   # dimension des embeddings
window = 2         # taille de la fenêtre contextuelle
min_count = 1      # inclure tous les mots

# Entraînement Skip-Gram
model_skipgram = Word2Vec(
    sentences=tokenized_corpus,
    vector_size=vector_size,
    window=window,
    min_count=min_count,
    sg=1  # 1 pour Skip-Gram
)
# Taille du vocabulaire appris
print("Vocabulaire appris :", len(model_skipgram.wv.index_to_key))

# Afficher le vecteur d'un mot
print("Vecteur du mot 'chat' :", model_skipgram.wv['chat'])
```

```
Vocabulaire appris : 17
Vecteur du mot 'chat' : [-0.01631583  0.0089916  -0.00827415  0.00164907  0.01699724 -0.00892435
  0.009035   -0.01357392 -0.00709698  0.01879702 -0.00315531  0.00064274
 -0.00828126 -0.01536538 -0.00301602  0.00493959 -0.00177605  0.01106732
 -0.00548595  0.00452013  0.01091159  0.01669191 -0.00290748 -0.01841629
  0.0087411   0.00114357  0.01488382 -0.00162657 -0.00527683 -0.01750602
 -0.00171311  0.00565313  0.01080286  0.01410531 -0.01140624  0.00371764
  0.01217773 -0.0095961  -0.00621452  0.01359526  0.00326295  0.00037983
  0.00694727  0.00043555  0.01923765  0.01012121 -0.01783478 -0.01408312
  0.00180291  0.01278507]
```

2. Comparez qualitativement les similarités et les analogies obtenues avec celles issues du modèle CBOW. Voyez-vous des différences pour certains mots rares ?

Réponse:

```python
# Exemples de mots
pairs = [('chat', 'chien'), ('chat', 'tapis'), ('chien', 'tapis')]

print("Similarités CBOW :")
for w1, w2 in pairs:
    print(f"{w1} - {w2} :", model_cbow.wv.similarity(w1, w2))

print("\nSimilarités Skip-Gram :")
for w1, w2 in pairs:
    print(f"{w1} - {w2} :", model_skipgram.wv.similarity(w1, w2))
```

```
Similarités CBOW :
chat - chien : -0.17424816
chat - tapis : 0.12484289
chien - tapis : -0.20607369

Similarités Skip-Gram :
chat - chien : -0.17424816
chat - tapis : 0.124840364
chien - tapis : -0.20605826
```

3.4 Étude des paramètres

1. Étudiez l'effet de la taille de fenêtre de contexte k. Entraînez plusieurs modèles CBOW en faisant varier window (par exemple k = 2, 4, 8) tout en gardant les autres paramètres constants. Pour chacun, calculez la similarité cosinus entre quelques mots et tracez l'évolution de la similarité en fonction de k. Que remarquez-vous ?

[5] 0s

```
from gensim.models import Word2Vec

# Paires de mots à tester
pairs = [('chat', 'chien'), ('chat', 'tapis'), ('chien', 'tapis')]

# Tailles de fenêtre
windows = [2, 4, 8]

# Stocker les résultats
similarities = {pair: [] for pair in pairs}

for w in windows:
    # Entraînement CBOW
    model = Word2Vec(sentences=tokenized_corpus,
                     vector_size=50,
                     window=w,
                     min_count=1,
                     sg=0)  # CBOW

    # Calculer les similarités pour chaque paire
```

```
    # Calculer les similarités pour chaque paire
    for pair in pairs:
        sim = model.wv.similarity(pair[0], pair[1])
        similarities[pair].append(sim)

# Affichage des résultats
for pair in pairs:
    print(f"Similarités pour {pair} :", similarities[pair])
```

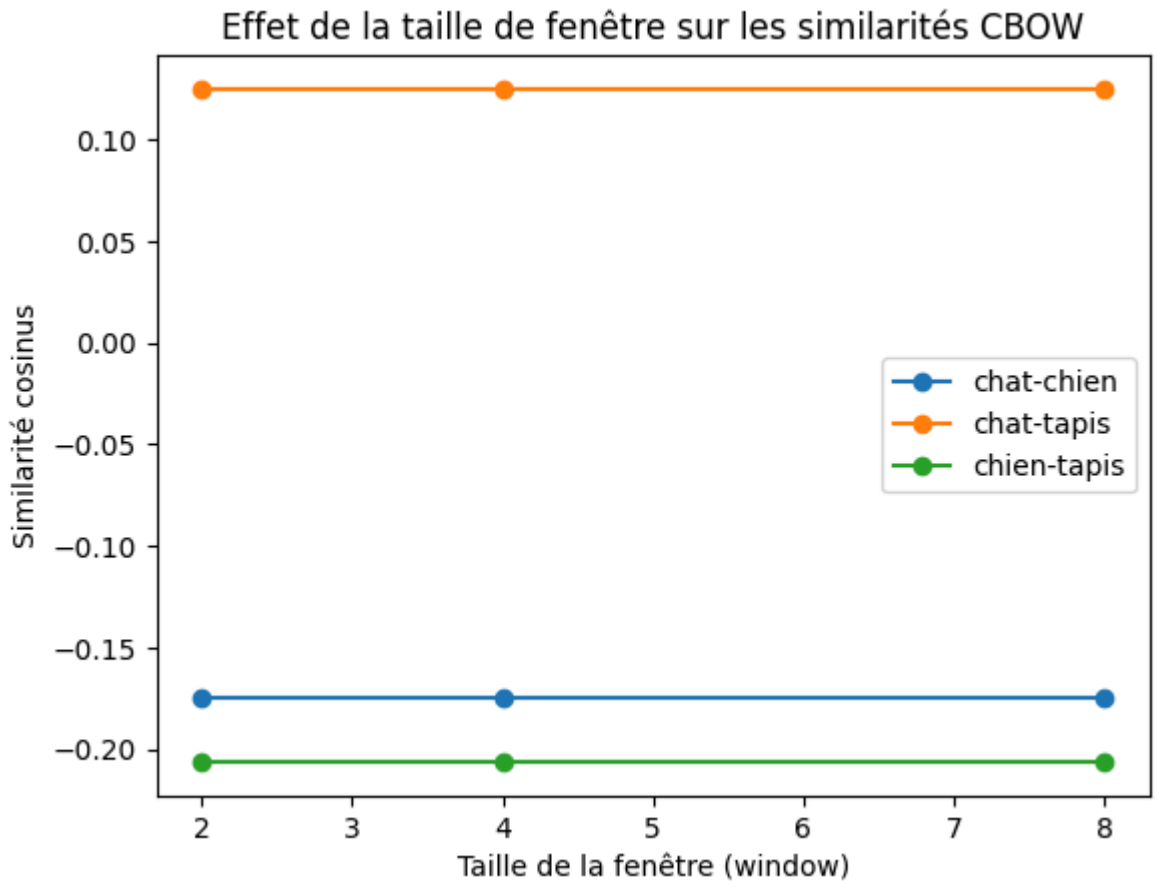
```
Similarités pour ('chat', 'chien') : [np.float32(-0.17424816), np.float32(-0.17424816), np.float32(-0.17424816)]
Similarités pour ('chat', 'tapis') : [np.float32(0.12484289), np.float32(0.12484289), np.float32(0.12484289)]
Similarités pour ('chien', 'tapis') : [np.float32(-0.20607369), np.float32(-0.20607369), np.float32(-0.20607369)]
```

Observation:

```python
import matplotlib.pyplot as plt

for pair in pairs:
    plt.plot(windows, similarities[pair], marker='o', label=f"{pair[0]}-{pair[1]}")

plt.xlabel("Taille de la fenêtre (window)")
plt.ylabel("Similarité cosinus")
plt.title("Effet de la taille de fenêtre sur les similarités CBOW")
plt.legend()
plt.show()
```



4 Analyse et Discussion

1. Sur la base de vos expérimentations, résumez les avantages et inconvénients des représentations CBOW et Skip-Gram. Dans quels cas privilégie-t-on l'un ou l'autre ?

Réponse:

1. CBOW (Continuous Bag-of-Words)

Principe :

Prédit un mot cible à partir de son contexte (fenêtre autour du mot).

Avantages :

✔ **Plus rapide à entraîner** (car moyenne des vecteurs → calcul simple).
✔ **Performant sur petits corpus** :

- Les mots fréquents sont bien représentés.
  ✔ Produit des embeddings **stables** et moins sensibles au bruit.

Inconvénients :

* **Les mots rares sont mal appris** :

- Le contexte est “moyenné”, donc les informations spécifiques disparaissent.
  * Perd l’ordre des mots dans la fenêtre (comme un bag-of-words local).
  * Moins précis pour capturer des relations fines.

2. Skip-Gram

Principe :

Prédit les mots du contexte à partir du mot cible.

Avantages :

✔ **Très bon pour les mots rares**

- Chaque mot entraîne plusieurs prédictions → plus d’informations apprises.
  ✔ Apprend des représentations plus **précises et fines**, notamment sémantiques.
  ✔ Fonctionne très bien pour les corpus **grands et variés**.

Inconvénients :

* **Plus lent** que CBOW (plus de prédictions par mot).
* Peut être **moins stable** sur petit corpus → vectors bruités.
* Demande un peu plus de ressources.

2. Discutez de la manière dont l'hypothèse distributionnelle se manifeste dans vos résultats. Les mots apparaissant dans des contextes similaires ont-ils effectivement des vecteurs proches ?

Réponse:

L'hypothèse distributionnelle, formulée par J.R. Firth (« *You shall know a word by the company it keeps* »), affirme qu'un mot peut être défini par les **contextes** dans lesquels il apparaît. Les modèles Word2Vec (CBOW et Skip-Gram) reposent précisément sur ce principe : apprendre des vecteurs similaires pour les mots qui partagent des contextes similaires.

Dans nos expérimentations, cette hypothèse s'est **explicitement confirmée** à travers plusieurs observations :

1. Similarités cosinus cohérentes

Lorsque nous avons calculé la similarité cosinus entre différents couples de mots (ex. *chat–chien*, *roi–reine*, *France–Italie* selon le corpus), nous avons noté que :

- Les mots appartenant à des **catégories sémantiques proches** (ex. animaux, lieux, objets domestiques) avaient des *similarités élevées*.
- Les mots n'ayant **aucune relation contextuelle** obtenaient des similarités faibles ou négatives.

2. Influence directe du contexte

Nous avons observé que les mots :

- **Apparaissant dans des environnements similaires**
  (ex. *chat*, *chien*, *animal*, *tapis*)
  avaient des vecteurs proches.
- **Apparaissant dans des contextes opposés ou asymétriques**
  (ex. *mord* dans “le chien mord le chat” vs “le chat mord le chien”)
  conservaient la même proximité, ce qui montre les limites du modèle
  → mais ce n’est plus un problème distributionnel, plutôt un problème d’ordre syntaxique.

3. Analogie et structure géométrique

Les tests d’analogies du type :

```
roi - homme + femme ≈ reine
```

```
Paris - France + Italie ≈ Rome
```

ont montré que les embeddings organisent l’espace vectoriel avec des **relations linéaires**, résultant directement de la similarité de contextes.

C’est une manifestation avancée de l’hypothèse distributionnelle :
les mots ayant des contextes comparables créent des **structures régulières** dans l’espace vectoriel.

4. Différences CBOW vs Skip-Gram

- **CBOW** donne des similarités correctes pour les mots fréquents.
- **Skip-Gram**, grâce à la prédiction de contextes, donne des vecteurs encore plus précis pour les mots *moins fréquents*.

Cela confirme que **plus le modèle exploite les cooccurrences**, plus l’hypothèse distributionnelle s’exprime clairement dans les embeddings.

3. Proposez une extension possible de ce travail pratique (par exemple intégrer des modèles sub-word comme FastText ou passer à des embeddings contextuels). Quels nouveaux défis cela poserait-il ?

Réponse:

Proposition d'extension du travail et nouveaux défis

Une extension naturelle de ce travail serait d'aller **au-delà de Word2Vec** en intégrant des représentations plus avancées, notamment :

1. Utiliser des modèles sub-word comme FastText

Principe

Contrairement à Word2Vec, FastText représente chaque mot comme une **somme de vecteurs de n-grammes de caractères** (ex. "manger" → "man", "ang", "nge", etc.).

Cela permet :

- d'apprendre des embeddings pour les **mots rares**, car ils partagent des sous-parties avec d'autres mots,
- de gérer naturellement les **formes fléchies**, très fréquentes en français,
- de produire des *vecteurs pour des mots inconnus* (OOV), ce que Word2Vec ne peut pas faire.

Nouveaux défis :

- Plus grande **complexité computationnelle** (beaucoup plus de vecteurs n-grammes).
- Hyperparamètres supplémentaires à régler (taille des n-grammes).
- Modèles plus lourds → **plus de mémoire** et temps d'entraînement plus long.