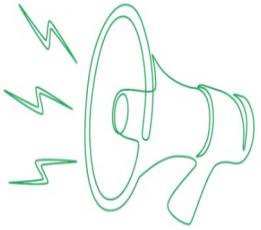
Classifiez automatiquement des biens de consommation

Projet 5

Sofia Velasco



Objectif:

Tester la faisabilité d'un moteur de classification, qui basé sur une image et une description puisse attribuer leur catégorie aux articles.

A. Caractéristiques générales des données

15 colonnes : On en garde 4

uniq_id	object		1050
crawl_timestamp	object		1050
product_url	object		1050
product_name	object	✓	1050
product_category_tree	object	✓	1050
pid	object		1050
retail_price	float64		1049
discounted_price	float64		1049
image	object	✓	1050
is_FK_Advantage_product	bool		1050
description	object	✓	1050
product_rating	object		1050
overall_rating	object		1050
brand	object		712
product_specifications	object		1049

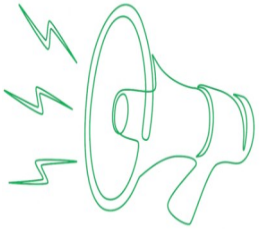
Pas de données manquantes pour les colonnes qui nous intéressent!

B. Extraction des ‘features’ texte

5 approches:

2 type **“bag-of-words”**:
- Countvectorizer
- Tf_idf

3 type **“word/sentence embedding”**:
- Word2Vec
- BERT
- USE (Universal Sentence Encoder).



B.1. Prétraitement

(conversion des données en quelque chose qu'un ordinateur peut comprendre)

4 Fonctions de prétraitement → construction des variables :

- 'product_name_bow_lem' + 'description_bow_lem' → Countvectorizer, Tf_idf, Word2Vec
- 'product_name_dl' + 'description_dl' → BERT et USE

1. Tokenizer → convertit/sépare phrases/commentaires en tokens
(ie. unités plus petites, qui peuvent être des mots, caractères ou sous-mots).

2. Stop_words → filtre mots inutiles ou 'vides' (ex: "le", "un", "une", "dans") économiser espace et temps de traitement.
'nltk' (Natural Language Toolkit) liste dans 16 langues différentes, ici l'anglais.

→ N'applique pas pour BERT et USE

3. lower case et alpha → met tout en minuscules et supprime '@' et 'http'.

4. Lemmatizer → joint des mots similaires pour les analyser ensemble

→ N'applique pas pour BERT et USE

B.2 Identification des 'main_category'

Premier élément du 'product_category_tree' :

- 7 catégories
- A comparer avec celles obtenues par notre automatisation

```
'Kitchen & Dining',
'Beauty and Personal Care',
'Computers',
'Watches',
'Home Furnishing',
'Baby Care',
'Home Decor & Festive Needs'
```

B.3 Approches de type 'bag of words'

- Représentation de texte décrivant occurrence de mots dans un document.
- **Implique:**
  - vocabulaire de mots connus
  - mesure présence de mots connus.
- **Procédure à suivre:**

  1. **Collecte** de la data (chaque ligne du dataset → 'un document').
  2. Élaboration de la **liste de mots** (ie. le vocabulaire).
  3. Scorer les mots dans chaque document → transformer chaque 'document' (ie. ligne) en un **vecteur**.

     Notation la plus simple (CountVectorizer) marque la présence de mots:
     - 0 pour absent
     - 1 pour présent
  4. Réduction de dimension : PCA et t-SNE
  5. Calcul de ARI entre les 'main_category' et les clusters obtenus
  6. Graphiques

- **Limitations:**
  - Vocabulaire (choisir avec soin)
  - Sparsity (grande surface de temps et d'espace par rapport à la quantité)
  - Signification (ignorer l'ordre des mots et donc le contexte)

B.3 Approches de type 'bag of words'

3 différentes combinaisons pour chaque méthode:

1. Uniquement sur 'product_name_bow_lem';
2. Uniquement sur 'description_bow_lem';
3. Sur 'product_name_bow_lem' et 'description_bow_lem (ie. Sur 'product_description_bow_lem').

Et on utilise 2 méthodes:

- **CountVectorizer** → uniquement sur la fréquence des mots et non l'importance des mots.
- **TfidfVectorizer** → fourni l'importance des mots → trouve les mots qui spécifient un thème, et supprime les moins importants (modèle moins complexe réduction dimensions d'entrée).

Note:
- max_df (float [0.0, 1.0] ou int) -> ignore termes qui ont une fréquence de document strictement supérieure au seuil donné.
- min_df (float [0.0, 1.0] ou int) -> ignore termes qui ont une fréquence de document strictement inférieure au seuil donné.
- Document ici signifie ligne du data set.

B.3 Approches de type 'bag of words'

ATTENTION

- **PCA:** non utilisé pour des questions de mémoire → erreur car trop de 'colonnes' (dimensions) dans nos matrices.
- **t-distributed Stochastic Neighbor Embedding (t-SNE):** réduction de dimension pour la visualisation de données en 2D et 3D, pouvant trouver des connexions non linéaires dans les données.

  n_components= Dimension de l'espace.

  perplexity= nombre de voisins les plus proches.

  learning_rate= [10 et 1000], 200 est l'auto, doit être ni trop grand ni trop petit.

  random_state= Détermine le générateur de nombres aléatoires. Différentes initialisations entraînent différents minima locaux de la fonction de coût.

- **ARI:** compare deux clustérisations. Bon (>=0.8) s'il y a une correspondance (un accord) entre deux segmentations (clustering).

  On veut des clusters qui reproduisent le mieux les clusters originaux→mauvais ARI montrant des graphiques qui différent trop n'est pas bon.

B.3 Approches de type 'bag of words'

ATTENTION

- **La forme générale des deux graphiques** (celui des catégories et celui des clusters obtenus avec notre modèle) **est la même**. Suite au t-SNE chaque ligne a un point fixe associé, ce qui varie entre eux est juste la couleur associée à chaque point.

Quelle méthode choisir ?

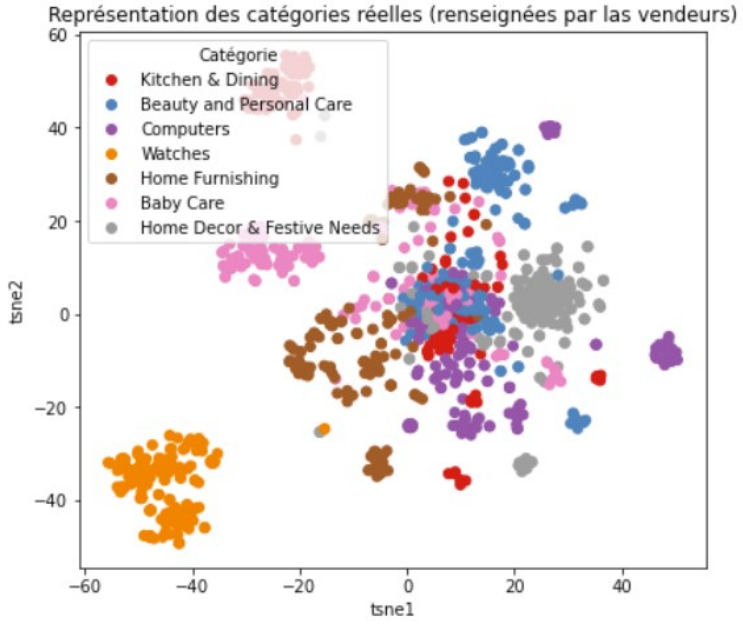
- Prend le moins de temps et
- Clusters bien séparés qui prédisent le mieux les catégories renseignées manuellement
  (ie. meilleur ARI → moins de différences entre graphiques).

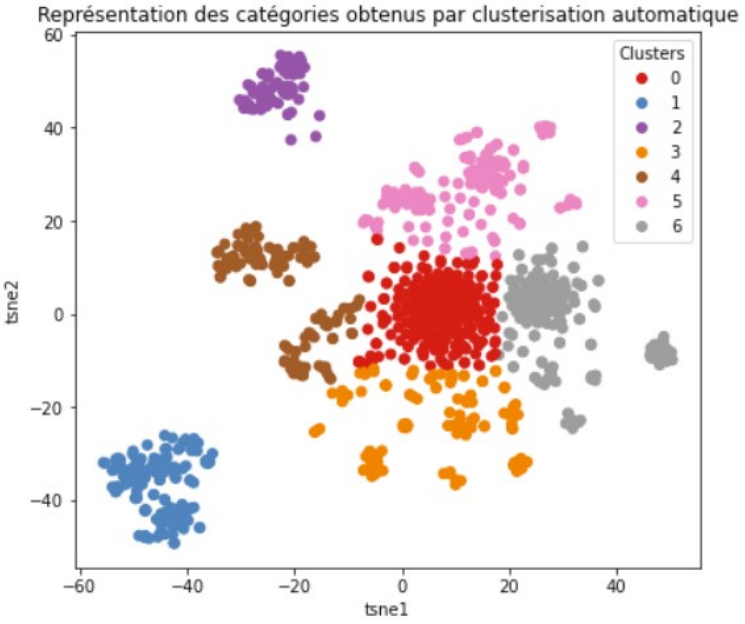
```
CountVectorizer :
-----------------
Uniquement sur 'product_name_bow_lem':
ARI :  0.3608 time :  7.0
Uniquement sur 'description_bow_lem':
ARI :  0.3678 time :  8.0
Sur 'product_name_bow_lem' et 'description_bow_lem:
ARI :  0.399 time :  8.0

Tf-idf :
--------
Uniquement sur 'product_name_bow_lem':
ARI :  0.5041 time :  7.0
Uniquement sur 'description_bow_lem':
ARI :  0.5154 time :  7.0
Sur 'product_name_bow_lem' et 'description_bow_lem:
ARI :  0.5596 time :  6.0
```

Les temps sont en secondes

B.3.1 CountVectorizer
('bag of words')

Uniquement sur **'product_name_bow_lem' :**

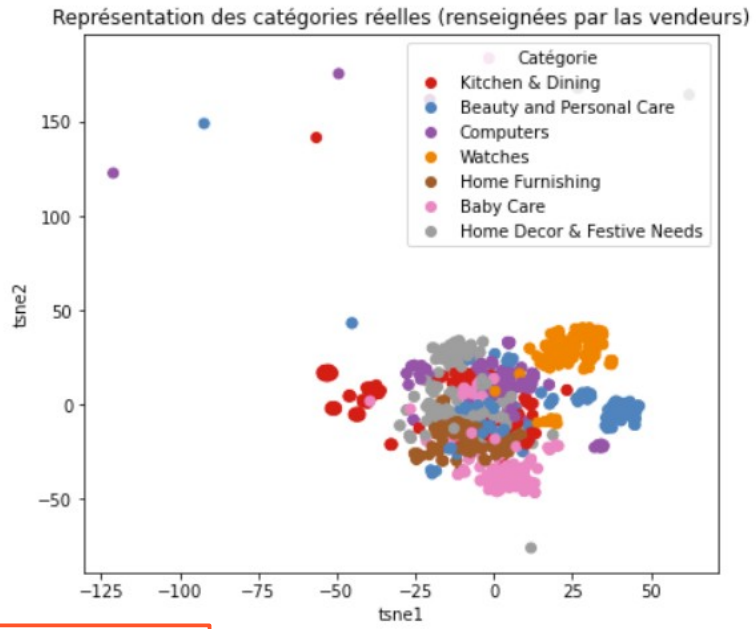


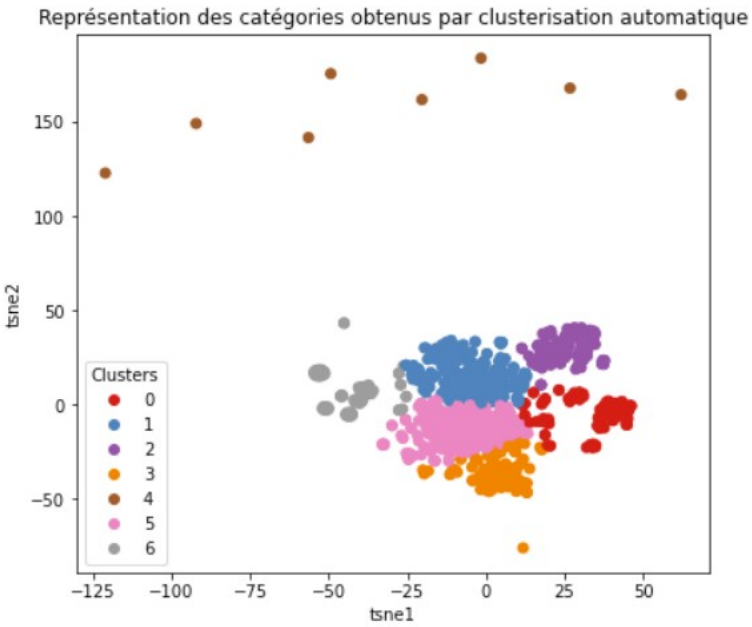


ARI : 0.3608 → Mauvais !

B.3.1 CountVectorizer
('bag of words')

Uniquement sur **'description_bow_lem' :**

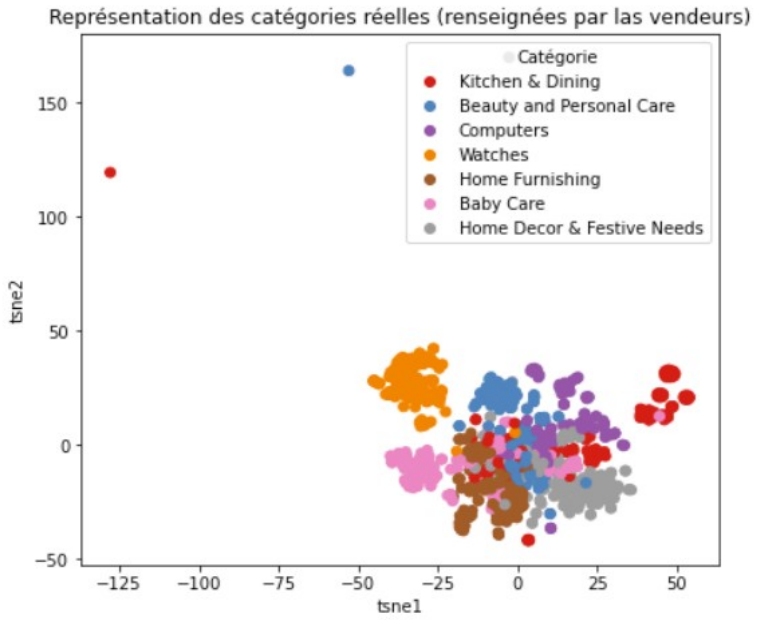


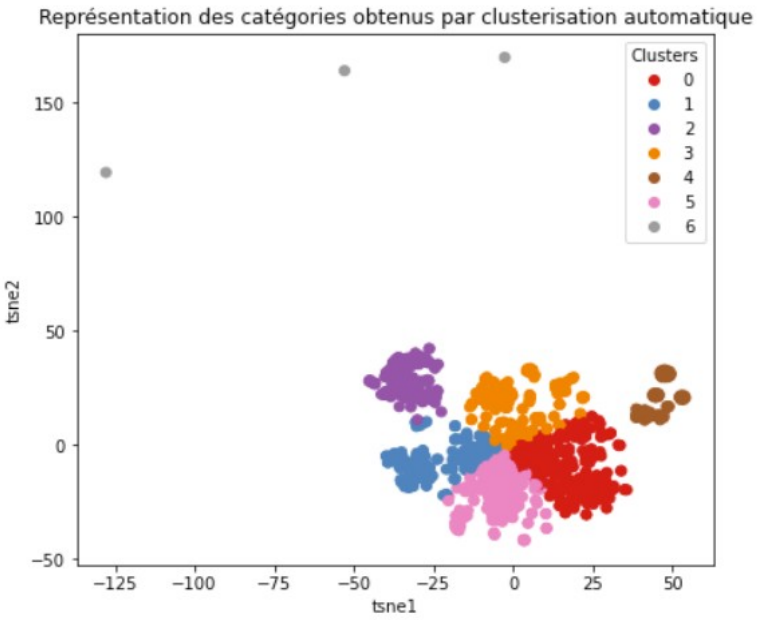


ARI : 0.3678 → Mauvais !

B.3.1 CountVectorizer
('bag of words')

Sur **'product_name_bow_lem'** et **'description_bow_lem :**

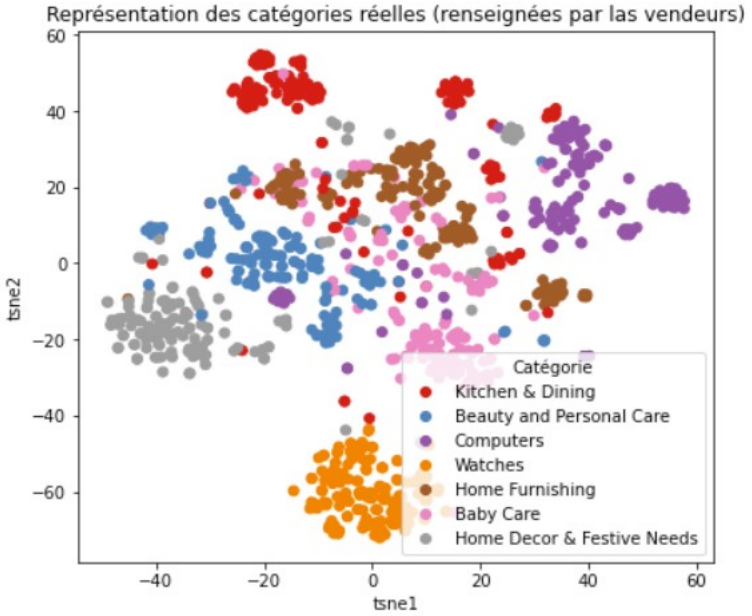


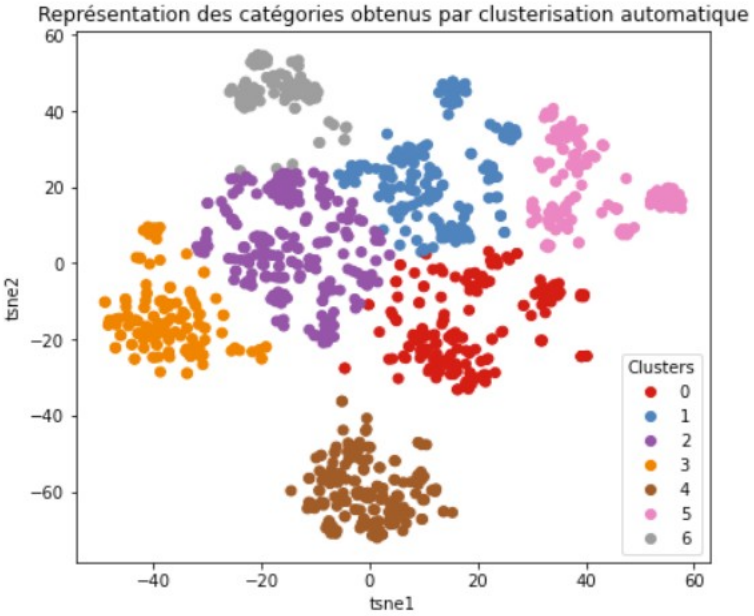


ARI : 0.399 → Mauvais !

B.3.2 TfidfVectorizer
('bag of words')

Uniquement sur **'product_name_bow_lem' :**



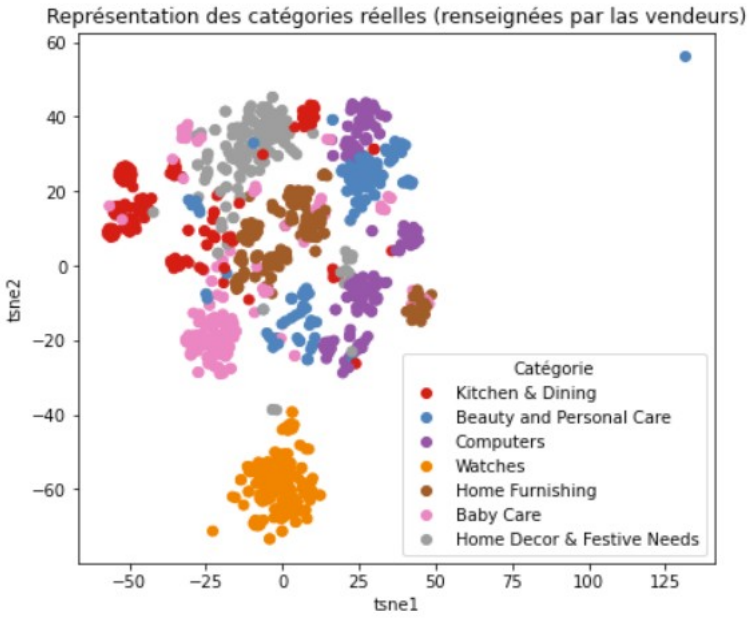


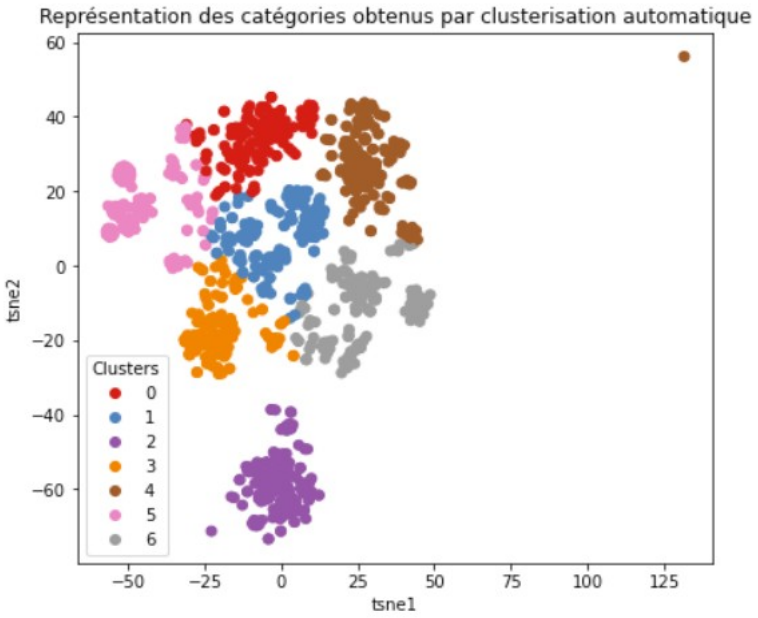
ARI : 0.5041

Déjà meilleur !

B.3.2 TfidfVectorizer
('bag of words')

Uniquement sur **'description_bow_lem' :**

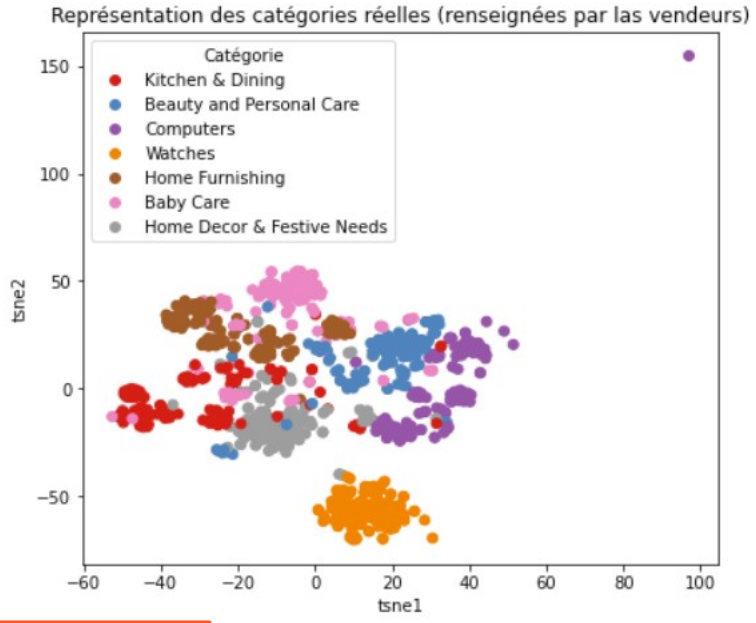


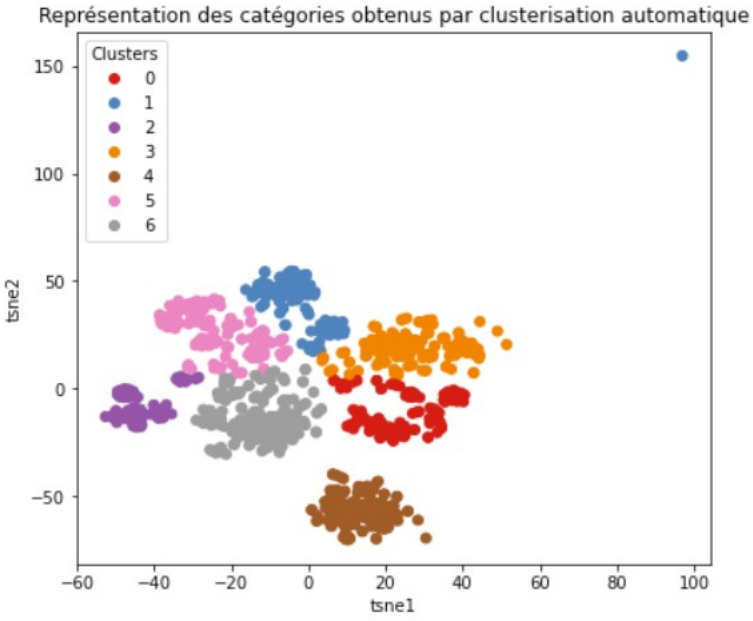


ARI : 0.5154 → Déjà meilleur !

B.3.2 TfidfVectorizer
('bag of words')

Sur **'product_name_bow_lem'** et **'description_bow_lem :**





ARI : 0.5596 → Déjà meilleur !

B.4 Approches de type 'word/sentence embedding'

- On construit des **vecteurs de faible dimension** (des 'embedding') qui représente une phrase (ou un mot) dans un format dense.
- **3 méthodes:** Word2Vec, BERT et USE.
- **3 différentes combinaisons pour chaque méthode:** (on présentera que la meilleure, meilleur ARI)
  1. Uniquement sur 'product_name_bow_lem';
  2. Uniquement sur 'description_bow_lem';
  3. Sur 'product_name_bow_lem' et 'description_bow_lem (ie. Sur 'product_description_bow_lem').

B.4.1 Word2Vec
('word/sentence embedding')

- Permet à des mots similaires d'avoir des dimensions similaires (vecteur associé similaire), apportant du contexte.
  Les mots de sens similaire sont plus proches dans l'espace, indiquant leur similitude sémantique (ex: femmes, hommes et humains seront regroupés dans un coin, jaune, rouge et bleu dans un autre)

- **Processus:**

1. Création et entraînement du modèle Word2Vec.

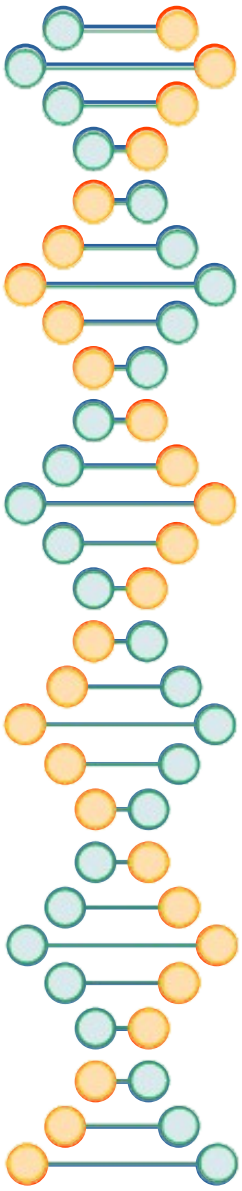
On utilise le packaging **'gensim'** :

```
#1. Uniquement sur 'product_name_bow_lem'.
w2v_model1 = gensim.models.Word2Vec(min_count=w2v_min_count, window=w2v_window,
                                    vector_size=w2v_size,
                                    seed=42,
                                    workers=1)

w2v_model1.build_vocab(sentences1)
w2v_model1.train(sentences1, total_examples=w2v_model1.corpus_count, epochs=w2v_epochs)
model_vectors1 = w2v_model1.wv
w2v_words1 = model_vectors1.index_to_key
print("Vocabulary size: %i" % len(w2v_words1))
```

-**.build_vocab** → façon dont le modèle découvre l'ensemble de tous les mots possibles et trouve quels mots apparaissent plus de min_count fois. Essentiel de le mettre avant '.train'.

-**.corpus_count** = donne le nombre total de mots.

2. Préparation des 'sentences' (tokenisation).



B.4.1 Word2Vec
('word/sentence embedding')

4. Création de la matrice d'embedding.

Un mot dans un espace → ~ 300 dimensions (ou 'embeddings' du mot). Les relations (le sens) entre les mots se représentent en comparant les 'embeddings' des mots.

Une **'embedding matrix'** → liste de tous les mots et de leurs 'embeddings'. Chaque ligne → un mot ; chaque colonne → une dimension (axe).

Stockage de l'info de manière dense → liste de mots et d'ID de ligne correspondante (Ex: { hello: 0, there: 1, texas: 2, world: 3, … })

Si entraînée avec succès, les synonymes sont proches.

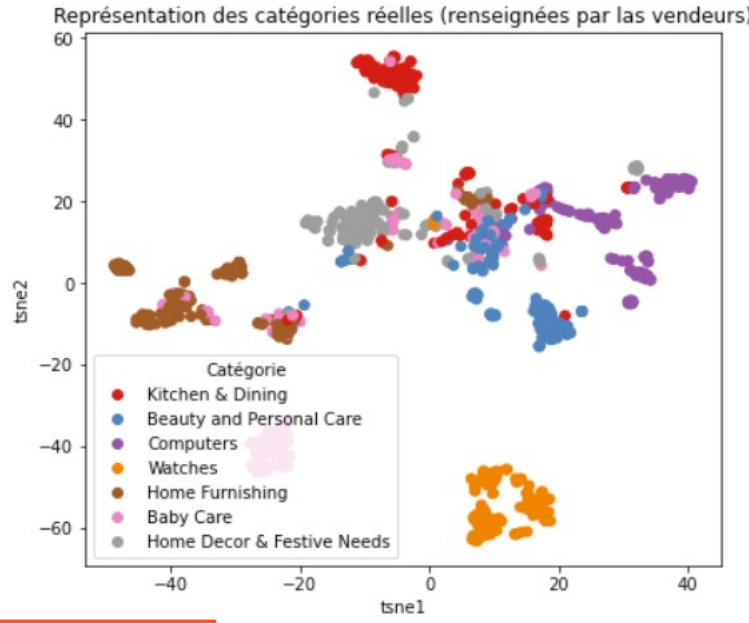
5. Création du modèle embedding.

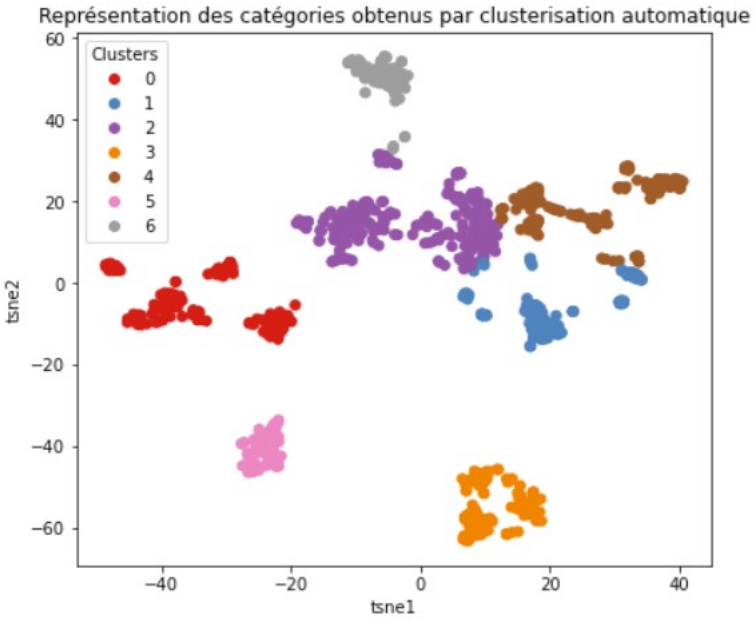
6. Exécution du modèle, calcul de l'ARI et graphiques.

B.4.1 Word2Vec
('word/sentence embedding')

Meilleur

Uniquement sur **'product_name_bow_lem' :** (le moins de mots le mieux)





ARI : 0.4938

Déjà meilleur !

time : 6.0

B.4.2 BERT

('word/sentence embedding')

- **Word2Vec** mappe chaque mot sur un vecteur → **les mots sont ‘fixés’ à un vecteur ou signification**

- **BERT** → méthode de modélisation du **langage masqué** qui empêche le mot de se "voir" (ie. d'avoir une signification fixe indépendante de son contexte). Les **mots** sont **définis par leur environnement,** et non par une identité préfixée.

- On peut utiliser BERT depuis les modules:**'transformers' (de hugging face)** ou **'tensorflow_hub**'.
  - 'hub' peut avoir plus de modèles déjà entraînés.
  - on utilise BERT de **'transformers' (de hugging face)**, avec **'base-uncased**' c'est à dire général :

```
model_type = 'bert-base-uncased'
model = TFAutoModel.from_pretrained(model_type)
```

- **Processus :**

  1. Préparation des sentences.

  2. Création des features.

  3. Calculs des ARI.

  4. Graphiques.

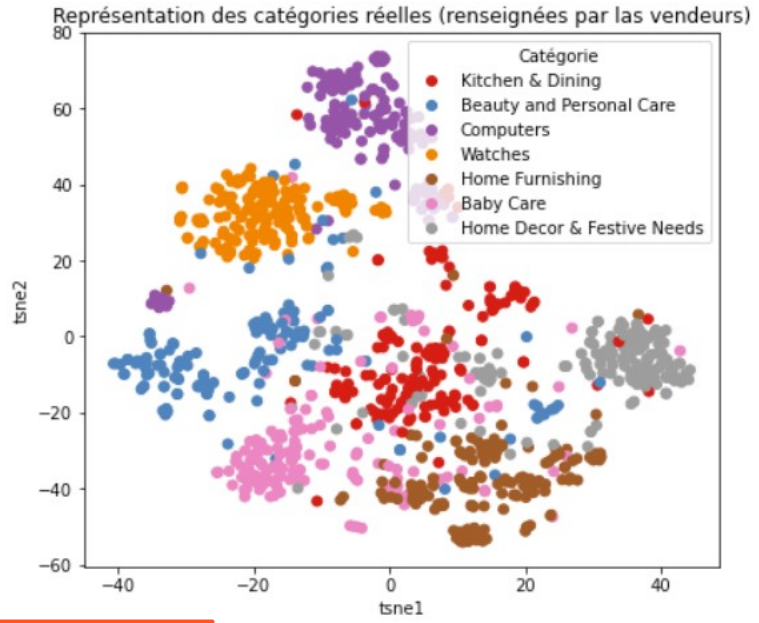
⚠ Plus lent que Word2Vec
Données de pré-traitées sans traitements des Stop_wordsots ni de Lemmatizer → Contexte est important

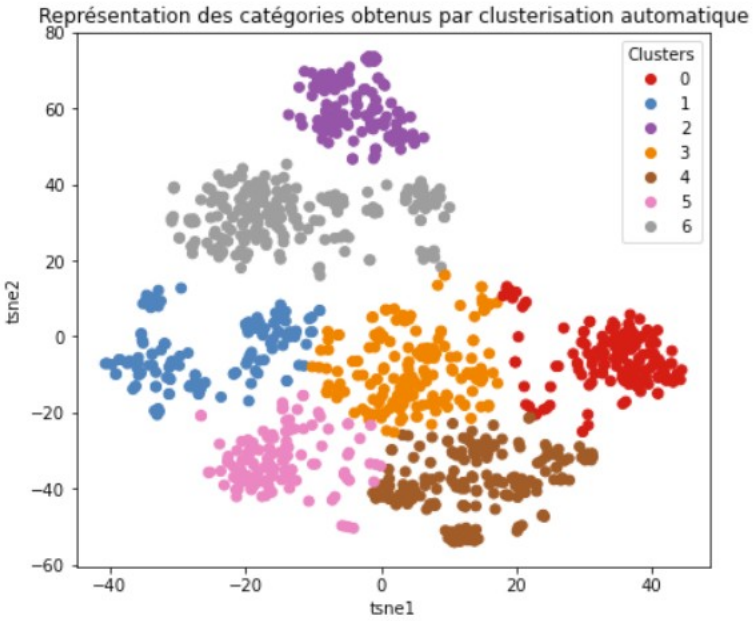
B.4.2 BERT

('word/sentence embedding')

Meilleur

Uniquement sur **'product_name_bow_lem' :** (le moins de mots le mieux)





ARI : 0.582

Déjà meilleur !

time : 8.0

B.4.3 USE

('word/sentence embedding')

- Universal Sentence Encoder (**USE**), **encode le texte dans des vecteurs de grande dimension** qui peuvent être utilisés pour :
  - la classification de texte,
  - la similarité sémantique,
  - le regroupement et
  - d'autres tâches en langage naturel.

- Contrairement à Word2Vec, un peu comme BERT, USE **inclut le contexte entier de la phrase dans la création des vecteur**.

  ⚠ Données de pré-traitées sans traitements des Stop_wordsots ni de Lemmatizer → Contexte est important

- **Processus :**

  On utilise le modèle 'USE' disponible dans le packaging 'tensorflow_hub', puis on calcule le ARI et les graphiques.

```
import tensorflow_hub as hub

embed = hub.load("..\\Data\\use")
```

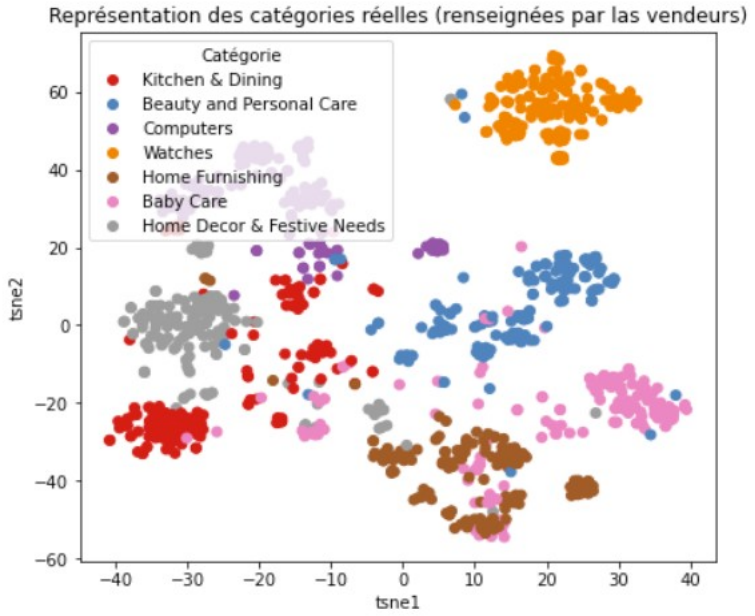
```
#a. Fonction générale:

def feature_USE_fct(sentences, b_size) :
    batch_size = b_size
    time1 = time.time()

    for step in range(len(sentences)//batch_size) :
        idx = step*batch_size
        feat = embed(sentences[idx:idx+batch_size])

        if step ==0 :
            features = feat
        else :
            features = np.concatenate((features,feat))

    time2 = np.round(time.time() - time1,0)
    return features
```

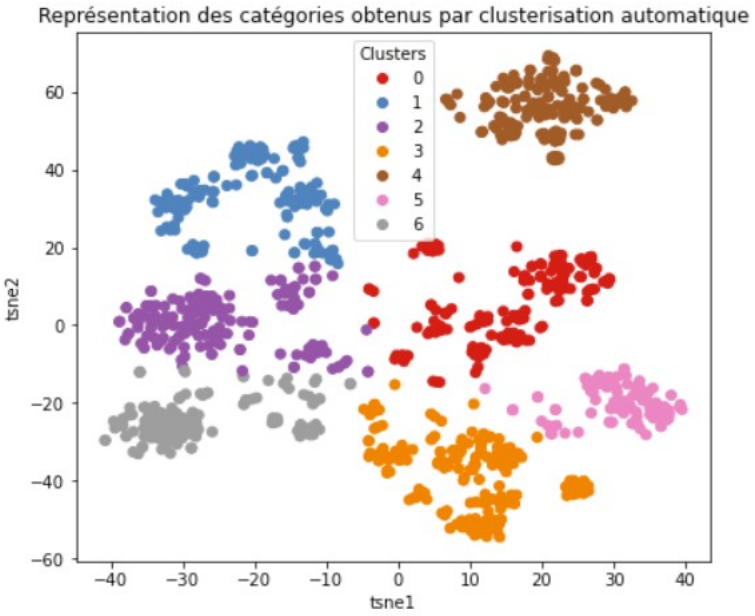
Meilleur
des Meilleurs

B.4.3 USE

('word/sentence embedding')

Uniquement sur **'product_name_bow_lem' :** (le moins de mots le mieux)





ARI : 0.6426

Le meilleur jusque là !

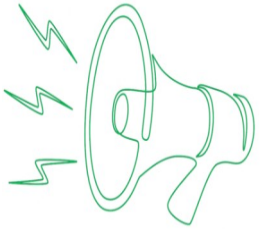
time : 8.0

C. Extraction des ‘features’ image

2 approches:

“SIFT” → plus performant de ce genre d’algorithmes dans la plupart des scénarios

“CNN Transfer Learning”



C.0 Affichage des images

‘imshow ‘ et ‘imread’ → 2 commandes qui permettent d’afficher les images:
plt.imshow(imread(filename))

```
for name in l_cat :
    print(name)
    # print("-------")
    for i in range(3):
        plt.subplot(130 + 1 + i)
        filename = path + list_fct(name)[i+10]
        image = imread(filename)
        plt.imshow(image)
    plt.show()
```

C.1 SIFT

- Méthode invariante par rapport à rotation et échelle. On utilise → **'cv.SIFT'.**
- Processus: **6 étapes:**

1. Prétraitement des images.

Passage à l'**échelle de gris (cv2.imread)**, et **équalisation (ie. ajustement du contraste) (cv2.equalizeHist)**.

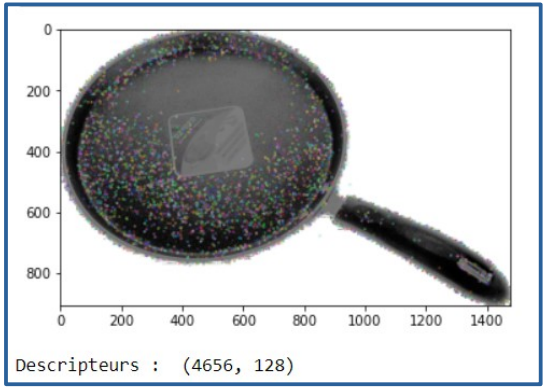
Note: **cv.SIFT a besoin en entrée d'une image de 8bits en échelle de gris.**

Ensuite **'sift.detectAndCompute'** fait les 3 étapes suivante intrinsèques à l'algorithme STIF:

2. Détection des descripteurs sur l'ensemble des images.

Zones circulaires (rayon ~ facteur d'échelle)→ résultant de la convolution par filtre gaussien qui lisse l'image (DoG – Difference of Gaussians)→ invariants à la rotation de dimension 128.

Note: On limite le nombre de descripteurs par image à 500 (pour avoir les plus remarquables).



1050 images x 500 ≈ 525 000 descripteurs pour l'ensemble des images

```
Nombre de descripteurs : (517351, 128)
temps de traitement SIFT descriptor :          364.70 secondes
```

C.1 SIFT

3. Création des clusters de descripteurs

On utilise MiniBatchKMeans pour des résultat plus rapides qu'avec K-means.

```
Nombre de clusters optimal:  719
```

4. Création des features (ie. variables caractéristiques) des images.

Pour chaque image on crée un histogramme du nombre de descripteurs par clusters ; puis une matrice d'histogrammes pour toutes les images.

```
1  im_features.shape
(1050, 719)
```

⇒ on a 1050 images (lignes) et 719 dimensions, features par images

5. Réductions de dimension.

PCA puis t-SNE (pour que le t-SNE -réduction à 2D- soit plus rapide).

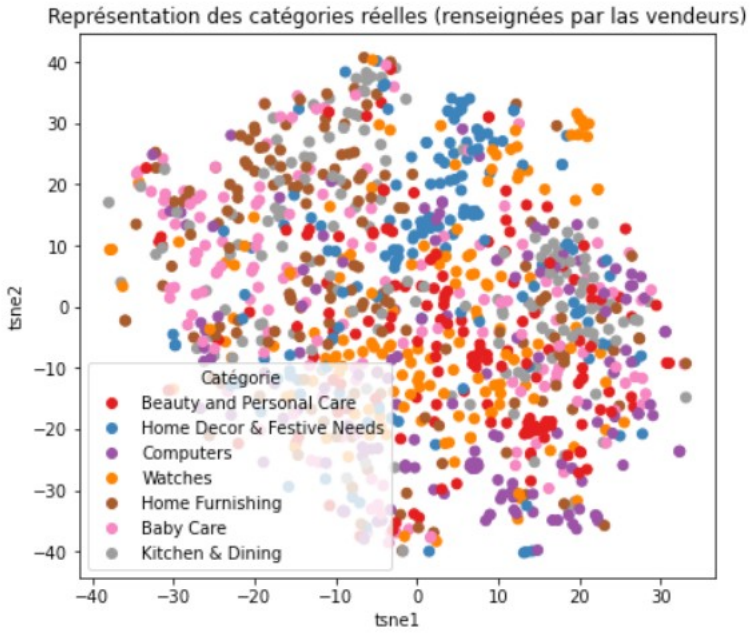
```
Dimensions dataset avant réduction PCA :  (1050, 719)
Dimensions dataset après réduction PCA :  (1050, 499)
```

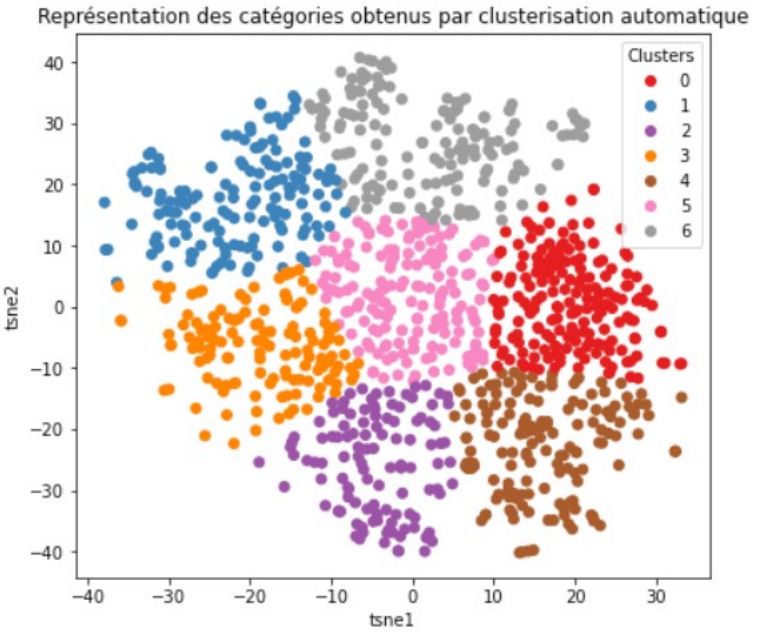
⇒ Grosse réduction du PCA !

6 . Clusters (K-means) à partir du t-SNE, calcul de l'ARI et graphique.

```
ARI :  0.0478 time :  10.0
```

C.1 SIFT





ARI : 0.0478 → Très mauvais !!!

Les clusters obtenus ne matchent pas avec les catégories déterminées manuellement par les vendeurs.

C'est normal→ SIFT se base dans l'identification de points qui déterminent la silhouette des objets, et on peut avoir des objets qui ont la même silhouette, ou une silhouette très proche qui sont de catégorie complètement différente.

C.2 CNN Transfer Learning

- **SIFT travaille sur des gradients et CNN sur des pixels** → plus précis, plus rapide et étendable à grande échelle.

- CNN → **convolutions** de l'image d'entrée **avec différents filtres** (ie. un kernel) pour obtenir un **'features map'** par filtre et les rassembler tous. **Chaque image est représentée par une matrice 3D** (ie. largeur, la hauteur et la profondeur - couleurs RVB-).

- 4 hyperparamètres importants sont:
  - -la taille du kernel;
  - -le nombre de filtres: combien de filtres voulons-nous utiliser;
  - -le stride: quelle est la taille des steps du filtre;
  - -le pandding.

- Processus long → Pour raccourcir on utilise des **'pre-trained models' comme VGG disponible sur Keras.**

C.2 CNN Transfer Learning

- Processus: 9 **étapes**

```
features_toto = []
for i in range(len(d1)):

    # 1. Chargement des images.
    image = load_img(d1['image_path'][i], target_size=(224, 224))

    # 2. Conversion des pixels de l'image en 'numpy array'.
    image = img_to_array(image)

    # 3. Remodélisation les données du modèle.
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))

    # 4. Préparation les images pour le modèle VGG.
    image = preprocess_input(image)

    # 5. Chargement du modèle.
    model = VGG16()

    # 6. Suppression de la couche de sortie.
    model = Model(inputs=model.inputs, outputs=model.layers[-2].output)

    # 7. Obtention des 'extracted features'.
    features= model.predict(image)

    features_toto.append(features)
    print(features.shape)

features_all = np.concatenate(np.asarray(features_toto), axis=0) #liste de descripteurs pour l'ensemble des images.
```

Image doit être dans le format 224x224 pixels

On veut juste les features pas la couche de classement déjà entraîné

C.2 CNN Transfer Learning

5. Réductions de dimension.

PCA (decompositon - SVD) puis t-SNE (pour que le t-SNE -réduction à 2D- soit plus rapide).

Note: On utilise'decomposition.PCA' (PCA via 'singular value decomposition (SVD)'), pour avoir des matrices diagonalisables→ plus simple à manipuler et plus rapide.

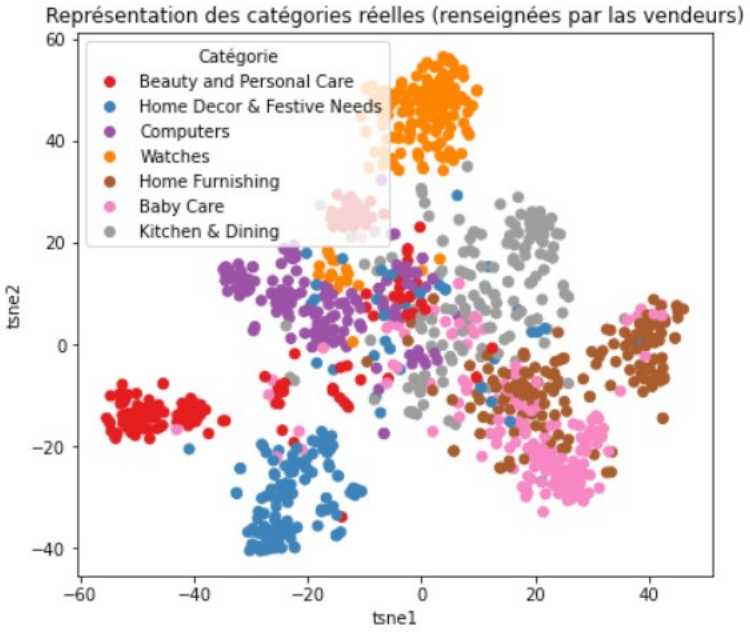
```
Dimensions dataset avant réduction PCA :  (1050, 4096)
Dimensions dataset après réduction PCA :  (1050, 499)
```

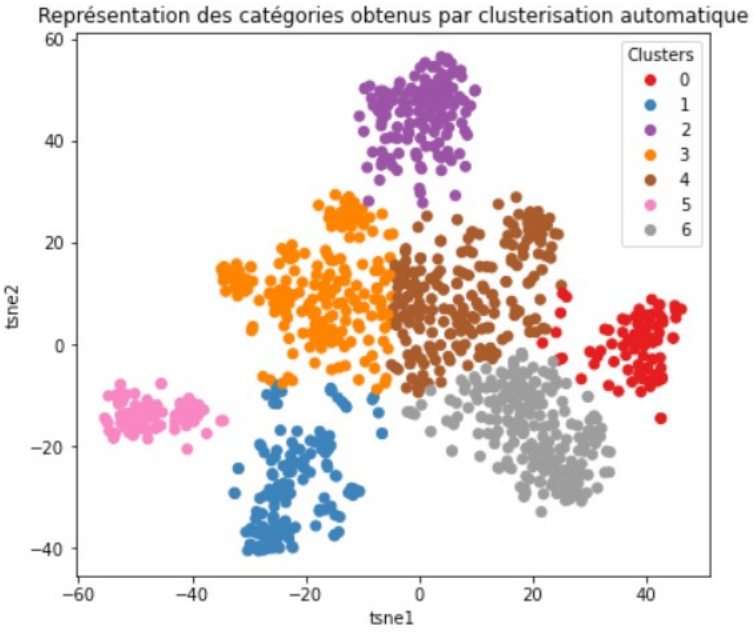
Grosse réduction du PCA !

6 . Clusters (K-means) à partir du t-SNE, calcul de l'ARI et graphique.

```
ARI :  0.4731 time :  10.0
```

C.2 CNN Transfer Learning





ARI : 0.4731 → Pas Mal !!!

Notez que les clusters obtenus par CNN matchent bien mieux que ceux obtenus pas SIFT !

CNN est en fait entraîné sur des millions d'images d'où son efficacité.

Résultats:

Le moteur de classification, qui basé sur une image et une description puisse attribuer leur catégorie aux articles est FAISABLE :

Extraction des ‘features’ texte → **USE** ('word/sentence embedding')
Uniquement sur **'product_name_bow_lem'**
(le moins de mots le mieux)

Extraction des ‘features’ image → **CNN**

MERCI