Normalisation et Vectorisation de Textes

Chapitre 5 : Techniques de prétraitement et représentation numérique des données textuelles

Introduction

Ce chapitre aborde les étapes essentielles pour transformer des données textuelles brutes en représentations structurées exploitables par les algorithmes d'analyse. La normalisation et la vectorisation constituent le socle méthodologique permettant de passer du langage naturel à des formats mathématiques adaptés au traitement informatique. Ces techniques sont fondamentales pour toute analyse quantitative de textes en sciences humaines et sociales.

1. Standardisation de textes

La standardisation constitue la première étape de prétraitement textuel, visant à homogénéiser le format des données et à réduire leur complexité superficielle.

1.1 Principes et objectifs de la standardisation

- **Définition** : Ensemble de transformations élémentaires appliquées au texte brut pour en faciliter le traitement ultérieur
- **Objectifs** :
    - Réduction de la variabilité non significative
    - Élimination des éléments non pertinents pour l'analyse
    - Préparation à la tokenisation
    - Réduction de la dimensionnalité initiale

1.2 Techniques de standardisation

1.2.1 Suppression des éléments non alphabétiques

- **Ponctuation** : virgules, points, points-virgules, etc.
- **Chiffres** : suppression ou remplacement par des tokens spéciaux
- **Caractères spéciaux** : symboles, émojis, caractères non-ASCII
- **Mise en œuvre pratique** :
- `import redef remove_special_chars(text):  # Supprimer la ponctuation et les caractères spéciaux  text = re.sub(r'[^\w\s]', '', text)  # Supprimer les chiffres  text = re.sub(r'\d+', '', text)  return text`

1.2.2 Normalisation de la casse

- **Conversion en minuscules** : réduction des variations dues à la capitalisation
- **Enjeux spécifiques** : conserver la casse pour les entités nommées dans certains cas
- **Mise en œuvre pratique** :
- `def lowercase_text(text):  return text.lower()`

1.2.3 Traitement des espaces et formatage

- **Suppression des espaces blancs superflus** : espaces multiples, tabulations, sauts de ligne
- **Normalisation des séparateurs** : homogénéisation des caractères d'espacement
- **Mise en œuvre pratique** :
- `def normalize_whitespace(text):  # Remplacer les espaces multiples par un seul espace  text = re.sub(r'\s+', ' ', text)  # Supprimer les espaces en début et fin de texte  text = text.strip()  return text`

1.3 Tokenisation

1.3.1 Principes de la tokenisation

- **Définition** : Processus de découpage d'un texte en unités élémentaires (tokens)
- **Types de tokens** :
  - Mots
  - Phrases
  - N-grammes (séquences de n mots ou caractères consécutifs)
  - Sous-mots (subword tokens)

1.3.2 Méthodes de tokenisation

- **Tokenisation par espaces** : simple mais limitée pour les langues sans séparateurs explicites
- **Tokenisation par règles** : utilisation d'expressions régulières et de règles linguistiques
- **Tokenisation statistique** : approches basées sur des modèles probabilistes

1.3.3 Outils et bibliothèques

- **NLTK** :
- `from nltk.tokenize import word_tokenize, sent_tokenize`
-
- `# Tokenisation en mots`
- `words = word_tokenize("Voici une phrase d'exemple à tokeniser.")`
-
- `# Tokenisation en phrases`
- `sentences = sent_tokenize("Première phrase. Deuxième phrase ! Et une troisième.")`
- **spaCy** :
- `import spacy`
-
- `nlp = spacy.load("fr_core_news_sm")`
- `doc = nlp("Voici une phrase d'exemple à tokeniser.")`
-
- `# Tokenisation en mots`
- `tokens = [token.text for token in doc]`

1.3.4 Défis spécifiques de la tokenisation

- **Traitement des contractions** : "l'arbre" → "l'" + "arbre"
- **Mots composés** : "porte-monnaie", "aujourd'hui"
- **Abréviations et acronymes** : "M.", "ONU", "etc."
- **Entités nommées multi-mots** : "New York", "Union européenne"

1.4 Applications pratiques de la standardisation

1.4.1 Préparation de corpus pour l'analyse quantitative

- **Exemple** : Standardisation d'un corpus d'articles de presse
- **Workflow complet** :
- `def standardize_text(text):  text = lowercase_text(text)  text = remove_special_chars(text)  text = normalize_whitespace(text)  tokens = word_tokenize(text)  return tokens# Application à un corpusstandardized_corpus = [standardize_text(document) for document in corpus]`

1.4.2 Évaluation des effets de la standardisation

- **Réduction du vocabulaire** : mesure de la diminution du nombre de types (tokens uniques)
- **Impact sur les analyses ultérieures** : tests comparatifs avec/sans standardisation
- **Visualisation des transformations** :
- `import pandas as pddf = pd.DataFrame({  'Original': [doc[:50] + '...' for doc in corpus[:5]],  'Standardisé': [' '.join(standardize_text(doc))[:50] + '...' for doc in corpus[:5]]})`

1.4.3 Exercice pratique : Standardisation d'un corpus multilingue

1. Importation d'un corpus de tweets ou d'articles en plusieurs langues
2. Adaptation des techniques de standardisation aux spécificités linguistiques
3. Comparaison des résultats selon les langues
4. Analyse de l'impact sur le vocabulaire et la distribution lexicale

2. Normalisation de textes

La normalisation va au-delà de la standardisation en apportant une dimension linguistique au prétraitement, visant à réduire la variabilité sémantique non pertinente.

2.1 Suppression des mots vides (stopwords)

2.1.1 Principe et définition

- **Mots vides** : Mots très fréquents apportant peu d'information sémantique (articles, prépositions, conjonctions, etc.)
- **Objectifs** :
  - Réduction de la dimensionnalité
  - Concentration sur les mots porteurs de sens
  - Amélioration de la performance des algorithmes d'analyse

2.1.2 Listes de stopwords

- **Listes prédéfinies** : disponibles dans les bibliothèques comme NLTK, spaCy
- **Listes adaptées au domaine** : construction de listes spécifiques au corpus étudié
- **Approches statistiques** : identification automatique basée sur la fréquence ou d'autres métriques

2.1.3 Mise en œuvre pratique

- **Avec NLTK** :
- `from nltk.corpus import stopwords`
-
- `# Chargement des stopwords français`
- `stop_words = set(stopwords.words('french'))`
-
- `# Filtrage des tokens`
- `filtered_tokens = [token for token in tokens if token.lower() not in stop_words]`
- **Avec spaCy** :
- `import spacy`
-
- `nlp = spacy.load("fr_core_news_sm")`
- `doc = nlp("Voici une phrase d'exemple à filtrer.")`
-
- `# Filtrage des stopwords`
- `filtered_tokens = [token.text for token in doc if not token.is_stop]`

2.1.4 Considérations méthodologiques

- **Impact sur l'analyse** : perte potentielle d'informations stylistiques ou contextuelles
- **Dépendance au domaine** : certains mots considérés comme vides peuvent être significatifs dans des contextes spécifiques
- **Adaptation selon le type d'analyse** : pertinence variable selon qu'on s'intéresse à la sémantique, au style, ou à la structure syntaxique

2.2 Stemming et lemmatisation

2.2.1 Principes et définitions

- **Stemming** : Réduction des mots à leur racine (stem) par troncature des suffixes et préfixes
- **Lemmatisation** : Réduction des mots à leur forme canonique (lemme) en tenant compte de la morphologie et de la syntaxe
- **Différences** :
    - Le stemming est plus rapide mais moins précis
    - La lemmatisation prend en compte le contexte linguistique et la partie du discours

2.2.2 Algorithmes de stemming

- **Porter Stemmer** : algorithme historique, basé sur des règles

- **Snowball (Porter2)** : version améliorée du Porter, disponible pour plusieurs langues
- **Lancaster Stemmer** : algorithme plus agressif, produisant des stems plus courts

2.2.3 Approches de lemmatisation

- **Dictionnaires** : utilisation de lexiques pour identifier les lemmes
- **Règles morphologiques** : application de règles linguistiques spécifiques à chaque langue
- **Approches statistiques** : modèles d'apprentissage automatique pour la lemmatisation

2.2.4 Mise en œuvre pratique

- **Stemming avec NLTK** :
- `from nltk.stem.snowball import FrenchStemmer`
-
- `stemmer = FrenchStemmer()`
- `stems = [stemmer.stem(token) for token in tokens]`
- **Lemmatisation avec spaCy** :
- `import spacy`
-
- `nlp = spacy.load("fr_core_news_sm")`
- `doc = nlp("Les chercheurs étudient les phénomènes sociaux contemporains.")`
-
- `lemmas = [token.lemma_ for token in doc]`

2.2.5 Comparaison et évaluation

- **Exemple comparatif** :
- `Mot original : "chercheurs"`
- `Stem (Snowball) : "cherch"`
- `Lemme : "chercheur"`
-
- `Mot original : "étudiaient"`
- `Stem (Snowball) : "étud"`
- `Lemme : "étudier"`
- **Métriques d'évaluation** :
    - Taux de réduction du vocabulaire
    - Précision sémantique (conservation du sens)
    - Rappel (capacité à regrouper toutes les variantes d'un même concept)

2.3 Détection et gestion des fautes d'orthographe

2.3.1 Importance en sciences sociales

- **Prévalence dans certains corpus** : médias sociaux, enquêtes ouvertes, transcriptions d'oral
- **Impact sur les analyses** : fragmentation du vocabulaire, perte d'information
- **Dimension sociolinguistique** : variations orthographiques comme marqueurs sociaux

2.3.2 Méthodes de détection

- **Vérification par dictionnaire** : comparaison avec des lexiques de référence
- **Distance d'édition** : mesure de la proximité avec des mots connus (Levenshtein, Jaro-Winkler)
- **Méthodes contextuelles** : utilisation du contexte pour identifier les erreurs probables
- **Méthodes statistiques** : modèles de langue prédictifs

2.3.3 Stratégies de correction

- **Correction automatique** : remplacement par la forme la plus probable
- **Suggestions multiples** : génération de plusieurs alternatives
- **Normalisation phonétique** : conversion en représentation phonétique pour gérer les erreurs phonétiquement plausibles
- **Apprentissage spécifique au domaine** : modèles adaptés au vocabulaire et aux erreurs typiques du corpus

2.3.4 Mise en œuvre pratique

- **Détection simple avec PySpellChecker** :
- `from spellchecker import SpellChecker`
-
- `spell = SpellChecker(language='fr')`
-
- `# Identifier les mots potentiellement mal orthographiés`
- `misspelled = spell.unknown(['recherche', 'annalyse', 'sociologue'])`
-
- `# Obtenir des corrections`
- `for word in misspelled:`
- `    print(f"Correction pour '{word}': {spell.correction(word)}")`
- `    print(f"Candidats: {spell.candidates(word)}")`
- **Correction contextuelle avec language-tool** :
- `import language_tool_python`
-
- `tool = language_tool_python.LanguageTool('fr')`
-
- `text = "Les chercheurs ont étudiés les différant aspects du phénomène."`
- `matches = tool.check(text)`
-
- `corrected = language_tool_python.utils.correct(text, matches)`
- `print(corrected)`

2.4 Applications de la normalisation textuelle

2.4.1 Analyse de contenu généré par les utilisateurs

- **Cas d'étude** : Normalisation de commentaires sur les réseaux sociaux
- **Défis spécifiques** : argot, abréviations, erreurs typiques
- **Workflow adapté** :
- `def normalize_user_content(text):    # Standardisation basique text = standardize_text(text)        # Correction orthographique adaptée    text = custom_spell_correction(text)        # Normalisation des expressions spécifiques    text =`

```
normalize_social_media_expressions(text)        # Lemmatisation
doc = nlp(text)    lemmas = [token.lemma_ for token in doc if not
token.is_stop]        return lemmas
```

2.4.2 Préparation pour l'analyse thématique

- **Objectif** : Optimiser la détection de thèmes et de sujets
- **Importance de la normalisation** : regroupement des variantes morphologiques d'un même concept
- **Évaluation comparative** : impact de différentes stratégies de normalisation sur la qualité des thèmes extraits

2.4.3 Exercice pratique : Normalisation d'un corpus d'entretiens

1. Importation d'un corpus de transcriptions d'entretiens
2. Application d'un pipeline complet de normalisation
3. Analyse comparative avant/après normalisation
4. Visualisation de l'impact sur la distribution du vocabulaire et les co-occurrences

3. Vectorisation de textes

La vectorisation transforme les textes normalisés en représentations numériques (vectorielles) exploitables par les algorithmes d'apprentissage automatique et d'analyse statistique.

3.1 Sac de mots (Bag of Words)

3.1.1 Principe et définition

- **Concept** : Représentation d'un texte comme un vecteur de fréquences de mots, sans considération de l'ordre
- **Matrice document-terme** : Matrice où chaque ligne représente un document, chaque colonne un terme, et chaque cellule la fréquence d'occurrence
- **Caractéristiques** :
    - Simplicité conceptuelle et computationnelle
    - Perte de l'information séquentielle et contextuelle
    - Haute dimensionnalité (vocabulaire complet)
    - Matrices généralement creuses (sparse)

3.1.2 Variantes du modèle

- **Matrice de présence/absence** : valeurs binaires (0/1) indiquant simplement la présence du terme
- **Matrice de fréquences brutes** : nombre d'occurrences de chaque terme
- **Matrice de fréquences normalisées** : ajustement des fréquences selon la longueur des documents

3.1.3 Mise en œuvre pratique

- **Avec scikit-learn** :

- ```
  from sklearn.feature_extraction.text import CountVectorizer# Corpus d'exemplecorpus = [    "Les sciences sociales étudient les phénomènes sociaux.",    "L'analyse des phénomènes sociaux requiert des méthodes mixtes.",    "Les méthodes qualitatives et quantitatives sont complémentaires."]# Vectorisation simplevectorizer = CountVectorizer()X = vectorizer.fit_transform(corpus)# Récupération du vocabulairefeature_names = vectorizer.get_feature_names_out()# Affichage de la matriceprint(X.toarray())print(feature_names)
  ```

3.1.4 Applications et limites

- **Applications** :
  - Classification de documents
  - Recherche d'information simple
  - Analyses préliminaires de corpus
- **Limites** :
  - Ne capture pas la sémantique
  - Ne tient pas compte de l'ordre des mots
  - Sensibilité au vocabulaire spécifique
  - Dimensionnalité potentiellement très élevée

3.2 TF-IDF (Term Frequency-Inverse Document Frequency)

3.2.1 Principe et définition

- **TF (Term Frequency)** : Mesure de la fréquence d'un terme dans un document
- **IDF (Inverse Document Frequency)** : Mesure de la rareté d'un terme dans le corpus entier
- **TF-IDF** : Produit de TF et IDF, pondérant l'importance d'un terme proportionnellement à sa fréquence dans le document et inversement proportionnellement à sa fréquence dans le corpus

3.2.2 Formules et calcul

- **Term Frequency (TF)** : $TF(t,d) = \frac{\text{nombre d'occurrences de }t\text{ dans }d}{\text{nombre total de termes dans }d}$
- **Inverse Document Frequency (IDF)** : $IDF(t) = \log\left(\frac{\text{nombre total de documents}}{\text{nombre de documents contenant }t}\right)$
- **TF-IDF** : $TF\text{-}IDF(t,d) = TF(t,d) \times IDF(t)$

3.2.3 Variantes et paramètres

- **Lissage de l'IDF** : Addition d'un terme constant pour éviter les divisions par zéro
- **Normalisation** : Diverses méthodes pour normaliser les vecteurs (L1, L2, etc.)
- **Sous-linéarité de TF** : Utilisation du logarithme pour atténuer l'impact des termes très fréquents

3.2.4 Mise en œuvre pratique

- **Avec scikit-learn** :
  ```

- `from sklearn.feature_extraction.text import TfidfVectorizer# Corpus d'exemplecorpus = ["Les sciences sociales étudient les phénomènes sociaux.", "L'analyse des phénomènes sociaux requiert des méthodes mixtes.", "Les méthodes qualitatives et quantitatives sont complémentaires."]# Vectorisation TF-IDFvectorizer = TfidfVectorizer()X = vectorizer.fit_transform(corpus)# Récupération du vocabulairefeature_names = vectorizer.get_feature_names_out()# Affichage des scores TF-IDFprint(X.toarray())`

3.2.5 Applications et avantages

- **Extraction de mots-clés** : Identification des termes les plus distinctifs de chaque document
- **Recherche d'information** : Amélioration des performances par rapport au simple comptage
- **Classification de textes** : Pondération plus pertinente pour les algorithmes d'apprentissage
- **Clustering de documents** : Regroupement basé sur les similarités thématiques

3.3 Word Embeddings (plongements lexicaux)

3.3.1 Principe et définition

- **Concept** : Représentation des mots dans un espace vectoriel continu de faible dimension où les relations sémantiques sont préservées
- **Caractéristiques** :
  - Capture des relations sémantiques et syntaxiques
  - Représentation dense (par opposition aux représentations creuses comme Bag of Words)
  - Conservation des propriétés analogiques (roi - homme + femme ≈ reine)
  - Apprentissage à partir des contextes d'utilisation

3.3.2 Word2Vec

- **Algorithmes** :
  - CBOW (Continuous Bag of Words) : prédiction d'un mot à partir de son contexte
  - Skip-gram : prédiction du contexte à partir d'un mot
- **Hyperparamètres importants** :
  - Taille de la fenêtre contextuelle
  - Dimension des vecteurs
  - Méthodes d'apprentissage (hierarchical softmax, negative sampling)
- **Mise en œuvre** :
- `from gensim.models import Word2Vec# Corpus tokenisésentences = [['les', 'sciences', 'sociales'], ['étudient', 'les', 'phénomènes'], ...]# Entraînement du modèlemodel = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)# Accès aux vecteursvector = model.wv['sciences']# Mots similairessimilar_words = model.wv.most_similar('sciences', topn=5)`

3.3.3 GloVe (Global Vectors)

- **Principe** : Combinaison de factorisation de matrices et d'apprentissage local du contexte
- **Spécificités** :
    - Utilisation des statistiques globales de co-occurrence
    - Préservation de la similarité cosinus entre vecteurs pour refléter la relation sémantique
- **Utilisation de modèles pré-entraînés** :
- ```
  import numpy as npfrom gensim.models import KeyedVectors# Chargement de vecteurs GloVe pré-entraînésglove_file = 'glove.6B.100d.txt'model = KeyedVectors.load_word2vec_format(glove_file, binary=False, no_header=True)# Utilisation des vecteurssimilarity = model.similarity('femme', 'homme')
  ```

3.3.4 Modèles contextuels et limitations des embeddings statiques

- **Limitation des embeddings statiques** : un seul vecteur par mot, indépendamment du contexte
- **Évolution vers les modèles contextuels** : BERT, ELMo, etc.
- **Embeddings contextuels** : génération de représentations différentes selon le contexte d'utilisation

3.3.5 Applications en sciences sociales

- **Analyse de discours** : Identification de champs sémantiques et de cadrages
- **Étude des biais** : Détection de biais sociaux dans les représentations linguistiques
- **Cartographie conceptuelle** : Visualisation des relations entre concepts dans un domaine
- **Analyse diachronique** : Étude de l'évolution des significations à travers le temps

3.4 Applications pratiques de la vectorisation

3.4.1 Classification de documents

- **Workflow complet** :
- ```
  from sklearn.feature_extraction.text import TfidfVectorizerfrom sklearn.naive_bayes import MultinomialNBfrom sklearn.pipeline import Pipeline# Pipeline de traitementpipeline = Pipeline([ ('vectorizer', TfidfVectorizer(max_features=1000)),    ('classifier', MultinomialNB())])# Entraînementpipeline.fit(X_train, y_train)# Prédictionpredictions = pipeline.predict(X_test)
  ```

3.4.2 Analyse thématique

- **Modélisation de sujets avec LDA** :
- ```
  from sklearn.decomposition import LatentDirichletAllocationfrom sklearn.feature_extraction.text import CountVectorizer# Vectorisationvectorizer = CountVectorizer(max_features=1000)X = vectorizer.fit_transform(corpus)# Modèle LDAlda = LatentDirichletAllocation(n_components=5, random_state=42)lda.fit(X)# Affichage des thèmesfeature_names = vectorizer.get_feature_names_out()for topic_idx, topic in enumerate(lda.components_):    print(f"Thème #{topic_idx + 1}:")
  ```
  ```

```
print(" ".join([feature_names[i] for i in topic.argsort()[:-10 - 1:-1]]))
```

3.4.3 Recherche sémantique

- **Utilisation d'embeddings pour la recherche** :
- ```
  def semantic_search(query, documents, model):    # Vectorisation de la requête    query_vec = model.wv[query.split()]    query_vec = np.mean(query_vec, axis=0)        # Vectorisation des documents    doc_vecs = []    for doc in documents:        doc_words = [w for w in doc.split() if w in model.wv]        if doc_words:            doc_vec = np.mean([model.wv[w] for w in doc_words], axis=0)            doc_vecs.append(doc_vec)        else:            doc_vecs.append(np.zeros(model.vector_size))        # Calcul des similarités    similarities = cosine_similarity([query_vec], doc_vecs)[0]        # Tri des résultats    results = [(documents[i], similarities[i]) for i in range(len(documents))]    results.sort(key=lambda x: x[1], reverse=True)        return results
  ```

3.4.4 Visualisation des représentations vectorielles

- **Réduction de dimensionnalité avec t-SNE** :
- ```
  from sklearn.manifold import TSNEimport matplotlib.pyplot as plt# Sélection des mots à visualiserwords = ["société", "culture", "politique", "économie", "science", ...]word_vectors = [model.wv[word] for word in words]# Réduction de dimensionnalitétsne = TSNE(n_components=2, random_state=42)reduced_vectors = tsne.fit_transform(word_vectors)# Visualisationplt.figure(figsize=(12, 8))for i, word in enumerate(words):    plt.scatter(reduced_vectors[i, 0], reduced_vectors[i, 1])    plt.annotate(word, (reduced_vectors[i, 0], reduced_vectors[i, 1]))plt.show()
  ```

4. Travaux pratiques intégrateurs

4.1 Projet 1 : Analyse comparative des discours politiques

1. **Objectif** : Comparer les discours de différentes personnalités politiques
2. **Étapes** :
    - Constitution d'un corpus de discours
    - Normalisation complète (standardisation, suppression des stopwords, lemmatisation)
    - Vectorisation avec TF-IDF
    - Analyse des spécificités lexicales
    - Visualisation des proximités thématiques

4.2 Projet 2 : Construction d'un système de recommandation d'articles

1. **Objectif** : Développer un système recommandant des articles similaires
2. **Étapes** :
    - Préparation d'un corpus d'articles académiques
    - Normalisation adaptée au vocabulaire spécialisé
    - Génération d'embeddings de documents

    - Calcul de similarités entre articles
    - Évaluation de la pertinence des recommandations

4.3 Projet 3 : Analyse de sentiment sur des commentaires d'utilisateurs

1. **Objectif** : Développer un classifieur de sentiment pour des avis en ligne
2. **Étapes** :
    - Collecte de commentaires (ex: avis sur des produits)
    - Normalisation adaptée au langage informel
    - Comparaison de différentes techniques de vectorisation
    - Entraînement de modèles de classification
    - Analyse des erreurs et améliorations

Conclusion

La normalisation et la vectorisation de textes constituent des étapes fondamentales pour transformer les données textuelles brutes en représentations structurées exploitables par les méthodes d'analyse quantitative. Ces techniques permettent de faire le pont entre l'approche qualitative traditionnelle des sciences humaines et sociales et les méthodes computationnelles modernes.

L'évolution rapide des technologies de traitement du langage naturel, notamment avec l'avènement des modèles de langue transformers, ouvre de nouvelles perspectives pour l'analyse textuelle en sciences sociales. Cependant, la maîtrise des techniques fondamentales présentées dans ce chapitre reste essentielle pour comprendre les principes sous-jacents et pour adapter les méthodes aux spécificités des corpus et des questions de recherche.

Bibliographie et ressources

Ouvrages et articles de référence

- Jurafsky, D., & Martin, J. H. (2020). *Speech and Language Processing* (3rd ed. draft). https://web.stanford.edu/~jurafsky/slp3/
- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- Mikolov, T., Chen, K.,