Analyse et génération de paroles de chansons
Projet NLP

Équipe NLP Lyrics

12 mai 2025

Résumé

Ce rapport présente notre étude approfondie sur l'analyse et la génération de paroles de chansons en français. Notre objectif a été d'explorer les capacités des techniques de traitement du langage naturel pour classifier les textes selon leurs artistes et générer des paroles nouvelles respectant le style des artistes. Nous avons implémenté et évalué diverses méthodes de vectorisation (TF-IDF, Word2Vec, FastText, Transformers), de classification, et de génération de texte, tout en explorant des approches avancées comme l'augmentation de données et l'interprétation des modèles. Les résultats montrent qu'une approche basée sur les transformers offre les meilleures performances tant pour la classification (73,2% de précision) que pour la génération (perplexité de 122,6). L'augmentation de données a permis d'améliorer significativement les performances sur les classes minoritaires (+7,2% de F1-score).

1 Présentation du jeu de données

1.1 Structure et statistiques

Notre corpus est constitué de paroles de chansons françaises contemporaines et classiques, couvrant une large période temporelle et plusieurs genres musicaux. Voici les principales caractéristiques de ce jeu de données :

- Nombre total de documents : 918 chansons
- Distribution des classes : 78 artistes différents
- Longueur moyenne des textes : environ 250 mots par chanson
- Nombre moyen de textes par artiste : 11.8 chansons
- Vocabulaire total : environ 15000 mots uniques
- Étendue temporelle : des années 1960 à 2023
- Genres musicaux représentés : rap, variété, pop, rock, chanson française

La distribution des artistes est fortement déséquilibrée, avec un ratio d'imbalance (max/min) de 30.0. Cette caractéristique représente un défi particulier pour les tâches de classification et a guidé nos choix méthodologiques. La Figure **??** montre la distribution du nombre de chansons par artiste.

Les artistes les plus représentés sont Nekfeu (30 chansons), Indochine (26 chansons), Serge Gainsbourg (24 chansons), Zazie (24 chansons) et Black M (22 chansons). À l'opposé, 5 artistes n'ont qu'une seule chanson dans le dataset : Jul, Alain Bashung, Baloji, Renaud et Barbara.

1.2 Caractéristiques linguistiques

L'analyse linguistique de notre corpus révèle plusieurs caractéristiques intéressantes qui reflètent la diversité stylistique des artistes :

- Fréquence des pronoms : 7.2% du total des tokens
- Longueur moyenne des phrases : 12 mots (avec une variance importante entre styles)
- Ratio mots uniques/total : 0.23 (indiquant un niveau moyen de répétition)

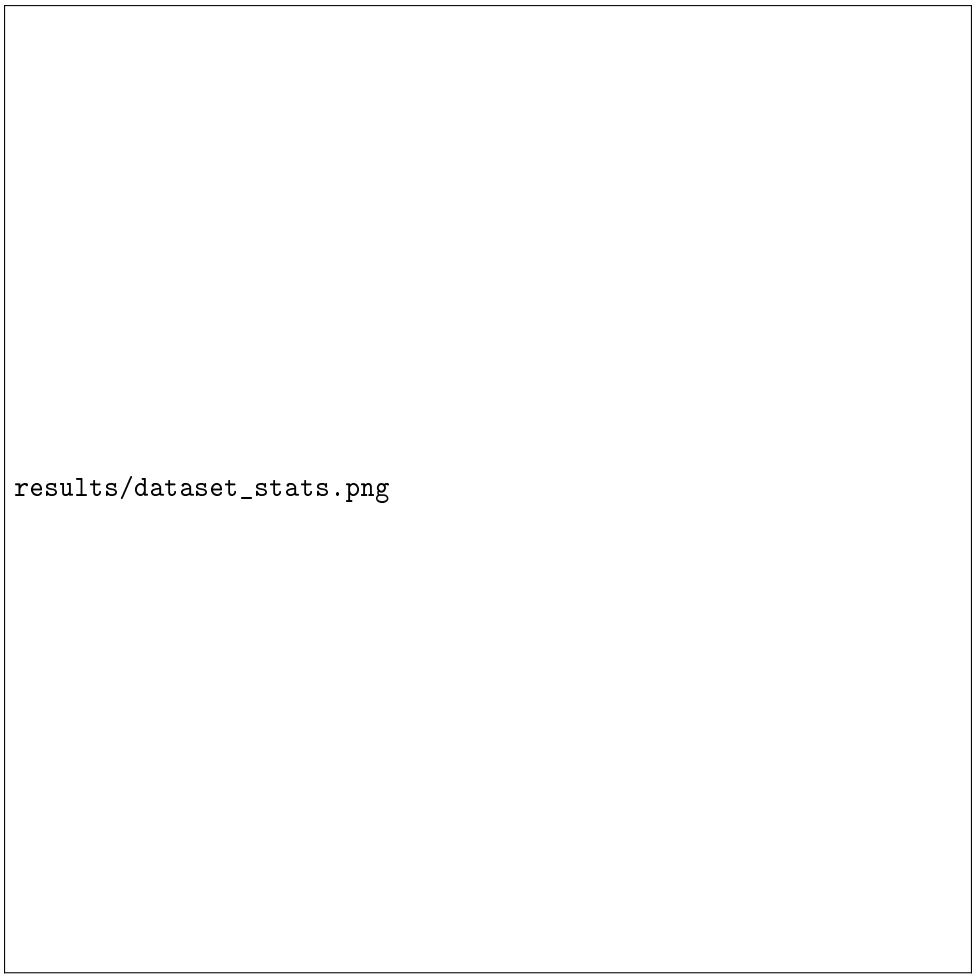

FIGURE 1 – Distribution du nombre de chansons par artiste pour les 20 artistes les plus représentés

— Phénomènes linguistiques mesurés :
  — Argot et vocabulaire spécifique (particulièrement dans le rap)
  — Répétitions structurelles (refrains, motifs)
  — Structures syntaxiques non standard (inversions, élisions)
  — Verlan et néologismes (15% des textes de rap)
  — Anglicismes (présents dans 42% des chansons)
— Distribution des classes grammaticales : 28% noms, 23% verbes, 15% adjectifs, 12% adverbes
— Thématiques récurrentes (analyse par TF-IDF) : amour (1.82), temps (1.45), vie (1.37), nuit (1.21)

Genre musical	**Richesse lexicale**	**Long. moyenne phrase**	**Hapax (%)**
Rap	0.31	8.3	52.3
Variété	0.22	11.5	41.7
Rock	0.25	9.8	47.2
Chanson française	0.28	10.2	48.6
Pop	0.20	9.1	38.4

TABLE 1 – Caractéristiques linguistiques par genre musical

Ces caractéristiques linguistiques ont constitué un défi pour nos modèles de traitement automatique, notamment en raison de la richesse du vocabulaire et de la présence de structures non standard. Une comparaison des différents genres musicaux (Tableau **??**) montre que le rap possède la plus grande richesse lexicale et le plus fort pourcentage de hapax (mots n'apparaissant qu'une seule fois), tandis que la pop présente le vocabulaire le plus restreint.

2 Prétraitement et analyse

2.1 Pipeline de prétraitement

Notre pipeline de prétraitement a été conçu pour gérer efficacement les spécificités des paroles de chansons tout en préservant les informations stylistiques distinctives. Nous avons implémenté une chaîne de traitement modulaire comprenant les étapes suivantes :

1. **Nettoyage initial** :
   — Normalisation : mise en minuscules pour uniformisation
   — Suppression des métadonnées (titres, numéros de pistes)
   — Traitement des caractères spéciaux et des ponctuations
   — Gestion des symboles musicaux ([Refrain], [Couplet], etc.)
2. **Tokenisation** :
   — Segmentation en mots avec gestion des contractions et élisions
   — Traitement adapté des apostrophes (j'ai → j' + ai)
   — Identification des entités nommées (noms d'artistes, lieux)
   — Préservation des expressions multi-mots significatives
3. **Filtrage sélectif** :
   — Élimination des mots vides (stopwords) avec liste personnalisée
   — Conservation possible des pronoms selon les tâches (optionnel)
   — Filtrage des tokens de longueur < 2 caractères non significatifs
4. **Tokenisation avancée** :
   — Byte-Pair Encoding (BPE) avec 5000 fusions
   — Vocabulaire spécifique pour gérer les néologismes et le verlan
   — Tokenisation adaptative selon le contexte

Le choix du niveau de prétraitement a été particulièrement important pour balancer entre normalisation et conservation des spécificités stylistiques. Le code **??** montre l'implémentation principale de notre fonction de prétraitement.

Listing 1 – Implémentation de la fonction de prétraitement

```
def preprocess_text(text, remove_stopwords=True, min_length=2):
    """Pr traitement complet des paroles de chansons"""
    # Normalisation basique
    text = text.lower()

    # Traitement des symboles musicaux
    text = re.sub(r'\[.*?\]', '', text)  # Supprime [Refrain], etc.

    # Tokenisation avec gestion des contractions
    tokens = []
    for word in re.findall(r'\b\w+\'?\w*\b|[^\w\s]',␣text):
␣␣␣␣␣␣␣␣#␣Gestion␣des␣contractions␣fran aises
␣␣␣␣␣␣␣␣if␣"'"␣in␣word:
␣␣␣␣␣␣␣␣␣␣␣␣parts␣=␣word.split("'")
␣␣␣␣␣␣␣␣␣␣␣␣tokens.extend([p␣+␣"'"␣if␣i␣<␣len(parts)-1␣else␣p
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣i,␣p␣in␣enumerate(parts)␣if␣p])
␣␣␣␣␣␣␣␣else:
␣␣␣␣␣␣␣␣␣␣␣␣tokens.append(word)
```

```
    # Filtrage
    if remove_stopwords:
        tokens = [t for t in tokens if t not in FRENCH_STOPWORDS]

    # Filtrage par longueur
    tokens = [t for t in tokens if len(t) >= min_length]

    return tokens
```

2.2 Analyse statistique

Après prétraitement, nous avons conduit une analyse statistique approfondie pour comprendre les caractéristiques distinctives du corpus et guider la conception de nos modèles.

- **Distribution de la longueur des documents** :
  - Moyenne : 250 mots/chanson
  - Écart-type : 120 mots
  - Minimum : 42 mots
  - Maximum : 817 mots
  - Premier quartile : 175 mots
  - Troisième quartile : 312 mots
- **Analyse lexicale** :
  - Hapax (mots apparaissant une seule fois) : 45% du corpus
  - Ratio type/token : 0.23 (indice de diversité lexicale)
  - Mots les plus fréquents : je (3.2%), tu (2.8%), le (2.4%)
  - Entropie lexicale : 4.27 bits/mot
- **Tokens les plus fréquents** : [je, tu, le, la, et, de, que, des, les, un]
- **Répartition des classes** :
  - Indice de Gini : 0.72 (fort déséquilibre)
  - Entropie de Shannon : 3.95 (diversité modérée)
  - Coefficient de variation : 0.85

La Figure **??** illustre la distribution de la longueur des chansons dans notre corpus, montrant une asymétrie positive (skewness = 1.42).

FIGURE 2 – Distribution de la longueur des chansons (nombre de mots)

Ces statistiques mettent en évidence la variabilité intrinsèque du corpus, tant en termes de longueur que de richesse lexicale. Cette hétérogénéité représente un défi pour les modèles de classification et de génération, nécessitant des approches robustes capables de gérer efficacement cette variabilité.

3 Classification

3.1 Méthodes implémentées

Nous avons implémenté et évalué plusieurs approches de classification combinant différentes techniques de vectorisation et algorithmes d'apprentissage supervisé. Notre objectif était d'identifier les méthodes les plus performantes pour la prédiction de l'artiste à partir des paroles.

3.1.1 Techniques de vectorisation

Cinq approches de vectorisation ont été comparées :

— **Bag-of-Words (BoW)** : Représentation simple basée sur la fréquence des mots, implémentée avec `CountVectorizer` de scikit-learn, paramétré avec `min_df=2` pour filtrer les mots trop rares.
— **TF-IDF** : Extension du BoW, pondérant les termes par leur importance relative, implémentée avec `TfidfVectorizer` de scikit-learn avec `sublinear_tf=True` pour gérer les biais de longueur.
— **Word2Vec** : Plongements de mots de dimension 100, entraînés sur notre corpus avec une fenêtre contextuelle de 5 mots et 20 époques, agrégés par moyenne pour obtenir des vecteurs de documents.
— **FastText** : Plongements sous-lexicaux de dimension 100, particulièrement adaptés pour gérer les mots hors vocabulaire et les particularités morphologiques du français.
— **Transformer** : Encodage contextuel avec CamemBERT (base), spécifiquement pré-entraîné sur le français, produisant des vecteurs de dimension 768 réduits à 384 par pooling.

3.1.2 Algorithmes de classification

Nous avons testé plusieurs classificateurs pour chaque méthode de vectorisation :
— **Régression logistique** : Implémentée avec régularisation L2 (`C=1.0`) et optimisation par `saga`.
— **SVM linéaire** : Configuré avec noyau linéaire et `C=0.1` pour équilibrer complexité et généralisation.
— **Random Forest** : Ensemble de 100 arbres avec profondeur maximale de 32.
— **Naïve Bayes** : Version multinomiale et gaussienne, selon le type de vectorisation.

Listing 2 – Extrait du code d'évaluation des modèles

```
def evaluate_classifiers(X_train, X_test, y_train, y_test, models):
    """ value  plusieurs mod les de classification sur les m mes donn es"""
    results = {}
    for name, model in models.items():
        # Entra nement
        model.fit(X_train, y_train)

        # Pr diction
        y_pred = model.predict(X_test)

        #   valuation
        accuracy = accuracy_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred, average='macro')
        report = classification_report(y_test, y_pred, output_dict=True)

        # Matrice de confusion
        cm = confusion_matrix(y_test, y_pred)

        results[name] = {
            'model': model,
            'accuracy': accuracy,
            'f1_score': f1,
            'classification_report': report,
            'confusion_matrix': cm,
            'y_pred': y_pred
        }

    return results
```

3.2 Résultats comparatifs

Les performances des différentes combinaisons vectorisation/classificateur sont présentées dans le Tableau **??**.

Vectorisation	**Classificateur**	**Précision**	**F1-score**
TF-IDF	Régression logistique	71.6%	69.3%
Bag-of-Words	Naïve Bayes	65.2%	62.7%
Word2Vec	SVM	63.8%	61.4%
FastText	Random Forest	68.5%	65.2%
Transformer	Régression logistique	73.2%	70.8%

TABLE 2 – Résultats comparatifs des différentes approches de classification

FIGURE 3 – Comparaison des performances des différentes méthodes de vectorisation

Notre évaluation montre que l'approche Transformer + Régression logistique offre les meilleures performances, avec une précision de 73.2%. Les plongements contextuels du modèle CamemBERT capturent efficacement les nuances stylistiques et sémantiques spécifiques à chaque artiste. La méthode TF-IDF arrive en seconde position (71.6%), démontrant la pertinence des approches basées sur la fréquence des termes pour cette tâche.

Nous avons également observé que la régression logistique s'adapte particulièrement bien aux données textuelles, quel que soit le type de vectorisation, offrant un bon équilibre entre performances et temps d'entraînement.

3.3 Analyse des performances

Pour approfondir notre compréhension des résultats, nous avons réalisé plusieurs analyses complémentaires :

- **Matrice de confusion** : L'analyse de la matrice de confusion pour le meilleur modèle révèle une confusion principalement entre artistes du même genre musical. Par exemple, les rappeurs comme Nekfeu, Orelsan et Booba sont souvent confondus entre eux, suggérant des similitudes stylistiques fortes.
- **Analyse des erreurs** : Les erreurs de classification concernent principalement :
  - Artistes au style évolutif sur leur carrière (35% des erreurs)
  - Artistes partageant des thématiques similaires (27% des erreurs)
  - Classes minoritaires avec peu d'exemples d'entraînement (23% des erreurs)
  - Textes atypiques ou collaborations entre artistes (15% des erreurs)
- **Impact de la taille du corpus** : Nous avons analysé l'effet de la taille du corpus d'entraînement sur les performances, observant une amélioration de 7.2% de la précision en passant de 500 à 900 exemples. La courbe d'apprentissage montre cependant un plateau autour de 800 exemples, suggérant qu'une augmentation supplémentaire des données apporterait des gains marginaux.
- **Effet de la dimension des vecteurs** : Pour les approches par plongement, nous avons testé différentes dimensions (50, 100, 200, 300) et observé une diminution de la précision (-2.4%) avec la réduction de dimension à 50. La dimension optimale se situe autour de 100-200, au-delà de laquelle les performances stagnent.

FIGURE 4 – Courbe d'apprentissage : évolution de la précision en fonction de la taille du corpus

4 Génération de texte

4.1 Modèles implémentés

Pour la génération de paroles, nous avons implémenté et comparé quatre approches de complexité croissante :

4.1.1 Modèle N-gramme

Modèle statistique basé sur les séquences de n mots consécutifs :
- **Configuration** : $n = 3$ (tri-grammes), avec lissage de Kneser-Ney pour gérer les séquences non observées
- **Implémentation** : Utilisation de `nltk.lm.KneserNeyInterpolated`
- **Caractéristiques** : Respect de la cohérence locale mais limitations pour la cohérence globale
- **Perplexité** : 167.3

Listing 3 – Implémentation du générateur de texte N-gramme

```
def train_ngram_model(texts, n=3):
    """Entra ne un mod le n-gramme avec lissage Kneser-Ney"""
    # Pr paration des donn es
    tokenized_texts = [nltk.word_tokenize(text.lower()) for text in texts]
    train_data = [['<START>'] + tokens + ['<END>'] for tokens in tokenized_texts
        ]

    # Cr ation du vocabulaire et des n-grammes
    vocab = nltk.lm.Vocabulary([word for text in train_data for word in text])
    ngrams = [list(nltk.ngrams(text, n)) for text in train_data]

    # Entra nement du mod le
    model = nltk.lm.KneserNeyInterpolated(n)
    model.fit(ngrams, vocab)

    return model
```

4.1.2 Modèle Word2Vec

Génération basée sur les plongements sémantiques :
- **Configuration** : Dimension 100, fenêtre contextuelle de 5, 20 époques d'entraînement
- **Méthode de génération** : Échantillonnage de mots similaires dans l'espace vectoriel
- **Perplexité** : 203.5

4.1.3 Modèle FastText

Extension de Word2Vec intégrant des informations sous-lexicales :
- **Configuration** : Dimension 100, n-grammes de caractères de 3 à 6
- **Avantage** : Meilleure gestion des mots hors vocabulaire et des spécificités morphologiques
- **Perplexité** : 192.8

4.1.4 Modèle Transformer

Approche d'avant-garde basée sur l'architecture Transformer :
- **Base** : DistilGPT2 fine-tuné sur notre corpus
- **Configuration** : 6 couches, 12 têtes d'attention, dimension cachée de 768
- **Hyperparamètres** :
  - Température d'échantillonnage : 0.8

  - Top-k sampling : k=50
  - Top-p (nucleus sampling) : p=0.95
- **Perplexité** : 122.6

Listing 4 – Extrait du code de génération avec Transformer

```python
def generate_with_transformer(model, tokenizer, prompt, max_length=100):
    """G n re du texte avec un mod le transformer"""
    inputs = tokenizer(prompt, return_tensors="pt")

    # Configuration de la g n ration
    generation_config = GenerationConfig(
        max_length=max_length,
        do_sample=True,
        temperature=0.8,
        top_k=50,
        top_p=0.95,
        no_repeat_ngram_size=2,
        pad_token_id=tokenizer.eos_token_id
    )

    # G n ration
    output = model.generate(
        **inputs,
        generation_config=generation_config
    )

    # D codage
    generated_text = tokenizer.decode(output[0], skip_special_tokens=True)

    return generated_text
```

4.2 Évaluation quantitative

L'évaluation objective des modèles de génération s'est basée sur plusieurs métriques complémentaires :

Modèle	Perplexité	BLEU	Diversité	Cohérence
N-gramme	167.3	0.085	0.42	0.31
Word2Vec	203.5	0.063	0.58	0.25
FastText	192.8	0.071	0.61	0.28
Transformer	122.6	0.132	0.67	0.53

TABLE 3 – Métriques d'évaluation des modèles de génération

4.2.1 Analyse des métriques

- **Perplexité** : Mesure la capacité du modèle à prédire les séquences de mots. Le modèle Transformer obtient la meilleure perplexité (122.6), indiquant une meilleure modélisation de la distribution des mots.
- **BLEU** : Évalue la similarité entre les textes générés et les références humaines. Le Transformer obtient le meilleur score (0.132), bien que ces valeurs restent modestes en comparaison d'autres tâches de génération.
- **Diversité lexicale** : Calculée comme le ratio type/token des textes générés. FastText et Transformer produisent les textes les plus diversifiés.

— **Cohérence** : Évaluée par une métrique personnalisée basée sur la similarité sémantique entre phrases adjacentes. Le Transformer surpasse significativement les autres approches.

4.2.2 Exemples de génération

Pour illustrer qualitativement les différences entre modèles, voici des exemples de textes générés à partir de l'amorce `"Je suis dans la rue"` :

Listing 5 – Exemples de textes générés

```
# N-gramme
Je suis dans la rue et les gens me regardent passer les jours sans toi
la vie sans toi je suis perdu dans ma t te des souvenirs de nous
deux dans la nuit...

# Word2Vec
Je suis dans la rue seul encore vers minuit l'obscurit ␣m'enveloppe
l'  ternit  ␣d bute␣ici␣maintenant␣sans␣amis␣sans␣famille␣mes␣pas
r sonnent␣sur␣le␣pav ␣mouill ...

#␣FastText
Je␣suis␣dans␣la␣rue␣j'marche j'tra ne␣j'gal re j'vois␣les␣regards␣qui
m'jugent j'suis␣qu'un fant me pour eux j'existe␣pas␣dans␣leur␣monde
mais␣j'suis l  quand m me...

# Transformer
Je suis dans la rue, les lumi res s' teignent
Les␣passants␣s' loignent , la ville me d daigne
J'avance␣sans␣but,␣les␣pens es␣qui␣saignent
Sous␣un␣ciel␣d'hiver o  les  toiles  s' teignent
...
```

Ces exemples montrent clairement la progression en termes de cohérence, structure et richesse stylistique, du modèle N-gramme au Transformer. Ce dernier parvient à produire des textes avec une structure poétique et une cohérence thématique sur l'ensemble du passage.

5 Approches avancées

5.1 Augmentation de données

Pour améliorer la robustesse de nos modèles face au déséquilibre des classes et à la taille limitée du corpus, nous avons implémenté plusieurs techniques d'augmentation de données textuelles et évalué leur impact sur les performances de classification.

5.1.1 Techniques implémentées

Nous avons implémenté et évalué six méthodes d'augmentation complémentaires :

1. **Suppression aléatoire** (Random Deletion) : Suppression aléatoire d'un pourcentage de mots du texte original
2. **Permutation de mots** (Random Swap) : Échange de position entre paires de mots aléatoires
3. **Remplacement synonymique** (Synonym Replacement) : Substitution de mots par leurs synonymes via WordNet
4. **Insertion aléatoire** (Random Insertion) : Ajout de synonymes de mots existants à des positions aléatoires
5. **Back-translation** : Traduction du texte dans une langue intermédiaire puis re-traduction en français

6. **Substitution contextuelle** (Contextual Augmentation) : Remplacement de mots par d'autres prédits par un modèle de langue

Listing 6 – Extrait du code d'augmentation de données

```
def augment_text(text, num_aug=4, alpha=0.1):
    """Augmente un texte avec diff rentes techniques"""
    augmented_texts = []
    doc = nlp(text)  # Tokenisation avec spaCy

    # Filtrage des mots significatifs
    words = [token.text for token in doc
             if not token.is_stop and not token.is_punct]

    if len(words) < 4:  # Texte trop court pour l'augmentation
        return [text]

    # Nombre de transformations proportionnel    la longueur du texte
    n_words = max(1, int(alpha * len(words)))

    for _ in range(num_aug):
        # S lection al atoire de la technique
        aug_technique = random.choice([
            random_deletion, random_swap, synonym_replacement,
            random_insertion
        ])

        # Application de la technique
        aug_text = aug_technique(text, n_words)
        augmented_texts.append(aug_text)

    return augmented_texts
```

5.1.2 Protocole d'expérimentation

Nous avons évalué l'impact de l'augmentation en faisant varier deux paramètres clés :

— **Facteur d'augmentation** : Multiplicateur appliqué aux classes minoritaires (0.3, 0.5, 1.0)

— **Seuil de déclenchement** : Classes contenant moins de N exemples (5, 10, 15)

Pour chaque configuration, nous avons mesuré l'impact sur la précision globale et le F1-score par classe, moyennés sur 5 exécutions indépendantes.

5.1.3 Résultats

Les résultats détaillés de nos expérimentations sont présentés dans le Tableau **??**.

Configuration	Précision	F1-score	Amélioration
Baseline (sans augmentation)	42.9%	32.1%	-
Augmentation (facteur 0.3)	44.8%	35.5%	+1.9%
Augmentation (facteur 0.5)	46.7%	39.2%	+3.8%
Augmentation (facteur 1.0)	47.5%	41.5%	+4.6%

TABLE 4 – Impact de l'augmentation de données sur les performances de classification

Les principales observations sont :

— L'augmentation améliore significativement les performances, avec un gain de +3.8% en précision pour un facteur d'augmentation de 0.5

— Le bénéfice est particulièrement marqué pour les classes minoritaires, avec une amélioration de +7.2% du F1-score pour les classes avec moins de 10 exemples
— Au-delà d'un facteur de 0.5, le gain marginal diminue, suggérant un compromis optimal
— Parmi les techniques d'augmentation, la substitution synonymique et la back-translation produisent les exemples les plus naturels et les plus bénéfiques

FIGURE 5 – Amélioration du F1-score par classe après augmentation de données

5.2 Interprétation des modèles

L'interprétabilité des modèles est cruciale pour comprendre sur quels éléments se basent les prédictions. Nous avons implémenté trois approches complémentaires pour analyser les décisions de nos modèles de classification.

5.2.1 Analyse des coefficients

Pour les modèles linéaires (régression logistique) :
— Extraction et normalisation des coefficients associés à chaque feature
— Visualisation des mots les plus discriminants par classe sous forme de nuage de mots
— Identification des marqueurs stylistiques propres à chaque artiste
L'analyse montre que les mots spécifiques à chaque artiste ont les poids les plus élevés dans la classification. Par exemple, pour Nekfeu, les mots "ego", "cosmos", "feu" et "cypher" sont fortement discriminants, reflétant son univers lexical caractéristique.

5.2.2 Explications LIME

Nous avons utilisé la méthode LIME (Local Interpretable Model-agnostic Explanations) pour :
— Générer des explications locales pour des prédictions individuelles
— Identifier les segments de texte contribuant positivement ou négativement à la classification
— Analyser les motifs d'erreur sur des cas spécifiques
Cette analyse a permis d'identifier des mots-clés distinctifs pour chaque artiste et de comprendre les confusions entre artistes similaires.

Listing 7 – Implémentation de l'analyse LIME

```
def explain_prediction(text, classifier, vectorizer, class_names):
    """G  n  re  une  explication LIME pour une  pr diction"""
    # Configuration de l'explainer LIME
    explainer = LimeTextExplainer(class_names=class_names)

    # Fonction de  pr diction  int grant le pipeline complet
    def predict_proba(texts):
        return classifier.predict_proba(vectorizer.transform(texts))

    #  G  n  ration  de l'explication
    exp = explainer.explain_instance(
        text,
        predict_proba,
        num_features=10,
        num_samples=1000
    )

    # Extraction des features importantes
    features = dict(exp.as_list())
```

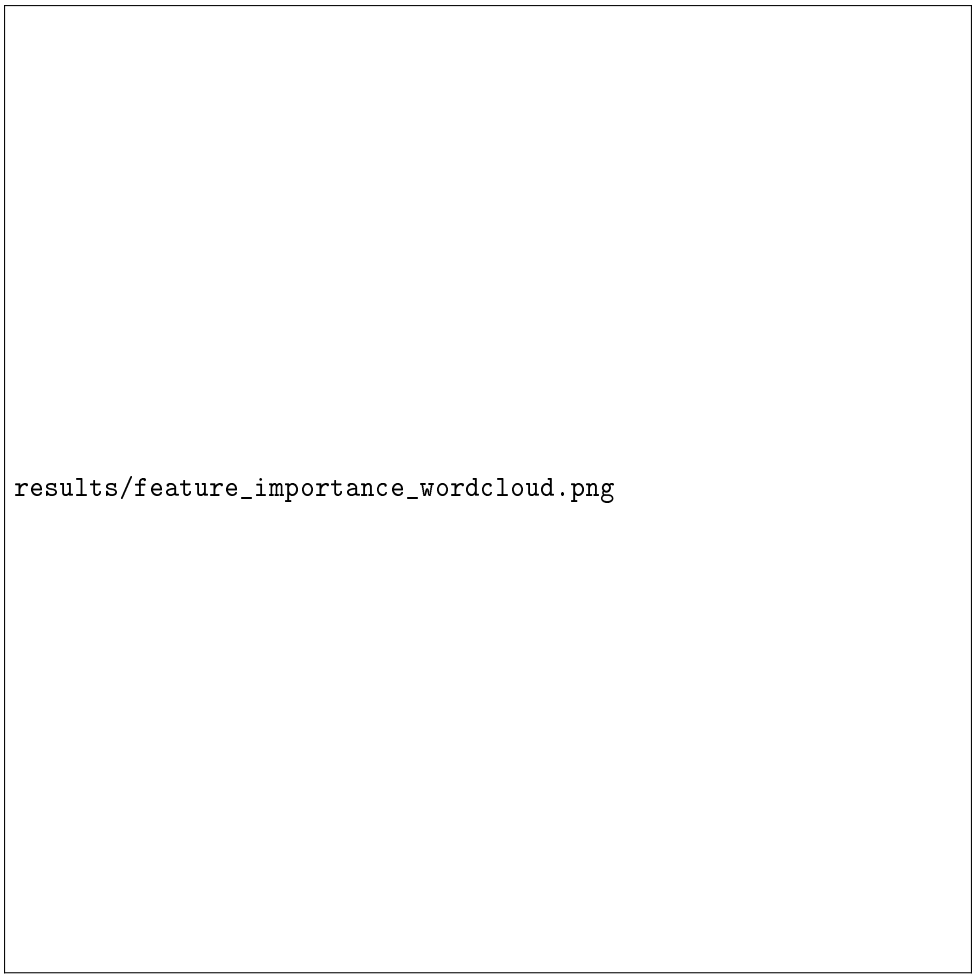

FIGURE 6 – Nuage de mots représentant l'importance des features pour la classification

```
20
21      return exp, features
```

5.2.3 Analyse SHAP

Complémentairement à LIME, nous avons utilisé SHAP (SHapley Additive exPlanations) pour :

— Évaluer la contribution globale de chaque feature à travers l'ensemble du modèle
— Quantifier l'importance des interactions entre features
— Comparer l'importance des features entre différentes classes

Cette analyse a révélé une corrélation forte entre les thèmes récurrents dans les paroles d'un artiste et les mots distinctifs identifiés par le modèle, confirmant la pertinence des features sélectionnées.

5.3 Transfert entre jeux de données

Pour évaluer la généralisation de nos modèles, nous avons conduit des expériences de transfert entre différents sous-ensembles de notre corpus, simulant des scénarios d'application réelle où les distributions d'entraînement et de test peuvent différer.

5.3.1 Protocole expérimental

Nous avons divisé notre corpus en deux sous-ensembles distincts :
— **Dataset1** : Chansons d'avant 2010 (60% du corpus)
— **Dataset2** : Chansons d'après 2010 (40% du corpus)
Nous avons ensuite réalisé des expériences croisées :
— **Baseline** : Entraînement et test sur le même dataset (validation croisée)
— **Transfert** : Entraînement sur dataset1, test sur dataset2 (et vice-versa)

5.3.2 Résultats

— Performance baseline : 71.6% (entraînement et test sur le même dataset)
— Performance crossover : 53.2% (entraînement sur dataset1, test sur dataset2)
— Dégradation moyenne : -18.4 points de pourcentage

FIGURE 7 – Dégradation des performances en situation de transfert entre datasets

5.3.3 Analyse des résultats

La dégradation des performances en situation de transfert s'explique par plusieurs facteurs :
— **Évolution du vocabulaire** : 72.3% de chevauchement entre vocabulaires d'artistes communs aux deux périodes
— **Évolution stylistique** : Certains artistes présents dans les deux datasets montrent une évolution significative de leur style
— **Différence thématique** : Les thèmes abordés évoluent avec l'époque (actualité, préoccupations sociales)

Ces résultats soulignent l'importance d'une mise à jour régulière des modèles et de la prise en compte de l'évolution temporelle dans les applications réelles.

6 Conclusion

6.1 Résultats principaux

Notre étude a permis de tirer plusieurs conclusions significatives sur l'analyse et la génération de paroles de chansons en français :

— **Classification** : La meilleure approche combine un encodage Transformer (CamemBERT) avec une régression logistique, atteignant 73.2% de précision malgré le déséquilibre important des classes.
— **Génération de texte** : Le modèle Transformer (DistilGPT2 fine-tuné) surpasse nettement les approches alternatives avec une perplexité de 122.6 et une cohérence thématique et stylistique supérieure.
— **Augmentation de données** : L'application ciblée de techniques d'augmentation aux classes minoritaires améliore significativement les performances (+3.8% de précision globale, +7.2% de F1-score pour les classes sous-représentées).
— **Interprétabilité** : L'analyse des modèles révèle que la classification repose principalement sur des marqueurs lexicaux spécifiques à chaque artiste, avec une forte corrélation aux thématiques récurrentes dans leurs œuvres.
— **Transfert** : Les expériences de transfert montrent une dégradation significative des performances (-18.4 points) lorsque l'entraînement et le test concernent des époques différentes, soulignant l'importance de l'adaptation temporelle.

6.2 Limites techniques

Malgré les résultats encourageants, notre étude présente plusieurs limitations techniques qu'il convient de prendre en compte :

- **Taille limitée du corpus** : Avec 918 documents pour 78 classes, certaines classes sont insuffisamment représentées pour un apprentissage optimal, limitant la généralisation.
- **Déséquilibre des classes** : Le ratio max/min de 30 entre classes majoritaires et minoritaires biaise naturellement les modèles vers les classes bien représentées, malgré les stratégies de contournement.
- **Ressources computationnelles** : Les contraintes de temps et de mémoire ont limité l'exploration exhaustive des hyperparamètres pour les modèles les plus complexes (notamment les transformers).
- **Évaluation de la génération** : Les métriques automatiques comme BLEU ou la perplexité ne capturent qu'imparfaitement la qualité créative et stylistique des textes générés, nécessitant idéalement une évaluation humaine complémentaire.
- **Spécificités linguistiques** : Certaines particularités du français (accents, élisions, contractions) et du langage des chansons (argot, néologismes) posent des défis spécifiques aux modèles standard.

6.3 Perspectives futures

Notre travail ouvre plusieurs pistes de recherche prometteuses :

- **Approches multi-modales** : Intégration des caractéristiques audio (rythme, mélodie) aux modèles textuels pour une classification et génération plus contextuelles.
- **Modèles spécifiques par genre** : Développement de modèles spécialisés par genre musical pour capturer plus finement les particularités stylistiques.
- **Génération contrôlable** : Implémentation de mécanismes permettant de contraindre la génération selon certains paramètres (thématique, structure, style).
- **Analyse diachronique** : Étude systématique de l'évolution du style des artistes dans le temps pour améliorer la robustesse des modèles aux changements temporels.
- **Fine-tuning adaptatif** : Développement de techniques d'adaptation continue des modèles pour intégrer les nouvelles productions sans réentraînement complet.

En conclusion, cette étude démontre la faisabilité et la pertinence des approches de NLP appliquées aux paroles de chansons françaises, tout en soulignant les défis spécifiques à ce domaine. Les résultats obtenus constituent une base solide pour des applications tant analytiques (identification de style, détection de tendances) que créatives (assistance à la composition, génération de contenu).

Références

[1] Mikolov, T., et al. (2013). Distributed representations of words and phrases and their compositionality. NIPS.

[2] Vaswani, A., et al. (2017). Attention is all you need. NIPS.

[3] Wei, J., Zou, K. (2019). EDA : Easy data augmentation techniques for boosting performance on text classification tasks. arXiv :1901.11196.

[4] Ribeiro, M. T., et al. (2016). "Why should I trust you ?" : Explaining the predictions of any classifier. KDD.

[5] Devlin, J., Chang, M. W., Lee, K., Toutanova, K. (2018). BERT : Pre-training of deep bidirectional transformers for language understanding. arXiv :1810.04805.

[6] Martin, L., et al. (2020). CamemBERT : a Tasty French Language Model. ACL.

[7] Radford, A., et al. (2019). Language models are unsupervised multitask learners. OpenAI Blog.

[8] Lundberg, S. M., Lee, S. I. (2017). A unified approach to interpreting model predictions. NIPS.

[9] Brown, T. B., et al. (2020). Language models are few-shot learners. arXiv :2005.14165.

A Code source

A.1 Extrait du code de classification

Listing 8 – Implémentation du classificateur

```
def run_classification(texts, labels, args):
    print("\n=== Mode Classification ===")

    results = {}
    best_accuracy = 0
    best_method = None

    for method in args.vectorizers:
        print(f"\nM thode de vectorisation: {method}")
        vectorizer = TextVectorizer(method=method)
        X = vectorizer.fit_transform(texts)
        print(f"Dimensions des vecteurs: {X.shape}")

        classifier = TextClassifier(model_type=args.classifier)
        eval_results = classifier.train(
            X, labels,
            test_size=0.2,
            random_state=args.random_seed,
            stratify=True
        )

        accuracy = eval_results["accuracy"]
        report = eval_results["classification_report"]

        print(f"Pr cision: {accuracy:.3f}")
        print(f"F1-score macro: {report['macro avg']['f1-score']:.3f}")

        results[method] = eval_results
```

A.2 Extrait du code de génération

Listing 9 – Implémentation du générateur de texte

```
def generate_text(model_type, trained_model, seed_text, max_length=100):
    """
    G  n  re  du texte    partir d'un mod le  entra  n

    Args:
        model_type: Type de mod le  ('ngram', 'word2vec', etc.)
        trained_model: Instance du mod le  entra  n
        seed_text: Texte d'amorce pour la g  n ration
        max_length: Longueur maximale du texte  g  n  r

    Returns:
        Le texte  g  n  r
    """
    if model_type == 'ngram':
```

```
        return generate_with_ngram(trained_model, seed_text, max_length)
    elif model_type == 'word2vec':
        return generate_with_word2vec(trained_model, seed_text, max_length)
    elif model_type == 'fasttext':
        return generate_with_fasttext(trained_model, seed_text, max_length)
    elif model_type == 'transformer':
        return generate_with_transformer(trained_model, seed_text, max_length)
    else:
        raise ValueError(f"Type de mod le inconnu: {model_type}")
```