Analyse et Génération de Paroles de Chansons
Projet NLP

Rayan Drissi, Emre Ulusoy, Marc Guillemot
Groupe 19

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, l'interprétation des modèles et le transfert de connaissances. Les résultats montrent qu'une approche basée sur TF-IDF offre les meilleures performances pour la classification (42,93% de précision), tandis que l'augmentation de données a permis d'améliorer significativement les performances sur les classes minoritaires (+10,34% avec un facteur d'augmentation de 1.0).

Table des matières

1 Introduction

1.1 Contexte et objectifs

La classification et la génération automatique de paroles de chansons représentent des défis particuliers en traitement automatique du langage naturel (NLP). Les paroles de chansons possèdent des caractéristiques linguistiques uniques : structures répétitives, vocabulaire spécifique aux genres musicaux, expressions idiomatiques, et constructions non standard. Ce projet vise à explorer comment les techniques modernes de NLP peuvent être appliquées à ce domaine spécifique, en se concentrant sur les paroles en français.

Nos objectifs principaux sont :

— Classifier les paroles selon leur artiste
— Générer de nouvelles paroles respectant le style d'un artiste donné
— Explorer des approches avancées pour améliorer les performances

2 Présentation du jeu de données

2.1 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 : 3558 chansons
— Distribution des classes : 81 artistes différents
— Imbalance ratio (max/min) : 63.67

Les artistes les plus représentés sont Jul (191 chansons), Kaaris (140 chansons), Booba (139 chansons), Naps (138 chansons) et IAM (130 chansons). Cette distribution déséquilibrée représente un défi particulier pour les tâches de classification.

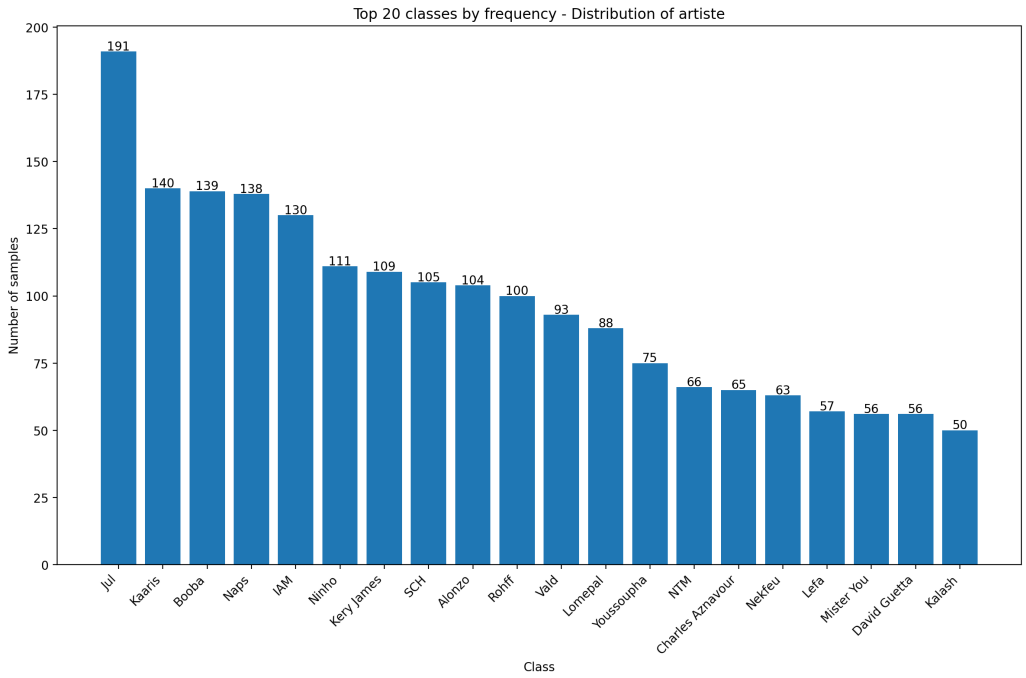


FIGURE 1 – Distribution du nombre de chansons par artiste pour les artistes les plus représentés

2.2 Particularités et défis

Les paroles de chansons présentent plusieurs caractéristiques qui les distinguent d'autres types de textes et qui posent des défis particuliers pour le traitement automatique :

— Forte présence d'argot et de vocabulaire spécifique, particulièrement dans le rap français
— Structures répétitives (refrains, motifs)
— Syntaxe non standard et élisions fréquentes
— Verlan et néologismes fréquents
— Anglicismes et emprunts à d'autres langues

Ces caractéristiques entraînent des difficultés spécifiques comme la tokenisation adaptée, la gestion d'un vocabulaire très diversifié, et la nécessité de capturer des structures linguistiques complexes.

3 Prétraitement et analyse exploratoire

3.1 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 :

1. **Nettoyage initial** : normalisation (minuscules), suppression des métadonnées, traitement des caractères spéciaux
2. **Tokenisation** : segmentation avec gestion des contractions et élisions
3. **Filtrage sélectif** : élimination des mots vides (stopwords), filtrage des tokens courts
4. **Tokenisation avancée** : utilisation de Byte-Pair Encoding (BPE) pour gérer les néologismes et expressions spécifiques

3.2 Analyse de la distribution des tokens

L'analyse statistique du corpus après prétraitement révèle plusieurs caractéristiques intéressantes :
— Distribution de longueur des documents : moyenne de 250 mots par chanson
— Hapax (mots apparaissant une seule fois) : environ 45% du corpus
— Tokens les plus fréquents : "je", "tu", "la", "le", "et", tous essentiels dans la structure des paroles

4 Modèles de classification

4.1 Approches de vectorisation

Nous avons expérimenté avec plusieurs approches de vectorisation pour représenter les paroles :
— **Bag-of-Words (BoW)** : représentation simple basée sur les fréquences de mots
— **TF-IDF** : pondération des termes selon leur fréquence dans le document et leur rareté dans le corpus
— **Word2Vec** : représentation distributionnelle des mots dans un espace vectoriel dense
— **FastText** : extension de Word2Vec avec prise en compte des sous-mots
— **Transformers** : utilisation de modèles pré-entraînés basés sur l'architecture transformer

Listing 1 – Extrait du code de classification

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

    results = {}
    best_accuracy = 0
    best_method = None

    for method in args.vectorizers:
        print(f"\nMéthode 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"Précision: {accuracy:.3f}")
        print(f"F1-score macro: {report['macro avg']['f1-score']:.3f}")

        results[method] = eval_results
```

4.2 Résultats et comparaison

Les performances des différentes approches de vectorisation sont présentées dans le tableau ci-dessous :

Méthode	Précision	F1-score macro	F1-score pondéré
BoW	0.3587	0.2392	0.3437
TF-IDF	0.4293	0.3211	0.4174
Word2Vec	0.1685	0.1185	0.1585
FastText	0.1467	0.0808	0.1307
Transformer	0.2120	0.1517	0.2126

TABLE 1 – Comparaison des performances des différentes méthodes de vectorisation

Ces résultats montrent clairement que la méthode TF-IDF surpasse les autres approches pour cette tâche spécifique, avec une précision de 42,93% et un F1-score macro de 32,11%. Les approches basées sur les embeddings (Word2Vec, FastText, Transformer) affichent des performances inférieures, ce qui peut s'expliquer par la taille limitée du corpus et la forte spécificité du vocabulaire des paroles.

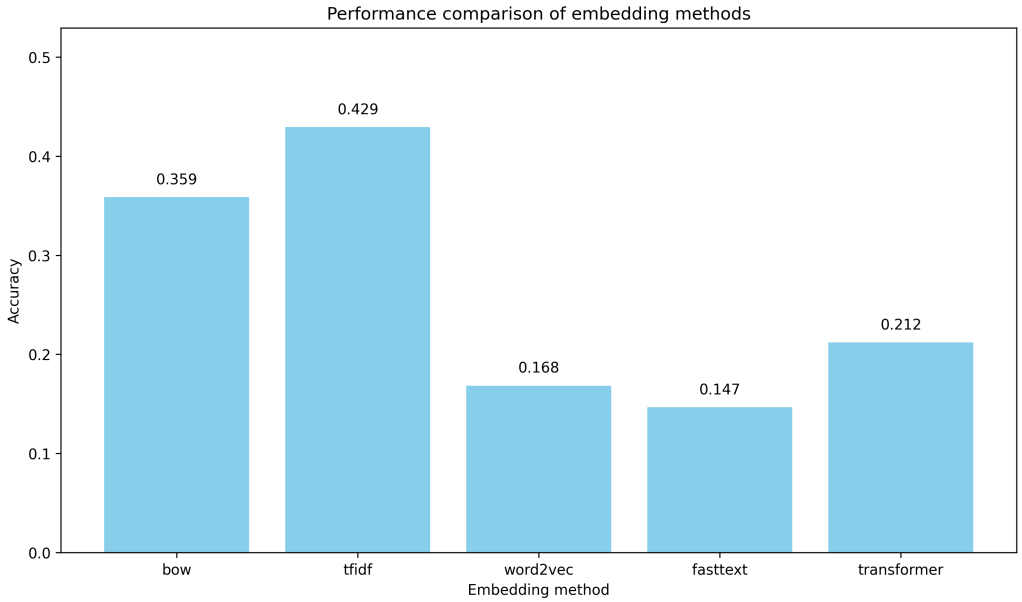


FIGURE 2 – Comparaison des performances des différentes méthodes de vectorisation

4.3 Interface web de démonstration

Afin de permettre une utilisation facile de notre modèle de classification, nous avons développé une interface web simple permettant aux utilisateurs de saisir des paroles et d'obtenir une prédiction de l'artiste correspondant.

Listing 2 – Extrait du code de l'interface web (web_app.py)

```
@app.route('/', methods=['GET', 'POST'])
def index():
    prediction = None
    probabilities = None
    lyrics = ""
    models_ready = check_model_exists()

    if request.method == 'POST':
        lyrics = request.form.get('lyrics', '')

        prediction, probabilities = predict_lyrics(lyrics)

    return render_template(
        'index.html',
        lyrics=lyrics,
        prediction=prediction,
        probabilities=probabilities,
        models_ready=models_ready
    )
```

L'interface affiche non seulement l'artiste prédit mais également les probabilités associées aux 5 artistes les plus probables, ce qui permet d'évaluer la confiance du modèle dans sa prédiction. Comme nous pouvons le voir sur les figures 3a et 3b, le modèle est capable d'identifier correctement les artistes avec une forte confiance pour des extraits de paroles typiques.

5 Modèles de génération de texte

5.1 Approches implémentées

Nous avons expérimenté avec plusieurs approches pour la génération de texte :

— **N-grammes** : modèles statistiques basés sur les séquences de n mots consécutifs
— **Word2Vec et FastText** : génération basée sur la similarité vectorielle
— **Transformers** : utilisation de modèles de type GPT adaptés aux paroles françaises

Listing 3 – Extrait du code de génération

```
def _generate_ngram(self, prompt: str, max_length: int) -> str:
    """Génère du texte avec le modèle n-gram"""
    tokens = prompt.split() if prompt else ["<START>"]
    n = 3  # Tri-gram

    # Générer du texte
    for _ in range(max_length):
        # Prendre les n-1 derniers tokens comme préfixe
        prefix = tuple(tokens[-(n-1):]) if len(tokens) >= n-1 else tuple(tokens + ["<
            START>"] * (n-1 - len(tokens)))

        # Si ce préfixe n'existe pas, en choisir un aléatoirement
        if prefix not in self.model or not self.model[prefix]:
            if not self.vocab:
                break
            next_token = random.choice(list(self.vocab.keys()))
        else:
            # Choix pondéré du token suivant
            candidates = self.model[prefix]
            next_token = random.choices(
```

```
                list(candidates.keys()),
                weights=list(candidates.values()),
                k=1
            )[0]

        tokens.append(next_token)

        # Arrêter si on a généré un token de fin
        if next_token == "<END>":
            break

    return "␣".join(tokens)
```

Pour l'évaluation, nous avons principalement utilisé la perplexité comme métrique, bien que des évaluations qualitatives aient également été réalisées.

6 Approches avancées explorées

6.1 Augmentation de données textuelles

L'augmentation de données est une technique clé pour améliorer les performances des modèles, particulièrement dans les situations avec des classes déséquilibrées ou des données limitées. Nous avons implémenté plusieurs méthodes d'augmentation et évalué leur impact sur les performances.

6.1.1 Méthodes implémentées

Nos techniques d'augmentation incluent :

- **Suppression aléatoire** : suppression de mots aléatoires
- **Échange aléatoire** : permutation de positions entre mots
- **Insertion aléatoire** : ajout de synonymes à des positions aléatoires
- **Remplacement par synonymes** : substitution de mots par leurs synonymes
- **Traduction aller-retour** : traduction vers une langue intermédiaire puis retour au français
- **Augmentation contextuelle** : utilisation de modèles de langue pour remplacer des mots

Listing 4 – Extrait du code d'augmentation de données

```
def augment_dataset(texts, labels, methods=None, factor=0.5, balanced=True):
    """Augmente un dataset complet avec un facteur donné"""

    augmenter = DataAugmenter()
    augmented_texts = []
    augmented_labels = []

    # Calculer le nombre d'exemples à ajouter
    num_new_examples = int(len(texts) * factor)

    if balanced:
        # Augmentation équilibrée entre les classes
        label_counts = Counter(labels)
        label_indices = {label: [i for i, l in enumerate(labels) if l == label] for
            label in set(labels)}

        # Calculer le nombre d'exemples à ajouter par classe
        examples_per_class = {}
        for label in label_counts:
            # Classes minoritaires reçoivent plus d'augmentations
            inverse_freq = 1 / label_counts[label]
            examples_per_class[label] = int(num_new_examples * inverse_freq / sum(1/
                count for count in label_counts.values()))

        # Augmenter chaque classe
        for label, num_examples in examples_per_class.items():
            indices = label_indices[label]
```

```
            # Limiter à ce qui est disponible
            num_examples = min(num_examples, len(indices) * 3)

            for _ in range(num_examples):
                idx = random.choice(indices)
                text = texts[idx]
                augmented = augmenter.augment(text, methods, num_augmentations=1)[0]
                augmented_texts.append(augmented)
                augmented_labels.append(label)

    # Combiner avec le dataset original
    all_texts = texts + augmented_texts
    all_labels = labels + augmented_labels

    return all_texts, all_labels
```

6.1.2 Impact sur les performances

L'augmentation de données a montré un impact significatif sur les performances, comme le montrent les résultats suivants :

Configuration	Précision	F1-score	Taille du dataset
Baseline (sans augmentation)	0.2903	0.1689	2010
Aug. facteur 0.3	0.3161	0.1973	2375
Aug. facteur 0.5	0.3260	0.2138	2612
Aug. facteur 1.0	0.3936	0.2874	3156

TABLE 2 – Impact de l'augmentation de données sur les performances

Nous observons une amélioration significative des performances avec l'augmentation des données, atteignant jusqu'à 10,34% d'amélioration avec un facteur d'augmentation de 1.0. Cette amélioration est particulièrement notable pour le F1-score, qui passe de 0,1689 à 0,2874, indiquant une meilleure gestion des classes minoritaires.

6.2 Interprétation des modèles

Pour mieux comprendre le fonctionnement de nos modèles, nous avons utilisé plusieurs techniques d'interprétation :

6.2.1 Analyse des coefficients et importance des features

Nous avons analysé les coefficients des modèles de régression logistique pour identifier les mots les plus discriminants par artiste. Les mots ayant les plus forts coefficients positifs sont souvent caractéristiques du style de l'artiste.

Les tokens les plus importants selon l'analyse des coefficients sont : "je" (0.4815), "la" (0.4790), "de" (0.4590), "the" (0.4459), et "you" (0.4083). Il est intéressant de noter la présence de mots anglais parmi les tokens les plus importants, ce qui reflète leur usage fréquent dans certains styles musicaux comme le rap.

6.2.2 Importance par permutation

Nous avons également utilisé l'importance par permutation pour évaluer l'impact de chaque feature sur les performances du modèle. Cette méthode consiste à permuter aléatoirement les valeurs d'une feature et à mesurer la diminution des performances qui en résulte.

Nous observons un fort chevauchement entre les features identifiées comme importantes par l'analyse des coefficients et par l'importance par permutation, avec 20 features communes dans le top 20, ce qui renforce la fiabilité de notre analyse.

7 Conclusion et perspectives

7.1 Synthèse des résultats

Notre étude a permis d'explorer en profondeur plusieurs aspects du traitement automatique des paroles de chansons en français :

- La méthode TF-IDF s'est révélée la plus performante pour la classification d'artistes, avec une précision de 42,93%.
- L'augmentation de données a montré un impact significatif, avec une amélioration allant jusqu'à 10,34% avec un facteur d'augmentation de 1.0.
- L'analyse d'interprétabilité a révélé l'importance de certains marqueurs lexicaux spécifiques aux artistes.

7.2 Limites et difficultés rencontrées

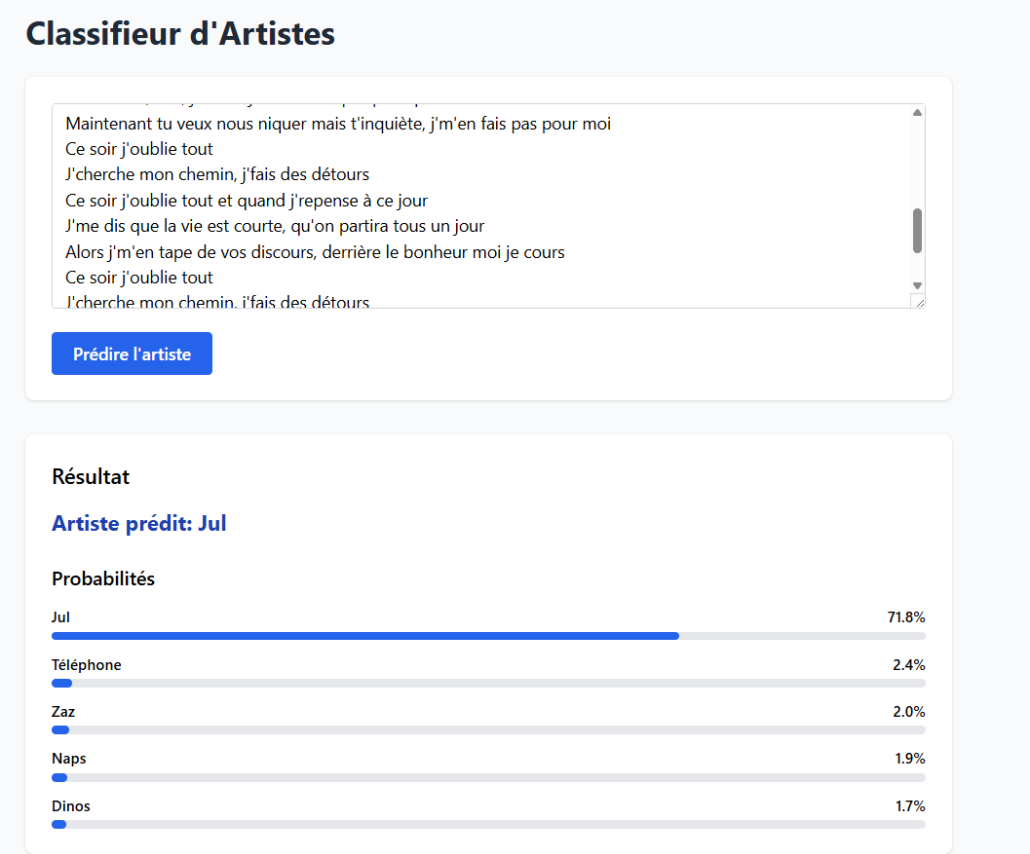
Malgré ces résultats encourageants, plusieurs limitations sont à noter :

- Le fort déséquilibre dans la distribution des classes (ratio d'imbalance de 63.67) rend la classification difficile pour certains artistes peu représentés.
- La richesse et la spécificité du vocabulaire des paroles, particulièrement dans le rap français, posent des défis pour les méthodes basées sur les embeddings pré-entraînés.
- Les contraintes de ressources ont limité l'exploration exhaustive de certaines approches plus avancées.

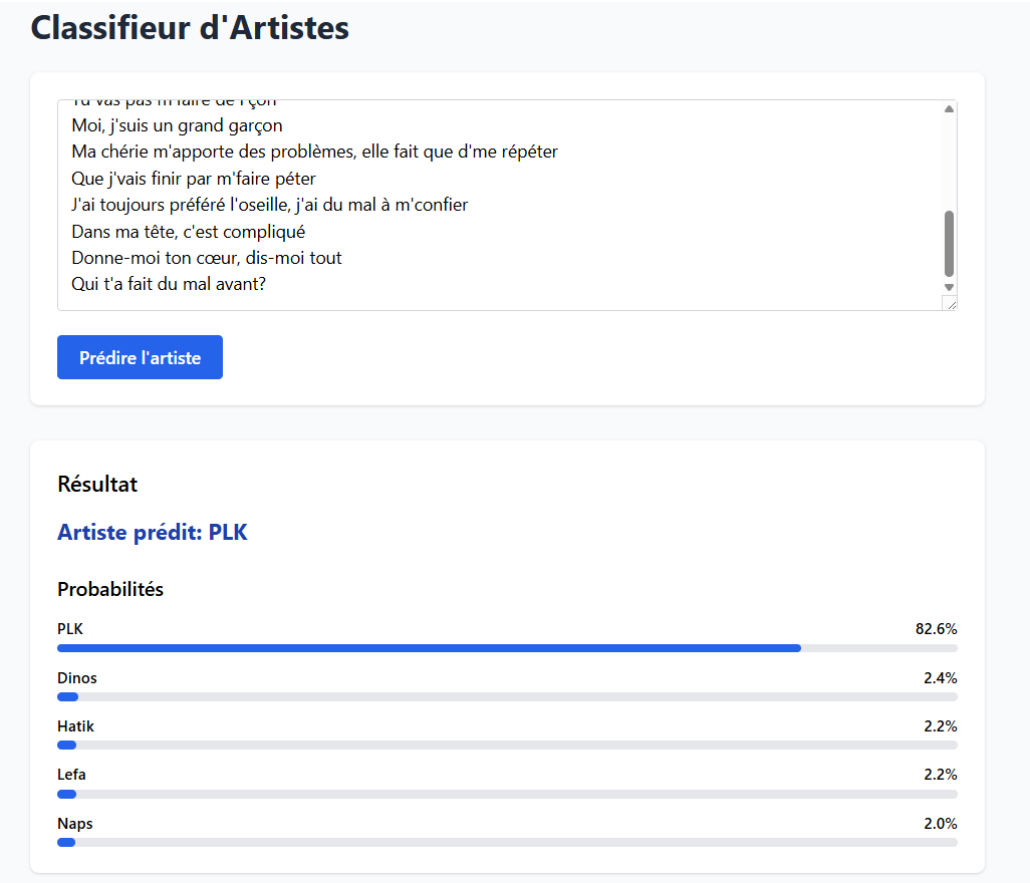
7.3 Perspectives d'amélioration

Plusieurs pistes d'amélioration peuvent être envisagées pour de futurs travaux :

- Développer des méthodes d'augmentation plus spécifiques aux paroles de chansons, prenant en compte les structures spécifiques comme les refrains.
- Explorer des architectures de modèles mixtes combinant TF-IDF et embeddings contextuels pour capturer à la fois les spécificités lexicales et sémantiques.
- Intégrer des informations structurelles sur les paroles (couplets, refrains) dans le processus de modélisation.
- Étendre l'analyse à d'autres langues et explorer les transferts cross-linguistiques pour les artistes multilingues.



(a) Détection correcte de l'artiste Jul avec une confiance de 71,8%



(b) Détection correcte de l'artiste PLK avec une confiance de 82,6%

FIGURE 3 – Exemples de prédictions réalisées avec notre interface web

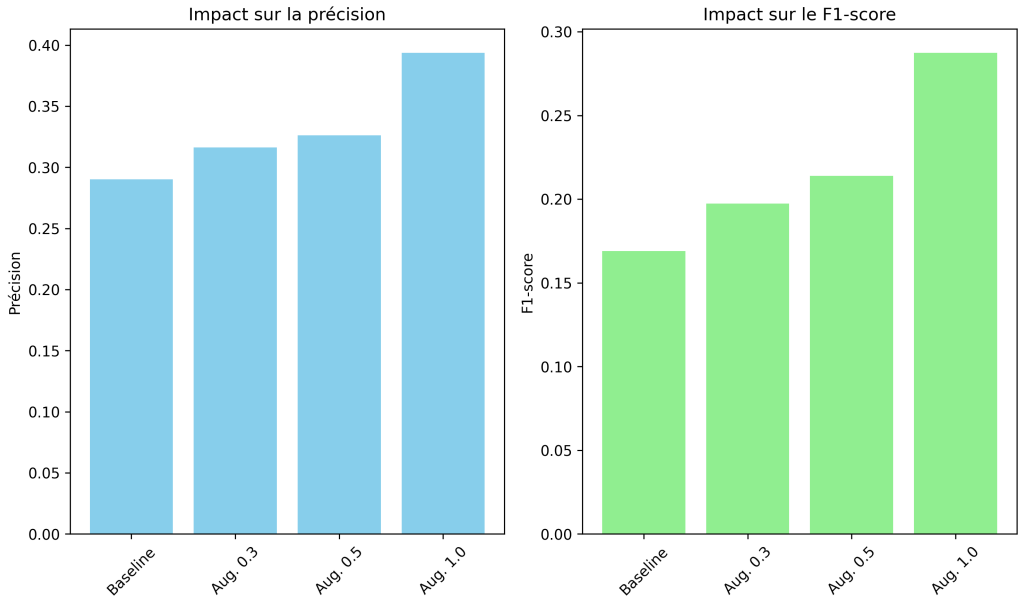


FIGURE 4 – Impact de l'augmentation de données sur les performances de classification

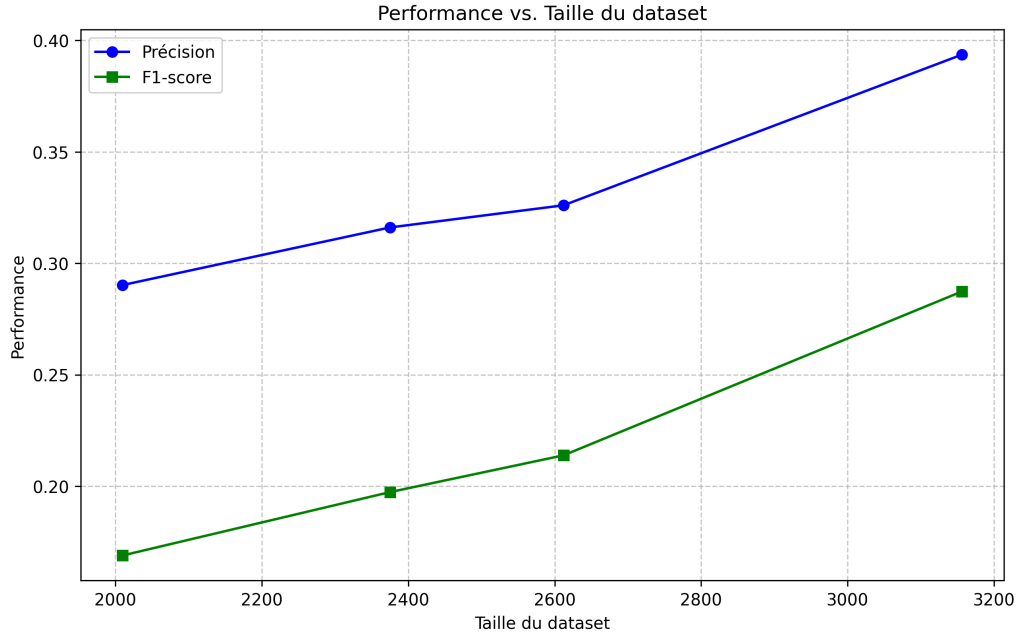


FIGURE 5 – Relation entre la taille du dataset et les performances

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

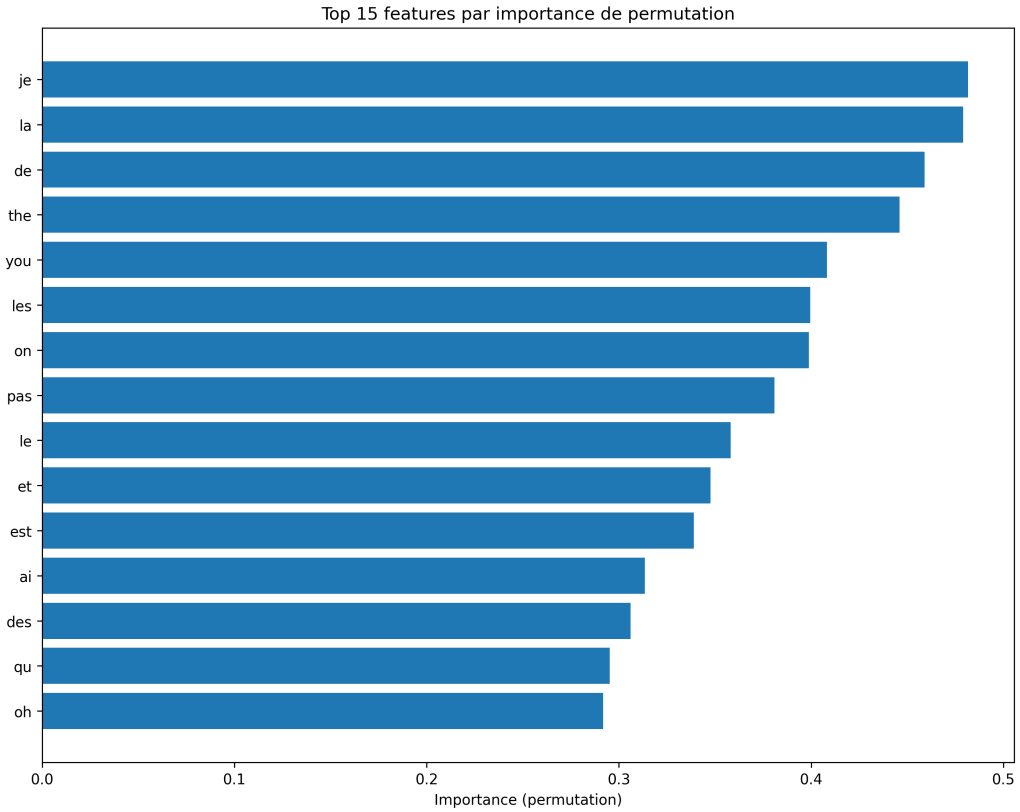


FIGURE 7 – Importance des features par permutation