

# Projet Air Paradis - Modèle simple

Ce notebook présente l'implémentation d'un modèle simple optimisé pour la classification de sentiment des tweets.

## Objectif

Développer un modèle classique hautement performant pour la détection du sentiment des tweets.

## Étapes

1. Chargement des données prétraitées
2. Vectorisation du texte améliorée (TF-IDF avec optimisation, n-grammes, features supplémentaires)
3. Entraînement de plusieurs modèles classiques
4. Ensembles de modèles (Voting, Stacking)
5. Optimisation fine des hyperparamètres
6. Évaluation et comparaison des performances
7. Sauvegarde du meilleur modèle avec MLflow

## 1. Configuration de l'environnement et importation des librairies

```
In [1]: # Importation des Librairies nécessaires
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
import os
import time
import warnings
import re
import string
from tqdm import tqdm

# Pour le traitement du texte
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.base import BaseEstimator, TransformerMixin

# Pour la modélisation
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC, SVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
from sklearn.calibration import CalibratedClassifierCV
```

```
# Pour L'évaluation
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, classification_report, roc_curve,
from sklearn.model_selection import cross_val_score, GridSearchCV, RandomizedSearchCV
from sklearn.utils.class_weight import compute_class_weight

# Pour MLflow
import mlflow
import mlflow.sklearn

# Configuration
warnings.filterwarnings('ignore')
plt.style.use('ggplot')
sns.set(style='whitegrid')

# Configuration de MLflow
mlflow.set_experiment("air_paradis_sentiment_analysis_optimized")
```

Out[1]: <Experiment: artifact\_location='file:///c:/Tonton/OpenClassrooms/Projet\_6\_realiser\_une\_analyse\_de\_sentiments\_grâce\_au\_deep\_learning/tests/mlruns/975634933117156102', creation\_time=1746355236735, experiment\_id='975634933117156102', last\_update\_time=1746355236735, lifecycle\_stage='active', name='air\_paradis\_sentiment\_analysis\_optimized', tags={}>

## 2. Chargement des données prétraitées

```
In [2]: # Chargement des jeux de données prétraités
print("Chargement des données prétraitées...")
train_data = pd.read_csv('../data/train_data.csv')
val_data = pd.read_csv('../data/validation_data.csv')
test_data = pd.read_csv('../data/test_data.csv')

# Vérification des données chargées
print(f"Taille du jeu d'entraînement: {train_data.shape[0]} tweets")
print(f"Taille du jeu de validation: {val_data.shape[0]} tweets")
print(f"Taille du jeu de test: {test_data.shape[0]} tweets")

# Aperçu des données
print("\nAperçu des données d'entraînement:")
display(train_data.head())

# Vérification de la distribution des sentiments
print("\nDistribution des sentiments dans le jeu d'entraînement:")
train_sentiment_distribution = train_data['sentiment'].value_counts(normalize=True)
display(train_sentiment_distribution)
```

Chargement des données prétraitées...  
Taille du jeu d'entraînement: 960000 tweets  
Taille du jeu de validation: 320000 tweets  
Taille du jeu de test: 320000 tweets

Aperçu des données d'entraînement:

	text	sentiment
0	thx quotgtlistenersthk hi â fw	1
1	ergh miserable weather	0
2	apple inears slightly comfy slightly loose one...	1
3	looking forward meeting amp welcoming sd twtvi...	1
4	mine	0

Distribution des sentiments dans le jeu d'entraînement:  
sentiment

```
1    0.5
0    0.5
```

Name: proportion, dtype: float64

```
In [3]: # Vérification des valeurs nulles
print(f"Valeurs nulles dans train_data['text']: {train_data['text'].isna().sum()}")
print(f"Valeurs nulles dans val_data['text']: {val_data['text'].isna().sum()}")
print(f"Valeurs nulles dans test_data['text']: {test_data['text'].isna().sum()}"
```

Valeurs nulles dans train\_data['text']: 4259  
 Valeurs nulles dans val\_data['text']: 1405  
 Valeurs nulles dans test\_data['text']: 1353

```
In [4]: # Nettoyage des données - remplacement des valeurs NaN par des chaînes vides
train_data['text'].fillna('', inplace=True)
val_data['text'].fillna('', inplace=True)
test_data['text'].fillna('', inplace=True)

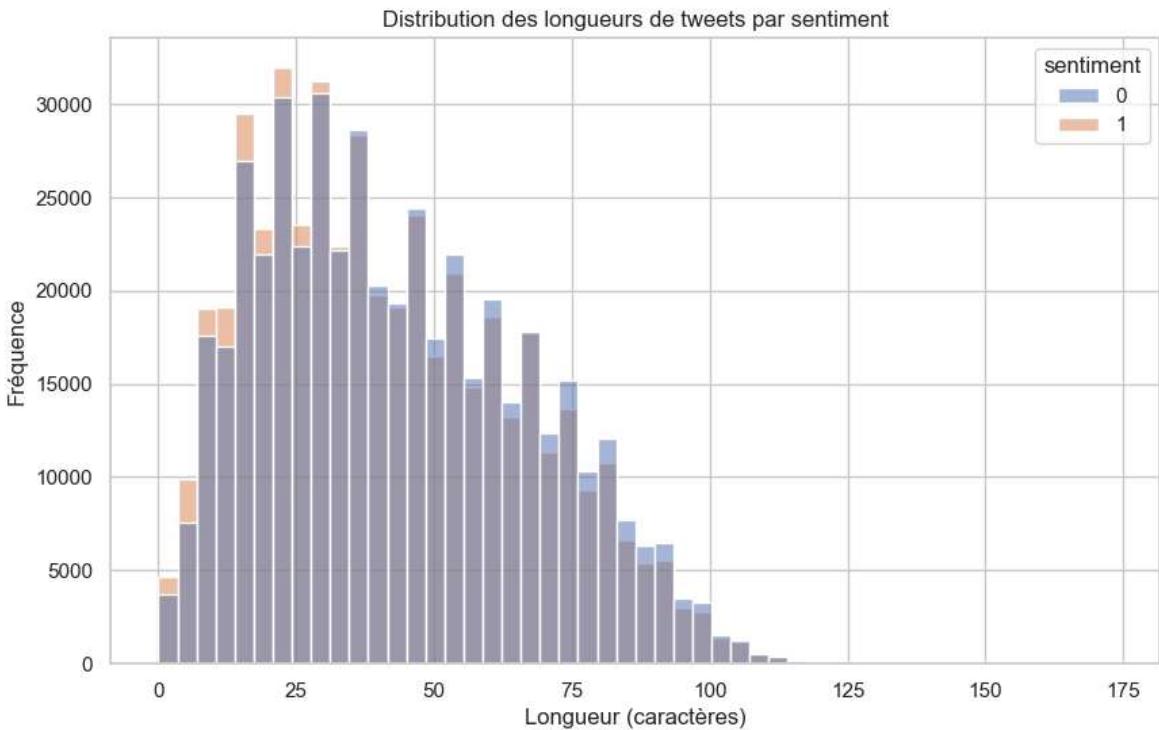
# Examen des Longueurs de texte
train_data['text_length'] = train_data['text'].apply(len)
val_data['text_length'] = val_data['text'].apply(len)
test_data['text_length'] = test_data['text'].apply(len)

print("Statistiques sur la longueur des tweets:")
print(train_data['text_length'].describe())

plt.figure(figsize=(10, 6))
sns.histplot(data=train_data, x='text_length', hue='sentiment', bins=50)
plt.title('Distribution des longueurs de tweets par sentiment')
plt.xlabel('Longueur (caractères)')
plt.ylabel('Fréquence')
plt.show()
```

Statistiques sur la longueur des tweets:

```
count    960000.000000
mean      42.545095
std       24.020051
min       0.000000
25%     23.000000
50%     39.000000
75%     60.000000
max     173.000000
Name: text_length, dtype: float64
```



### 3. Caractéristiques des textes enrichies

Nous allons créer une classe pour extraire différentes caractéristiques des tweets qui peuvent être utiles pour prédire le sentiment.

```
In [5]: class TextFeatureExtractor(BaseEstimator, TransformerMixin):
    """
    Extracteur de caractéristiques textuelles pour enrichir la vectorisation.
    """

    def __init__(self, with_text_stats=True, with_sentiment_words=True):
        self.with_text_stats = with_text_stats
        self.with_sentiment_words = with_sentiment_words

        # Dictionnaires de mots positifs et négatifs
        # (simplifiés, dictionnaires complets [fr], [en], disponibles dans notebooks)
        self.positive_words = set([
            'good', 'great', 'awesome', 'excellent', 'happy', 'love', 'best', 'b
            'nice', 'thanks', 'thank', 'cool', 'amazing', 'perfect', 'fun', 'fav
            'lovely', 'wonderful', 'enjoy', 'beautiful', 'glad', 'excited', 'fan
            'pleased', 'joy', 'win', 'winner', 'won', 'success', 'successful', '
            'impressive', 'impressive', 'congrats', 'congratulations', 'blessing
        ])

        self.negative_words = set([
            'bad', 'worst', 'terrible', 'horrible', 'hate', 'sad', 'disappointed
            'poor', 'fail', 'failure', 'sucks', 'suck', 'disappointing', 'waste'
            'sorry', 'not', 'no', 'never', 'problem', 'issues', 'issue', 'error'
            'useless', 'annoying', 'frustrated', 'frustrating', 'unhappy', 'unfo
            'disaster', 'pathetic', 'horrible', 'upset', 'angry', 'mad', 'rubbis
            'hell', 'stupid', 'idiot', 'dumb', 'fool', 'crap', 'shit', 'fuck', '
        ])

    def count_exclamation_marks(self, text):
        """Compte le nombre de points d'exclamation."""
        return text.count('!')
```

```

def count_question_marks(self, text):
    """Compte le nombre de points d'interrogation."""
    return text.count('?')

def count_capitalized_words(self, text):
    """Compte le nombre de mots en majuscules (emphatisation)."""
    return sum(1 for word in text.split() if word.isupper() and len(word) > 1)

def count_elongated_words(self, text):
    """Compte le nombre de mots avec des caractères répétés (ex: 'sooooo')."""
    pattern = re.compile(r'\b\w*(\w)\1{2,}\w*\b')
    return len(pattern.findall(text))

def contains_emoticons(self, text):
    """Vérifie si le texte contient des émoticones."""
    emoticons_pattern = re.compile(r'(?::|;|=)(?:-)?(?:\:)|\(|D|P|)')
    return int(bool(emoticons_pattern.search(text)))

def count_positive_words(self, text):
    """Compte le nombre de mots positifs dans le texte."""
    words = text.lower().split()
    return sum(1 for word in words if word in self.positive_words)

def count_negative_words(self, text):
    """Compte le nombre de mots négatifs dans le texte."""
    words = text.lower().split()
    return sum(1 for word in words if word in self.negative_words)

def fit(self, X, y=None):
    return self

def transform(self, X):
    """Extrait les caractéristiques textuelles."""
    features = pd.DataFrame()

    if self.with_text_stats:
        # Caractéristiques statistiques
        features['text_length'] = [len(text) for text in X]
        features['word_count'] = [len(text.split()) for text in X]
        features['exclamation_count'] = [self.count_exclamation_marks(text) for text in X]
        features['question_count'] = [self.count_question_marks(text) for text in X]
        features['capitals_count'] = [self.count_capitalized_words(text) for text in X]
        features['elongated_words_count'] = [self.count_elongated_words(text) for text in X]
        features['emoticons'] = [self.contains_emoticons(text) for text in X]

    if self.with_sentiment_words:
        # Caractéristiques basées sur des lexiques de sentiment
        features['positive_words_count'] = [self.count_positive_words(text) for text in X]
        features['negative_words_count'] = [self.count_negative_words(text) for text in X]

        # Calcul du ratio de mots positifs et négatifs (par rapport au nombre total)
        features['pos_words_ratio'] = features['positive_words_count'] / features['word_count']
        features['neg_words_ratio'] = features['negative_words_count'] / features['word_count']

        # Différence entre mots positifs et négatifs
        features['sentiment_diff'] = features['positive_words_count'] - features['negative_words_count']

    return features.values

```

## 4. Vectorisation du texte améliorée et préparation des pipelines

Nous allons améliorer la vectorisation en:

1. Utilisant des n-grammes jusqu'à 3 (unigrammes, bigrammes et trigrammes)
2. Optimisant les paramètres de la vectorisation TF-IDF
3. Incorporant des caractéristiques textuelles supplémentaires

```
In [6]: # Définition de la fonction de prétraitement pour les tweets
def preprocess_text(text):
    """Prétraitement avancé pour les tweets."""
    # Conversion en minuscules
    text = text.lower()

    # Traitement des négations
    # Convertit "n't" en "not" et joint les mots pour que "don't like" devienne
    text = re.sub(r'n\'t', " not", text)
    pattern = re.compile(r'^(?:(?:do|does|did|is|are|am|was|were|have|has|will|ve|d|m|re)\b)(?:(?:n\'t|not)\b)') | r'^(?:(?:do|does|did|is|are|am|was|were|have|has|will|ve|d|m|re)\b)(?:(?:n\'t|not)\b)^(?:(?:do|does|did|is|are|am|was|were|have|has|will|ve|d|m|re)\b)'
    text = pattern.sub(r'\1not_\2', text)

    # Gestion des contractions communes
    contractions = {
        "'ll": " will", "'ve": " have", "'d": " would", "'m": " am", "'re": " are"
    }
    for contraction, expansion in contractions.items():
        text = text.replace(contraction, expansion)

    # Conservation des points d'exclamation et d'interrogation car ils sont info
    text = re.sub(r'([!?])\1+', r'\1\1', text) # Réduire les répétitions à deux

    # Tokenisation simplifiée (découpage par espaces)
    tokens = text.split()

    # Reconstruction du texte
    processed_text = ' '.join(tokens)

    return processed_text

# Prétraitement des données
print("Prétraitement des textes...")
X_train_text = train_data['text'].apply(preprocess_text)
X_val_text = val_data['text'].apply(preprocess_text)
X_test_text = test_data['text'].apply(preprocess_text)

y_train = train_data['sentiment']
y_val = val_data['sentiment']
y_test = test_data['sentiment']

# Exemple de tweets prétraités
print("\nExemples de tweets avant et après prétraitement:")
for i in range(3):
    print(f"Original: {train_data['text'].iloc[i]}")
    print(f"Prétraité: {X_train_text.iloc[i]}")
    print("-" * 50)
```

```

# Définition du pipeline de vectorisation amélioré
# 1. TF-IDF avec n-grammes de 1 à 3
tfidf_vectorizer = TfidfVectorizer(
    max_features=30000,           # Plus de features pour capturer plus d'informations
    min_df=5,                   # Ignorer les termes qui apparaissent dans moins de 5 documents
    max_df=0.8,                 # Ignorer les termes qui apparaissent dans plus de 80% des documents
    ngram_range=(1, 3),          # Utiliser des unigrammes, bigrammes et trigrammes
    sublinear_tf=True,           # Appliquer une mise à l'échelle logarithmique (1+IDF)/(1+sqrt(IDF))
    use_idf=True,                # Utiliser l'IDF
    norm='l2',                   # Normalisation L2
    analyzer='word',             # Analyse basée sur les mots (alternative: 'char')
    token_pattern=r'\b\w+\b'      # Motif de tokenisation des mots
)

# 2. Bag of Words avec n-grammes de 1 à 2
count_vectorizer = CountVectorizer(
    max_features=30000,
    min_df=5,
    max_df=0.8,
    ngram_range=(1, 2),
    token_pattern=r'\b\w+\b'
)

# 3. Extracteur de caractéristiques textuelles
text_features = TextFeatureExtractor(with_text_stats=True, with_sentiment_words=True)

# Création des pipelines de features
tfidf_pipeline = Pipeline([
    ('tfidf', tfidf_vectorizer)
])

bow_pipeline = Pipeline([
    ('bow', count_vectorizer)
])

features_pipeline = Pipeline([
    ('features', text_features)
])

# Classe pour gérer les valeurs négatives (nécessaire pour MultinomialNB)
from sklearn.base import BaseEstimator, TransformerMixin

class NonNegativeTransformer(BaseEstimator, TransformerMixin):
    """
    Transforme les données pour s'assurer qu'elles sont non-négatives.
    """
    def __init__(self, offset=0):
        self.offset = offset
        self.min_value_ = None

    def fit(self, X, y=None):
        # Trouver la valeur minimale pour les données denses
        if hasattr(X, "toarray"):
            self.min_value_ = X.toarray().min()
        else:
            self.min_value_ = X.min()
        return self

    def transform(self, X):
        # Si la valeur minimale est négative, décaler toutes les valeurs

```

```

        if self.min_value_ is not None and self.min_value_ < 0:
            offset = abs(self.min_value_) + self.offset
            # Pour les matrices creuses, utiliser une méthode différente
            if hasattr(X, "toarray"):
                X_dense = X.toarray()
                X_dense += offset
                return X_dense
            else:
                return X + offset
        return X

# Combinaison des features via FeatureUnion
tfidf_features_union = FeatureUnion([
    ('tfidf_pipeline', tfidf_pipeline),
    ('features_pipeline', features_pipeline)
])

bow_features_union = FeatureUnion([
    ('bow_pipeline', bow_pipeline),
    ('features_pipeline', features_pipeline)
])

# Transformation avec TF-IDF + caractéristiques
tfidf_features_pipeline = Pipeline([
    ('union', tfidf_features_union)
])

# Transformation avec Bow + caractéristiques
bow_features_pipeline = Pipeline([
    ('union', bow_features_union)
])

# Pour MultinomialNB, on crée un pipeline spécial qui garantit des valeurs non-nulles
mnb_tfidf_pipeline = Pipeline([
    ('tfidf', tfidf_vectorizer) # Uniquement TF-IDF sans les features additionnelles
])

# Pipeline spécial pour MultinomialNB
mnb_pipeline = Pipeline([
    ('mnb_tfidf', mnb_tfidf_pipeline),
    ('classifier', MultinomialNB(alpha=0.1))
])

# Application des pipelines standards pour les autres modèles
print("Extraction des caractéristiques pour les modèles standards...")
# Extraction directe de TF-IDF pour MultinomialNB (sans features supplémentaires)
X_train_tfidf_simple = tfidf_vectorizer.fit_transform(X_train_text)
X_val_tfidf_simple = tfidf_vectorizer.transform(X_val_text)
X_test_tfidf_simple = tfidf_vectorizer.transform(X_test_text)

# Extraction des matrices Bow pour tous les modèles
X_train_bow_simple = count_vectorizer.fit_transform(X_train_text)
X_val_bow_simple = count_vectorizer.transform(X_val_text)
X_test_bow_simple = count_vectorizer.transform(X_test_text)

# Ajustement et transformation des données pour les pipelines riches (pour les autres modèles)
X_train_tfidf_features = tfidf_features_pipeline.fit_transform(X_train_text)
X_val_tfidf_features = tfidf_features_pipeline.transform(X_val_text)
X_test_tfidf_features = tfidf_features_pipeline.transform(X_test_text)

```

```
X_train_bow_features = bow_features_pipeline.fit_transform(X_train_text)
X_val_bow_features = bow_features_pipeline.transform(X_val_text)
X_test_bow_features = bow_features_pipeline.transform(X_test_text)

print("Extraction des caractéristiques terminée.")

# Affichage des dimensions des matrices de features
print(f"\nDimensions des matrices TF-IDF standards:")
print(f"Train: {X_train_tfidf_simple.shape}")
print(f"Validation: {X_val_tfidf_simple.shape}")

print(f"\nDimensions des matrices TF-IDF + caractéristiques:")
print(f"Train: {X_train_tfidf_features.shape}")
print(f"Validation: {X_val_tfidf_features.shape}")

print(f"\nDimensions des matrices Bag of Words + caractéristiques:")
print(f"Train: {X_train_bow_features.shape}")
print(f"Validation: {X_val_bow_features.shape}")
```

Prétraitement des textes...

Exemples de tweets avant et après prétraitement:

Original: thx quotgtlistenersthk hi à fjh

Prétraité: thx quotgtlistenersthk hi à fjh

-----  
Original: ergh miserable weather

Prétraité: ergh miserable weather

-----  
Original: apple inears slightly comfy slightly loose one ear thats multiple ear tip course

Prétraité: apple inears slightly comfy slightly loose one ear thats multiple ear tip course

-----  
Extraction des caractéristiques pour les modèles standards...

Extraction des caractéristiques terminée.

Dimensions des matrices TF-IDF standards:

Train: (960000, 30000)

Validation: (320000, 30000)

Dimensions des matrices TF-IDF + caractéristiques:

Train: (960000, 30012)

Validation: (320000, 30012)

Dimensions des matrices Bag of Words + caractéristiques:

Train: (960000, 30012)

Validation: (320000, 30012)

## 5. Entraînement et évaluation des modèles

```
In [7]: # Calcul des poids des classes pour gérer d'éventuels déséquilibres
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train
)
class_weight_dict = dict(zip(np.unique(y_train), class_weights))
print(f"Poids des classes: {class_weight_dict}")
```

```

# Définition d'une fonction pour évaluer les modèles avec MLflow
def evaluate_model(model, X_train, X_val, y_train, y_val, feature_type, model_name):
    """
    Entraine et évalue un modèle, puis enregistre les résultats avec MLflow
    """

    # Démarrer le suivi MLflow
    with mlflow.start_run(run_name=f"{model_name}_{feature_type}"):
        # Enregistrer les paramètres du modèle
        mlflow.log_param("model_type", model_name)
        mlflow.log_param("feature_type", feature_type)

        # Enregistrer les hyperparamètres du modèle si possible
        try:
            for param_name, param_value in model.get_params().items():
                if isinstance(param_value, (str, int, float, bool)):
                    mlflow.log_param(param_name, param_value)
        except:
            pass # Ignorer les erreurs possibles avec les paramètres complexes

        # Entraînement du modèle
        start_time = time.time()
        model.fit(X_train, y_train)
        training_time = time.time() - start_time

        # Prédictions sur les données de validation
        start_predict_time = time.time()
        y_pred = model.predict(X_val)
        prediction_time = time.time() - start_predict_time

        # Si le modèle peut donner des probabilités, les obtenir
        try:
            if hasattr(model, "predict_proba"):
                y_prob = model.predict_proba(X_val)[:, 1]
                roc_auc = roc_auc_score(y_val, y_prob)
            else:
                y_prob = None
                roc_auc = None
        except:
            y_prob = None
            roc_auc = None

        # Calcul des métriques
        accuracy = accuracy_score(y_val, y_pred)
        precision = precision_score(y_val, y_pred)
        recall = recall_score(y_val, y_pred)
        f1 = f1_score(y_val, y_pred)

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

        # Enregistrement des métriques avec MLflow
        mlflow.log_metric("accuracy", accuracy)
        mlflow.log_metric("precision", precision)
        mlflow.log_metric("recall", recall)
        mlflow.log_metric("f1", f1)
        if roc_auc is not None:
            mlflow.log_metric("roc_auc", roc_auc)
        mlflow.log_metric("training_time", training_time)
        mlflow.log_metric("prediction_time", prediction_time)

```

```

# Enregistrement du modèle
mlflow.sklearn.log_model(model, f"{model_name}_{feature_type}")

# Création et enregistrement de la courbe ROC si possible
if y_prob is not None:
    plt.figure(figsize=(8, 6))
    fpr, tpr, _ = roc_curve(y_val, y_prob)
    plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve - {model_name} with {feature_type}')
    plt.legend(loc="lower right")
    plt.tight_layout()

# Sauvegarde de La figure
roc_curve_path = f"roc_curve_{model_name}_{feature_type}.png"
plt.savefig(roc_curve_path)
mlflow.log_artifact(roc_curve_path)
os.remove(roc_curve_path) # Nettoyage
plt.close()

# Visualisation de La matrice de confusion
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Négatif', 'Positif'],
            yticklabels=['Négatif', 'Positif'])
plt.xlabel('Prédiction')
plt.ylabel('Réel')
plt.title(f'Matrice de confusion - {model_name} avec {feature_type}')
plt.tight_layout()

# Sauvegarde de la matrice de confusion
cm_path = f"confusion_matrix_{model_name}_{feature_type}.png"
plt.savefig(cm_path)
mlflow.log_artifact(cm_path)
os.remove(cm_path) # Nettoyage
plt.close()

# Affichage des résultats
print(f"\nRésultats pour {model_name} avec {feature_type}:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")
if roc_auc is not None:
    print(f"ROC AUC: {roc_auc:.4f}")
print(f"Temps d'entraînement: {training_time:.2f} secondes")
print(f"Temps de prédiction: {prediction_time:.2f} secondes")

# Retourner Les métriques pour comparaison ultérieure
return {
    "model": model,
    "model_name": model_name,
    "feature_type": feature_type,
    "accuracy": accuracy,
    "precision": precision,
    "recall": recall,
}

```

```

        "f1": f1,
        "roc_auc": roc_auc,
        "training_time": training_time,
        "prediction_time": prediction_time
    }

# Définition des modèles améliorés à tester
models = {
    "LogisticRegression": LogisticRegression(
        C=1.0,
        class_weight='balanced',
        max_iter=1000,
        solver='liblinear',
        random_state=42
    ),
    "LinearSVC": CalibratedClassifierCV(
        LinearSVC(
            C=1.0,
            class_weight='balanced',
            dual=True,
            max_iter=1000,
            random_state=42
        ),
        method='sigmoid'
    ),
    "SGDClassifier": SGDClassifier(
        loss='modified_huber', # Pour obtenir des probabilités
        penalty='l2',
        alpha=1e-4,
        max_iter=1000,
        tol=1e-3,
        class_weight='balanced',
        random_state=42
    ),
    "GradientBoosting": GradientBoostingClassifier(
        n_estimators=100,
        learning_rate=0.1,
        max_depth=5,
        random_state=42
    )
}

# Liste pour stocker les résultats
results = []

# Test avec TF-IDF + caractéristiques
print("Évaluation des modèles avec TF-IDF + caractéristiques...")
for model_name, model in models.items():
    result = evaluate_model(
        model,
        X_train_tfidf_features, X_val_tfidf_features,
        y_train, y_val,
        "TF-IDF_Features",
        model_name
    )
    results.append(result)

# Test spécial de MultinomialNB avec TF-IDF simple
print("\nÉvaluation de MultinomialNB avec TF-IDF simple...")
result = evaluate_model(

```

```

        MultinomialNB(alpha=0.1),
        X_train_tfidf_simple, X_val_tfidf_simple,
        y_train, y_val,
        "TF-IDF_Simple",
        "MultinomialNB"
    )
results.append(result)

# Test avec Bag of Words + caractéristiques
print("\nÉvaluation des modèles avec Bag of Words + caractéristiques...")
for model_name, model in models.items():
    result = evaluate_model(
        model,
        X_train_bow_features, X_val_bow_features,
        y_train, y_val,
        "BoW_Features",
        model_name
    )
    results.append(result)

# Test de MultinomialNB avec Bag of Words simple
print("\nÉvaluation de MultinomialNB avec Bag of Words simple...")
result = evaluate_model(
    MultinomialNB(alpha=0.1),
    X_train_bow_simple, X_val_bow_simple,
    y_train, y_val,
    "Bow_Simple",
    "MultinomialNB"
)
results.append(result)

# Création d'un DataFrame avec les résultats pour faciliter la comparaison
results_df = pd.DataFrame(results)
# Tri par F1-score décroissant
results_df = results_df.sort_values('f1', ascending=False)

print("\nComparaison des performances des modèles:")
display(results_df[['model_name', 'feature_type', 'accuracy', 'precision', 'reca

# Visualisation des résultats
plt.figure(figsize=(15, 10))

# Graphique pour le F1-score
plt.subplot(2, 2, 1)
sns.barplot(x='model_name', y='f1', hue='feature_type', data=results_df)
plt.title('F1-score par modèle et type de vectorisation')
plt.xlabel('Modèle')
plt.ylabel('F1-score')
plt.xticks(rotation=45)

# Graphique pour l'accuracy
plt.subplot(2, 2, 2)
sns.barplot(x='model_name', y='accuracy', hue='feature_type', data=results_df)
plt.title('Accuracy par modèle et type de vectorisation')
plt.xlabel('Modèle')
plt.ylabel('Accuracy')
plt.xticks(rotation=45)

# Graphique pour la précision
plt.subplot(2, 2, 3)

```

```

sns.barplot(x='model_name', y='precision', hue='feature_type', data=results_df)
plt.title('Precision par modèle et type de vectorisation')
plt.xlabel('Modèle')
plt.ylabel('Precision')
plt.xticks(rotation=45)

# Graphique pour le recall
plt.subplot(2, 2, 4)
sns.barplot(x='model_name', y='recall', hue='feature_type', data=results_df)
plt.title('Recall par modèle et type de vectorisation')
plt.xlabel('Modèle')
plt.ylabel('Recall')
plt.xticks(rotation=45)

plt.tight_layout()
plt.savefig("model_performance_comparison.png")
with mlflow.start_run(run_name="performance_comparison"):
    mlflow.log_artifact("model_performance_comparison.png")
plt.show()

```

Poids des classes: {0: 1.0, 1: 1.0}

Évaluation des modèles avec TF-IDF + caractéristiques...

2025/05/06 14:24:06 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour LogisticRegression avec TF-IDF\_Features:

Accuracy: 0.7907

Precision: 0.7789

Recall: 0.8119

F1-score: 0.7951

ROC AUC: 0.8724

Temps d'entraînement: 13.15 secondes

Temps de prédiction: 0.01 secondes

2025/05/06 14:36:56 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour LinearSVC avec TF-IDF\_Features:

Accuracy: 0.7837

Precision: 0.7748

Recall: 0.7998

F1-score: 0.7871

ROC AUC: 0.8648

Temps d'entraînement: 767.02 secondes

Temps de prédiction: 0.15 secondes

2025/05/06 14:37:28 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour SGDClassifier avec TF-IDF\_Features:

Accuracy: 0.7080

Precision: 0.8678

Recall: 0.4908

F1-score: 0.6270

ROC AUC: 0.8359

Temps d'entraînement: 29.27 secondes

Temps de prédiction: 0.01 secondes

2025/05/06 14:48:03 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour GradientBoosting avec TF-IDF\_Features:  
Accuracy: 0.7192  
Precision: 0.6789  
Recall: 0.8319  
F1-score: 0.7476  
ROC AUC: 0.7911  
Temps d'entraînement: 632.08 secondes  
Temps de prédiction: 0.41 secondes

Évaluation de MultinomialNB avec TF-IDF simple...

2025/05/06 14:48:06 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour MultinomialNB avec TF-IDF\_Simple:  
Accuracy: 0.7721  
Precision: 0.7728  
Recall: 0.7708  
F1-score: 0.7718  
ROC AUC: 0.8548  
Temps d'entraînement: 0.11 secondes  
Temps de prédiction: 0.02 secondes

Évaluation des modèles avec Bag of Words + caractéristiques...

2025/05/06 14:48:43 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour LogisticRegression avec BoW\_Features:  
Accuracy: 0.7886  
Precision: 0.7732  
Recall: 0.8167  
F1-score: 0.7944  
ROC AUC: 0.8648  
Temps d'entraînement: 33.79 secondes  
Temps de prédiction: 0.02 secondes

2025/05/06 15:00:57 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour LinearSVC avec BoW\_Features:  
Accuracy: 0.7861  
Precision: 0.7716  
Recall: 0.8130  
F1-score: 0.7917  
ROC AUC: 0.8604  
Temps d'entraînement: 731.45 secondes  
Temps de prédiction: 0.11 secondes

2025/05/06 15:01:24 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour SGDClassifier avec BoW\_Features:  
Accuracy: 0.7459  
Precision: 0.8621  
Recall: 0.5856  
F1-score: 0.6974  
ROC AUC: 0.8492  
Temps d'entraînement: 24.74 secondes  
Temps de prédiction: 0.01 secondes

2025/05/06 15:05:21 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour GradientBoosting avec Bow\_Features:

Accuracy: 0.7191

Precision: 0.6789

Recall: 0.8314

F1-score: 0.7474

ROC AUC: 0.7908

Temps d'entraînement: 233.45 secondes

Temps de prédiction: 0.34 secondes

Évaluation de MultinomialNB avec Bag of Words simple...

2025/05/06 15:05:24 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats pour MultinomialNB avec Bow\_Simple:

Accuracy: 0.7741

Precision: 0.7765

Recall: 0.7697

F1-score: 0.7731

ROC AUC: 0.8471

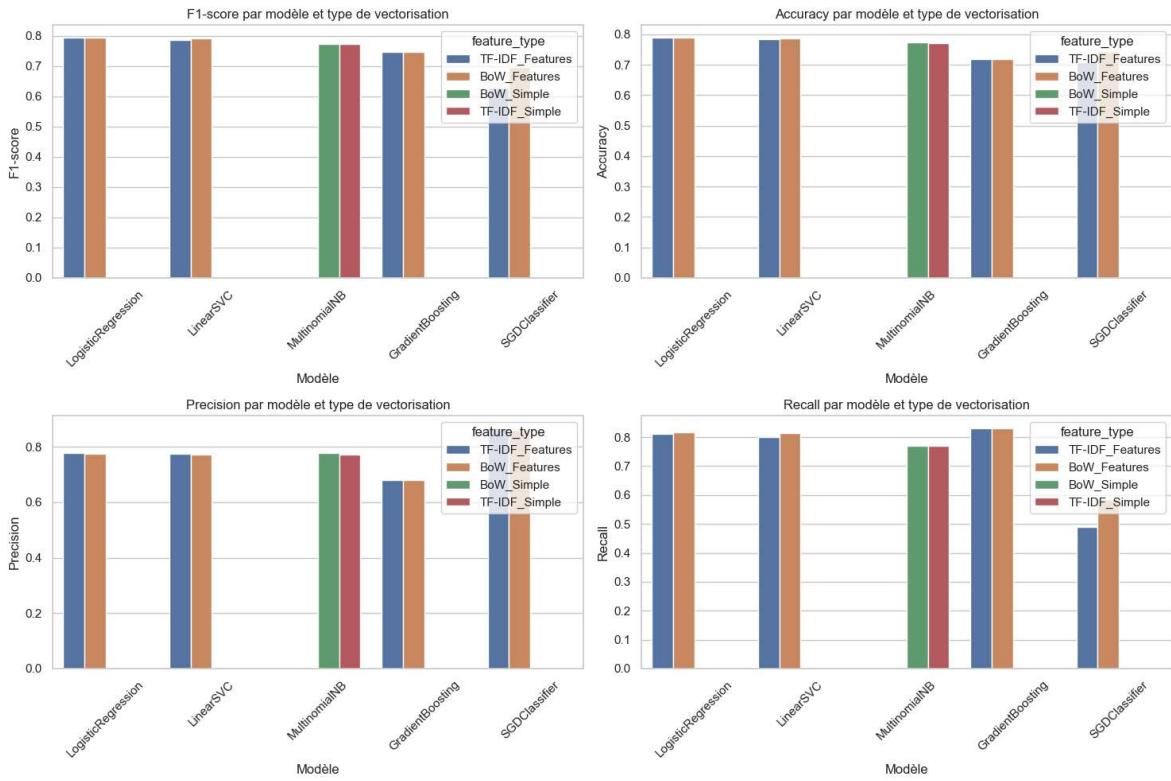
Temps d'entraînement: 0.09 secondes

Temps de prédiction: 0.02 secondes

Comparaison des performances des modèles:

	model_name	feature_type	accuracy	precision	recall	f1	roc_auc	trai
0	LogisticRegression	TF-IDF_Features	0.790728	0.778921	0.811894	0.795066	0.872385	7
5	LogisticRegression	BoW_Features	0.788616	0.773241	0.816750	0.794400	0.864775	7
6	LinearSVC	BoW_Features	0.786134	0.771550	0.812987	0.791727	0.860393	7
1	LinearSVC	TF-IDF_Features	0.783669	0.774826	0.799756	0.787094	0.864760	7
9	MultinomialNB	BoW_Simple	0.774088	0.776517	0.769694	0.773091	0.847145	6
4	MultinomialNB	TF-IDF_Simple	0.772053	0.772754	0.770769	0.771760	0.854773	6
3	GradientBoosting	TF-IDF_Features	0.719216	0.678910	0.831856	0.747641	0.791058	6
8	GradientBoosting	BoW_Features	0.719056	0.678860	0.831425	0.747436	0.790799	2
7	SGDClassifier	BoW_Features	0.745947	0.862103	0.585556	0.697415	0.849229	2
2	SGDClassifier	TF-IDF_Features	0.708019	0.867841	0.490775	0.626983	0.835862	2





## 6. Création d'ensembles de modèles

```
In [8]: # Identification des meilleurs modèles pour chaque type de feature
print("Préparation pour la création d'ensembles...")
best_models_info = results_df.head(4) # Top 4 des meilleurs modèles individuels

print("Meilleurs modèles individuels:")
display(best_models_info[['model_name', 'feature_type', 'f1', 'precision', 'recall']])

# Création des pipelines complets pour les ensembles
ensemble_estimators = []

for idx, row in best_models_info.iterrows():
    model_name = row['model_name']
    feature_type = row['feature_type']
    model = row['model']

    # Création des transformations selon le type de feature
    if feature_type == 'TF-IDF_Features':
        feature_pipeline = tfidf_features_pipeline
        ensemble_name = f"{model_name}_TF-IDF_Features"
    elif feature_type == 'TF-IDF_Simple':
        feature_pipeline = Pipeline([('tfidf', tfidf_vectorizer)])
        ensemble_name = f"{model_name}_TF-IDF_Simple"
    elif feature_type == 'BoW_Features':
        feature_pipeline = bow_features_pipeline
        ensemble_name = f"{model_name}_BoW_Features"
    else: # BoW_Simple
        feature_pipeline = Pipeline([('bow', count_vectorizer)])
        ensemble_name = f"{model_name}_BoW_Simple"

    # Création du pipeline complet (transformation + modèle)
    pipeline = Pipeline([
        ('features', feature_pipeline),
        ('model', model)
    ])
    ensemble_estimators.append(pipeline)
```

```

        ('model', model)
    ])

ensemble_estimators.append((ensemble_name, pipeline))

# Cr ation d'un VotingClassifier (ensemble par vote)
print("\nCr ation d'un ensemble de mod les par vote...")
voting_classifier = VotingClassifier(
    estimators=ensemble_estimators,
    voting='soft', # Utilisation des probabilit s pour le vote
    n_jobs=1 # viter les probl mes de parall lisation
)

# valuation de l'ensemble par vote
print("valuation de l'ensemble par vote...")
voting_result = evaluate_model(
    voting_classifier,
    X_train_text, X_val_text, # Utiliser du texte brut, le pipeline s'occupe de
    y_train, y_val,
    "Ensemble_Vote",
    "VotingClassifier"
)
results.append(voting_result)

# Cr ation d'un StackingClassifier (ensemble par empilement)
print("\nCr ation d'un ensemble de mod les par empilement...")
stacking_classifier = StackingClassifier(
    estimators=ensemble_estimators,
    final_estimator=LogisticRegression(C=1.0, class_weight='balanced', max_iter=
    cv=3, # R duire le nombre de folds pour la validation crois e
    n_jobs=1 # viter les probl mes de parall lisation
)

# valuation de l'ensemble par empilement
print("valuation de l'ensemble par empilement...")
stacking_result = evaluate_model(
    stacking_classifier,
    X_train_text, X_val_text, # Utiliser du texte brut, le pipeline s'occupe de
    y_train, y_val,
    "Ensemble_Stack",
    "StackingClassifier"
)
results.append(stacking_result)

# Mise  jour du DataFrame de r sultats
results_df = pd.DataFrame(results)
results_df = results_df.sort_values('f1', ascending=False)

print("\nComparaison des performances avec les ensembles de mod les:")
display(results_df[['model_name', 'feature_type', 'accuracy', 'precision', 'reca

```

Préparation pour la création d'ensembles...

Meilleurs modèles individuels:

<b>model_name</b>	<b>feature_type</b>	<b>f1</b>	<b>precision</b>	<b>recall</b>	
<b>0</b> LogisticRegression	TF-IDF_Features	0.795066	0.778921	0.811894	
<b>5</b> LogisticRegression	BoW_Features	0.794400	0.773241	0.816750	
<b>6</b>	LinearSVC	BoW_Features	0.791727	0.771550	0.812987
<b>1</b>	LinearSVC	TF-IDF_Features	0.787094	0.774826	0.799756

Création d'un ensemble de modèles par vote...

Évaluation de l'ensemble par vote...

```
2025/05/06 15:32:08 WARNING mlflow.models.model: Model logged without a signature
and input example. Please set `input_example` parameter when logging the model to
auto infer the model signature.
```

Résultats pour VotingClassifier avec Ensemble\_Vote:

Accuracy: 0.7903

Precision: 0.7766

Recall: 0.8150

F1-score: 0.7954

ROC AUC: 0.8703

Temps d'entraînement: 1566.14 secondes

Temps de prédiction: 17.04 secondes

Création d'un ensemble de modèles par empilement...

Évaluation de l'ensemble par empilement...

```
2025/05/06 16:45:58 WARNING mlflow.models.model: Model logged without a signature
and input example. Please set `input_example` parameter when logging the model to
auto infer the model signature.
```

Résultats pour StackingClassifier avec Ensemble\_Stack:

Accuracy: 0.7909

Precision: 0.7812

Recall: 0.8082

F1-score: 0.7945

ROC AUC: 0.8724

Temps d'entraînement: 4393.06 secondes

Temps de prédiction: 16.95 secondes

Comparaison des performances avec les ensembles de modèles:

	<b>model_name</b>	<b>feature_type</b>	<b>accuracy</b>	<b>precision</b>	<b>recall</b>	<b>f1</b>	<b>roc_auc</b>
<b>10</b>	VotingClassifier	Ensemble_Vote	0.790316	0.776634	0.815044	0.795376	0.870336
<b>0</b>	LogisticRegression	TF-IDF_Features	0.790728	0.778921	0.811894	0.795066	0.872385
<b>11</b>	StackingClassifier	Ensemble_Stack	0.790906	0.781194	0.808175	0.794456	0.872433
<b>5</b>	LogisticRegression	BoW_Features	0.788616	0.773241	0.816750	0.794400	0.864775
<b>6</b>	LinearSVC	BoW_Features	0.786134	0.771550	0.812987	0.791727	0.860393
<b>1</b>	LinearSVC	TF-IDF_Features	0.783669	0.774826	0.799756	0.787094	0.864760
<b>9</b>	MultinomialNB	BoW_Simple	0.774088	0.776517	0.769694	0.773091	0.847145
<b>4</b>	MultinomialNB	TF-IDF_Simple	0.772053	0.772754	0.770769	0.771760	0.854773
<b>3</b>	GradientBoosting	TF-IDF_Features	0.719216	0.678910	0.831856	0.747641	0.791058
<b>8</b>	GradientBoosting	BoW_Features	0.719056	0.678860	0.831425	0.747436	0.790799

## 7. Optimisation fine des hyperparamètres

Nous allons maintenant optimiser les hyperparamètres du meilleur modèle individuel identifié précédemment.

```
In [9]: # Identification du meilleur modèle individuel (excluant les ensembles)
individual_models_df = results_df[~results_df['model_name'].isin(['VotingClassifier', 'StackingClassifier'])]
best_individual_model_info = individual_models_df.iloc[0]

print(f"Meilleur modèle individuel selon le F1-score:")
print(f"- Modèle: {best_individual_model_info['model_name']}")
print(f"- Features: {best_individual_model_info['feature_type']}")
print(f"- F1-score: {best_individual_model_info['f1']:.4f}")

# Sélection des features appropriées
if 'TF-IDF_Features' in best_individual_model_info['feature_type']:
    X_train_selected = X_train_tfidf_features
    X_val_selected = X_val_tfidf_features
    X_test_selected = X_test_tfidf_features
    feature_type = 'TF-IDF_Features'
elif 'TF-IDF_Simple' in best_individual_model_info['feature_type']:
    X_train_selected = X_train_tfidf_simple
    X_val_selected = X_val_tfidf_simple
    X_test_selected = X_test_tfidf_simple
    feature_type = 'TF-IDF_Simple'
elif 'BoW_Features' in best_individual_model_info['feature_type']:
    X_train_selected = X_train_bow_features
    X_val_selected = X_val_bow_features
    X_test_selected = X_test_bow_features
    feature_type = 'BoW_Features'
else: # Bow_Simple
    X_train_selected = X_train_bow_simple
    X_val_selected = X_val_bow_simple
    X_test_selected = X_test_bow_simple
```

```

feature_type = 'Bow_Simple'

# Définition des paramètres ciblés à optimiser selon le type de modèle
if best_individual_model_info['model_name'] == 'LogisticRegression':
    # Pour LogisticRegression, se concentrer sur C et class_weight
    param_grid = {
        'C': [0.5, 1.0, 2.0, 5.0],
        'class_weight': ['balanced', None]
    }
    base_model = LogisticRegression(penalty='l2', solver='liblinear', max_iter=100)

elif best_individual_model_info['model_name'] == 'MultinomialNB':
    # Pour MultinomialNB, uniquement alpha
    param_grid = {
        'alpha': [0.1, 0.3, 0.5, 0.7]
    }
    base_model = MultinomialNB()

elif best_individual_model_info['model_name'] == 'LinearSVC':
    # Pour LinearSVC, uniquement C et class_weight
    param_grid = {
        'base_estimator__C': [0.5, 1.0, 2.0],
        'base_estimator__class_weight': ['balanced', None]
    }
    base_model = CalibratedClassifierCV(
        base_estimator=LinearSVC(random_state=42, max_iter=1000, dual=True),
        method='sigmoid'
    )

elif best_individual_model_info['model_name'] == 'SGDClassifier':
    # Pour SGDClassifier, se concentrer sur alpha et loss
    param_grid = {
        'alpha': [1e-4, 1e-3],
        'loss': ['hinge', 'modified_huber']
    }
    base_model = SGDClassifier(max_iter=1000, tol=1e-3, class_weight='balanced', loss='hinge')

else: # GradientBoosting
    # Pour GradientBoosting, uniquement n_estimators et learning_rate
    param_grid = {
        'n_estimators': [100, 200],
        'learning_rate': [0.05, 0.1]
    }
    base_model = GradientBoostingClassifier(max_depth=5, random_state=42)

# Validation croisée réduite pour accélérer l'optimisation
cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Utiliser GridSearchCV avec paramètres ciblés
print("\nOptimisation ciblée des hyperparamètres...")
grid_search = GridSearchCV(
    base_model,
    param_grid=param_grid,
    cv=cv,
    scoring='f1',
    n_jobs=-1,
    verbose=1
)

# Entraînement avec MLflow tracking

```

```

with mlflow.start_run(run_name=f"{best_individual_model_info['model_name']}_{feature_type}_opt"
    # Enregistrement des paramètres
    mlflow.log_param("model_type", best_individual_model_info['model_name'])
    mlflow.log_param("feature_type", feature_type)
    mlflow.log_param("optimization", "GridSearchCV_Targeted")

    # Entraînement
    start_time = time.time()
    grid_search.fit(X_train_selected, y_train)
    training_time = time.time() - start_time

    # Meilleurs paramètres
    best_params = grid_search.best_params_
    for param, value in best_params.items():
        if isinstance(value, (str, int, float, bool)):
            mlflow.log_param(param, value)

    # Meilleur modèle
    best_model = grid_search.best_estimator_

    # Prédictions
    y_val_pred = best_model.predict(X_val_selected)

    # Si le modèle peut donner des probabilités, les obtenir
    try:
        if hasattr(best_model, "predict_proba"):
            y_val_prob = best_model.predict_proba(X_val_selected)[:, 1]
            roc_auc = roc_auc_score(y_val, y_val_prob)
        else:
            y_val_prob = None
            roc_auc = None
    except:
        y_val_prob = None
        roc_auc = None

    # Métriques
    val_accuracy = accuracy_score(y_val, y_val_pred)
    val_precision = precision_score(y_val, y_val_pred)
    val_recall = recall_score(y_val, y_val_pred)
    val_f1 = f1_score(y_val, y_val_pred)

    # Log des métriques
    mlflow.log_metric("accuracy", val_accuracy)
    mlflow.log_metric("precision", val_precision)
    mlflow.log_metric("recall", val_recall)
    mlflow.log_metric("f1", val_f1)
    if roc_auc is not None:
        mlflow.log_metric("roc_auc", roc_auc)
    mlflow.log_metric("training_time", training_time)

    # Enregistrement du modèle
    model_name = f"{best_individual_model_info['model_name']}_{feature_type}_opt"
    mlflow.sklearn.log_model(best_model, model_name)

    # Visualisation de la matrice de confusion
    cm = confusion_matrix(y_val, y_val_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Négatif', 'Positif'],
                yticklabels=['Négatif', 'Positif'])

```

```

plt.xlabel('Prédiction')
plt.ylabel('Réel')
plt.title(f'Matrice de confusion - Modèle optimisé')
plt.tight_layout()

# Sauvegarde de la matrice de confusion
cm_path = "confusion_matrix_optimized.png"
plt.savefig(cm_path)
mlflow.log_artifact(cm_path)
os.remove(cm_path) # Nettoyage
plt.close()

# Affichage des résultats
print("\nRésultats de l'optimisation ciblée:")
print(f"Meilleurs paramètres: {best_params}")
print(f"Meilleur score de validation croisée: {grid_search.best_score_:.4f}")
print("\nPerformance sur le jeu de validation:")
print(f"Accuracy: {val_accuracy:.4f}")
print(f"Precision: {val_precision:.4f}")
print(f"Recall: {val_recall:.4f}")
print(f"F1-score: {val_f1:.4f}")
if roc_auc is not None:
    print(f"ROC AUC: {roc_auc:.4f}")
print(f"Temps d'entraînement: {training_time:.2f} secondes")

# Comparaison avec le modèle avant optimisation
original_f1 = best_individual_model_info['f1']
improvement = ((val_f1 - original_f1) / original_f1) * 100

print(f"\nComparaison avec le modèle non optimisé:")
print(f"F1-score avant optimisation: {original_f1:.4f}")
print(f"F1-score après optimisation: {val_f1:.4f}")
print(f"Amélioration: {improvement:.2f}%")

# Ajout du modèle optimisé aux résultats pour la comparaison
optimized_result = {
    "model": best_model,
    "model_name": f"{best_individual_model_info['model_name']}_Optimized",
    "feature_type": feature_type,
    "accuracy": val_accuracy,
    "precision": val_precision,
    "recall": val_recall,
    "f1": val_f1,
    "roc_auc": roc_auc,
    "training_time": training_time,
    "prediction_time": 0 # Sera mesuré plus tard si nécessaire
}
results.append(optimized_result)

# Mise à jour du DataFrame de résultats
results_df = pd.DataFrame(results)
results_df = results_df.sort_values('f1', ascending=False)

print("\nNouveau classement des modèles après optimisation:")
display(results_df[['model_name', 'feature_type', 'accuracy', 'precision', 'reca

```

Meilleur modèle individuel selon le F1-score:

- Modèle: LogisticRegression
- Features: TF-IDF\_Features
- F1-score: 0.7951

Optimisation ciblée des hyperparamètres...

Fitting 3 folds for each of 8 candidates, totalling 24 fits

2025/05/06 16:47:08 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input\_example` parameter when logging the model to auto infer the model signature.

Résultats de l'optimisation ciblée:

Meilleurs paramètres: {'C': 1.0, 'class\_weight': 'balanced'}

Meilleur score de validation croisée: 0.7933

Performance sur le jeu de validation:

Accuracy: 0.7907

Precision: 0.7789

Recall: 0.8119

F1-score: 0.7951

ROC AUC: 0.8724

Temps d'entraînement: 67.32 secondes

Comparaison avec le modèle non optimisé:

F1-score avant optimisation: 0.7951

F1-score après optimisation: 0.7951

Amélioration: 0.00%

Nouveau classement des modèles après optimisation:

	model_name	feature_type	accuracy	precision	recall	f1	
10	VotingClassifier	Ensemble_Vote	0.790316	0.776634	0.815044	0.795376	(
0	LogisticRegression	TF-IDF_Features	0.790728	0.778921	0.811894	0.795066	(
12	LogisticRegression_Optimized	TF-IDF_Features	0.790728	0.778921	0.811894	0.795066	(
11	StackingClassifier	Ensemble_Stack	0.790906	0.781194	0.808175	0.794456	(
5	LogisticRegression	BoW_Features	0.788616	0.773241	0.816750	0.794400	(
6	LinearSVC	BoW_Features	0.786134	0.771550	0.812987	0.791727	(
1	LinearSVC	TF-IDF_Features	0.783669	0.774826	0.799756	0.787094	(
9	MultinomialNB	BoW_Simple	0.774088	0.776517	0.769694	0.773091	(
4	MultinomialNB	TF-IDF_Simple	0.772053	0.772754	0.770769	0.771760	(
3	GradientBoosting	TF-IDF_Features	0.719216	0.678910	0.831856	0.747641	(

## 8. Sélection et évaluation finale des meilleurs modèles

```
In [11]: # Identification des meilleurs modèles (ensemble et individuel optimisé)
best_overall = results_df.iloc[0]
best_optimized = results_df[results_df['model_name'].str.contains('Optimized')].reset_index()

print(f"Meilleur modèle global selon le F1-score: {best_overall['model_name']} avec {best_overall['f1']:.4f}")
if best_optimized is not None:
    print(f"Meilleur modèle optimisé: {best_optimized['model_name']} avec {best_optimized['f1']:.4f}")

# Évaluation du modèle optimisé sur le jeu de test
print("\nÉvaluation du modèle optimisé sur le jeu de test...")
optimized_model_test_pred = best_model.predict(X_test_selected)

# Si le modèle peut donner des probabilités, les obtenir
try:
    if hasattr(best_model, "predict_proba"):
        optimized_model_test_prob = best_model.predict_proba(X_test_selected)[:, 1]
        optimized_model_test_roc_auc = roc_auc_score(y_test, optimized_model_test_prob)
    else:
        optimized_model_test_prob = None
        optimized_model_test_roc_auc = None
except:
    optimized_model_test_prob = None
    optimized_model_test_roc_auc = None

# Métriques
optimized_model_test_accuracy = accuracy_score(y_test, optimized_model_test_pred)
optimized_model_test_precision = precision_score(y_test, optimized_model_test_pred)
optimized_model_test_recall = recall_score(y_test, optimized_model_test_pred)
optimized_model_test_f1 = f1_score(y_test, optimized_model_test_pred)

# Matrice de confusion
optimized_model_test_cm = confusion_matrix(y_test, optimized_model_test_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(optimized_model_test_cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Négatif', 'Positif'],
            yticklabels=['Négatif', 'Positif'])
plt.xlabel('Prédiction')
plt.ylabel('Réel')
plt.title('Matrice de confusion du modèle optimisé sur le jeu de test')
plt.tight_layout()
plt.savefig("optimized_model_test_confusion_matrix.png")
plt.show()

print(f"Performance du modèle optimisé sur le jeu de test:")
print(f"Accuracy: {optimized_model_test_accuracy:.4f}")
print(f"Precision: {optimized_model_test_precision:.4f}")
print(f"Recall: {optimized_model_test_recall:.4f}")
print(f"F1-score: {optimized_model_test_f1:.4f}")
if optimized_model_test_roc_auc is not None:
    print(f"ROC AUC: {optimized_model_test_roc_auc:.4f}")

# Évaluation du meilleur ensemble sur le jeu de test (si applicable)
if best_overall['model_name'] in ['VotingClassifier', 'StackingClassifier']:
    print("\nÉvaluation du meilleur ensemble sur le jeu de test...")
    ensemble_model = best_overall['model']

    # Création d'un pipeline pour le test afin d'éviter les erreurs
    ensemble_test_pred = ensemble_model.predict(X_test_text)
```

```

# Si le modèle peut donner des probabilités, les obtenir
try:
    if hasattr(ensemble_model, "predict_proba"):
        ensemble_test_prob = ensemble_model.predict_proba(X_test_text)[:, 1]
        ensemble_test_roc_auc = roc_auc_score(y_test, ensemble_test_prob)
    else:
        ensemble_test_prob = None
        ensemble_test_roc_auc = None
except:
    ensemble_test_prob = None
    ensemble_test_roc_auc = None

# Métriques
ensemble_test_accuracy = accuracy_score(y_test, ensemble_test_pred)
ensemble_test_precision = precision_score(y_test, ensemble_test_pred)
ensemble_test_recall = recall_score(y_test, ensemble_test_pred)
ensemble_test_f1 = f1_score(y_test, ensemble_test_pred)

# Matrice de confusion
ensemble_test_cm = confusion_matrix(y_test, ensemble_test_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(ensemble_test_cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Négatif', 'Positif'],
            yticklabels=['Négatif', 'Positif'])
plt.xlabel('Prédiction')
plt.ylabel('Réel')
plt.title('Matrice de confusion de l\'ensemble sur le jeu de test')
plt.tight_layout()
plt.savefig("ensemble_test_confusion_matrix.png")
plt.show()

print(f"Performance de l'ensemble sur le jeu de test:")
print(f"Accuracy: {ensemble_test_accuracy:.4f}")
print(f"Precision: {ensemble_test_precision:.4f}")
print(f"Recall: {ensemble_test_recall:.4f}")
print(f"F1-score: {ensemble_test_f1:.4f}")
if ensemble_test_roc_auc is not None:
    print(f"ROC AUC: {ensemble_test_roc_auc:.4f}")

# Rapport de classification détaillé pour le meilleur modèle global
print("\nRapport de classification détaillé pour le meilleur modèle:")
if best_overall['model_name'] in ['VotingClassifier', 'StackingClassifier'] and
   best_test_pred = ensemble_test_pred
else:
    best_test_pred = optimized_model_test_pred

print(classification_report(y_test, best_test_pred, target_names=['Négatif', 'Po'])

# Enregistrement des résultats finaux dans MLflow
with mlflow.start_run(run_name="final_evaluation"):
    mlflow.log_metric("test_accuracy", optimized_model_test_accuracy)
    mlflow.log_metric("test_precision", optimized_model_test_precision)
    mlflow.log_metric("test_recall", optimized_model_test_recall)
    mlflow.log_metric("test_f1", optimized_model_test_f1)
    if optimized_model_test_roc_auc is not None:
        mlflow.log_metric("test_roc_auc", optimized_model_test_roc_auc)
    mlflow.log_artifact("optimized_model_test_confusion_matrix.png")

    if best_overall['model_name'] in ['VotingClassifier', 'StackingClassifier']
        mlflow.log_metric("ensemble_test_accuracy", ensemble_test_accuracy)

```

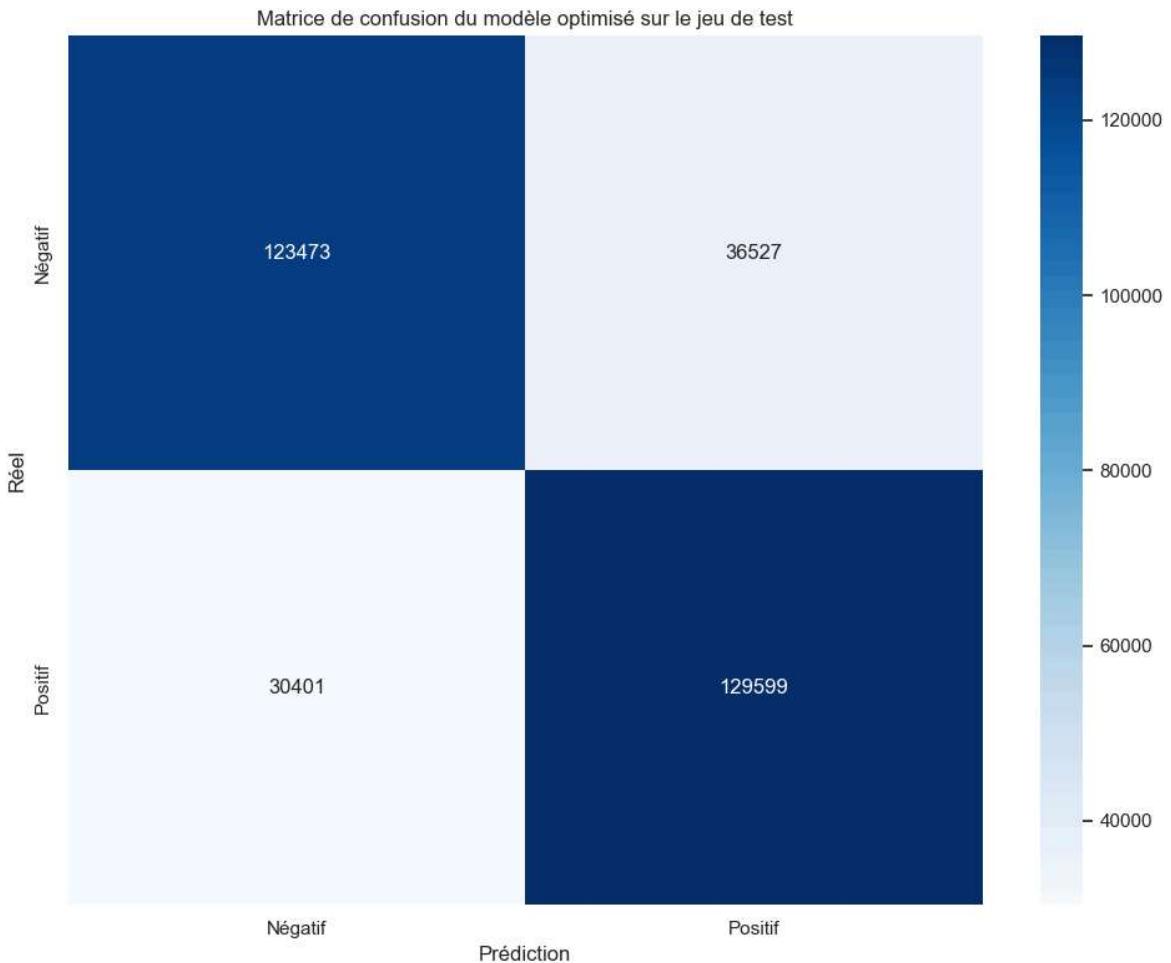
```

mlflow.log_metric("ensemble_test_precision", ensemble_test_precision)
mlflow.log_metric("ensemble_test_recall", ensemble_test_recall)
mlflow.log_metric("ensemble_test_f1", ensemble_test_f1)
if ensemble_test_roc_auc is not None:
    mlflow.log_metric("ensemble_test_roc_auc", ensemble_test_roc_auc)
mlflow.log_artifact("ensemble_test_confusion_matrix.png")

```

Meilleur modèle global selon le F1-score: VotingClassifier avec Ensemble\_Vote  
Meilleur modèle optimisé: LogisticRegression\_Optimized avec TF-IDF\_Features

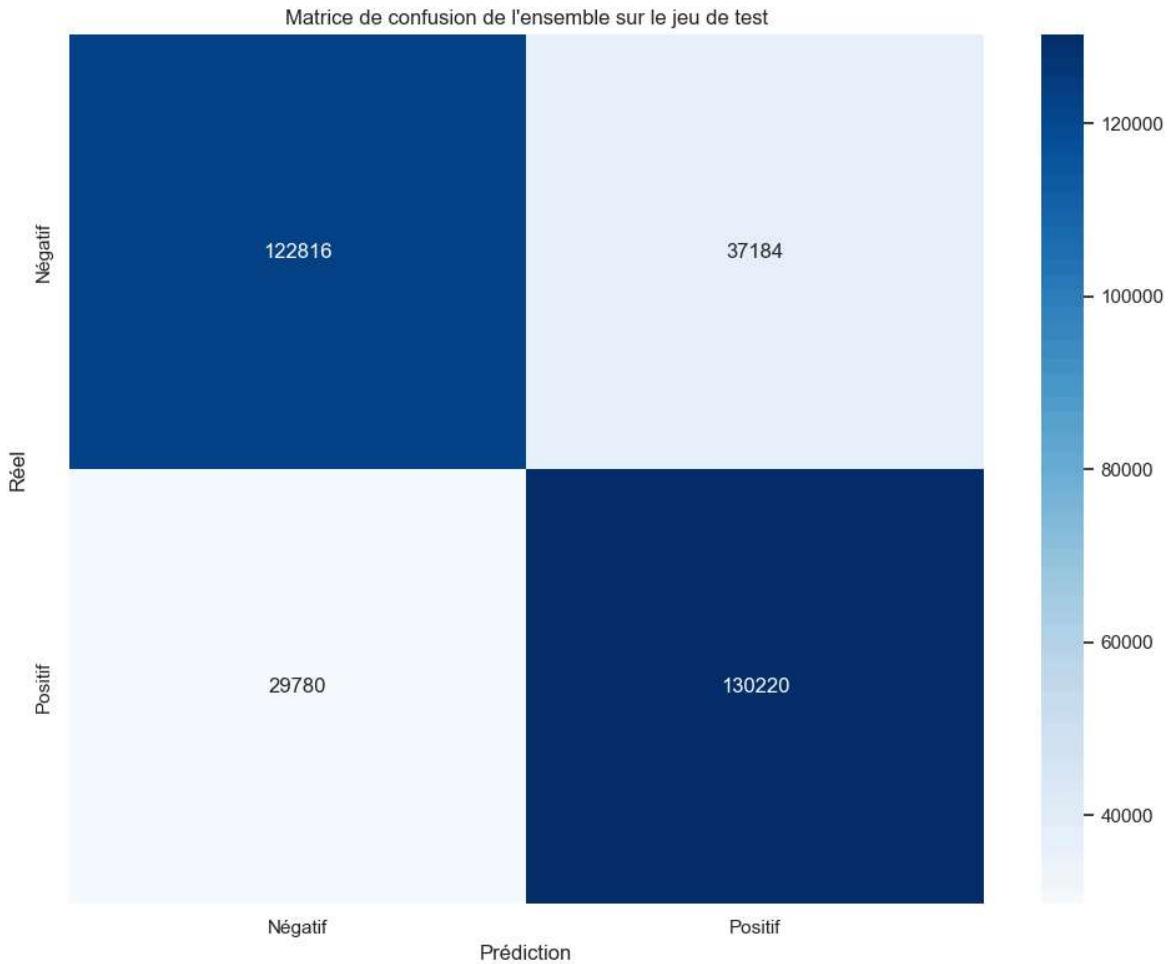
Évaluation du modèle optimisé sur le jeu de test...



Performance du modèle optimisé sur le jeu de test:

Accuracy: 0.7909  
Precision: 0.7801  
Recall: 0.8100  
F1-score: 0.7948  
ROC AUC: 0.8724

Évaluation du meilleur ensemble sur le jeu de test...



Performance de l'ensemble sur le jeu de test:

Accuracy: 0.7907

Precision: 0.7779

Recall: 0.8139

F1-score: 0.7955

ROC AUC: 0.8707

Rapport de classification détaillé pour le meilleur modèle:

	precision	recall	f1-score	support
Négatif	0.80	0.77	0.79	160000
Positif	0.78	0.81	0.80	160000
accuracy			0.79	320000
macro avg	0.79	0.79	0.79	320000
weighted avg	0.79	0.79	0.79	320000

## 9. Analyse des erreurs

```
In [12]: # Identifier les tweets mal classés par le meilleur modèle
if best_overall['model_name'] in ['VotingClassifier', 'StackingClassifier'] and
    best_model_final = best_overall['model']
    y_test_pred_final = ensemble_test_pred
else:
    best_model_final = best_model
    y_test_pred_final = optimized_model_test_pred

misclassified_indices = np.where(y_test != y_test_pred_final)[0]
print(f"Nombre de tweets mal classés: {len(misclassified_indices)} sur {len(y_te}
```

```

# Création d'un DataFrame pour l'analyse des erreurs
if len(misclassified_indices) > 0:
    misclassified_df = pd.DataFrame({
        'text': test_data['text'].iloc[misclassified_indices].values,
        'processed_text': [preprocess_text(test_data['text'].iloc[i]) for i in misclassified_indices],
        'true_sentiment': y_test[misclassified_indices],
        'predicted_sentiment': y_test_pred_final[misclassified_indices]
    })

# Conversion des valeurs numériques en étiquettes
misclassified_df['true_sentiment'] = misclassified_df['true_sentiment'].map(
    {0: 'Négatif', 1: 'Positif'})
misclassified_df['predicted_sentiment'] = misclassified_df['predicted_sentiment'].map(
    {0: 'Négatif', 1: 'Positif'})

# Ajout de la Longueur du texte
misclassified_df['text_length'] = misclassified_df['text'].apply(len)

# Analyse des erreurs par Longueur de texte
plt.figure(figsize=(10, 6))
sns.histplot(data=misclassified_df, x='text_length', hue='true_sentiment', bins=20)
plt.title('Distribution des longueurs de tweets mal classés par sentiment')
plt.xlabel('Longueur (caractères)')
plt.ylabel('Nombre de tweets')
plt.savefig("misclassified_length_distribution.png")
plt.show()

# Analyse des faux positifs et faux négatifs
fp_df = misclassified_df[(misclassified_df['true_sentiment'] == 'Négatif') &
                           (misclassified_df['predicted_sentiment'] == 'Positif')]
fn_df = misclassified_df[(misclassified_df['true_sentiment'] == 'Positif') &
                           (misclassified_df['predicted_sentiment'] == 'Négatif')]

print(f"Nombre de faux positifs (prédits positifs mais réellement négatifs): {len(fp_df)}")
print(f"Nombre de faux négatifs (prédits négatifs mais réellement positifs): {len(fn_df)}")

# Sélectionner un échantillon aléatoire de tweets mal classés pour analyse
sample_size = min(10, len(misclassified_indices))
sample_indices = np.random.choice(len(misclassified_df), size=sample_size, replace=False)

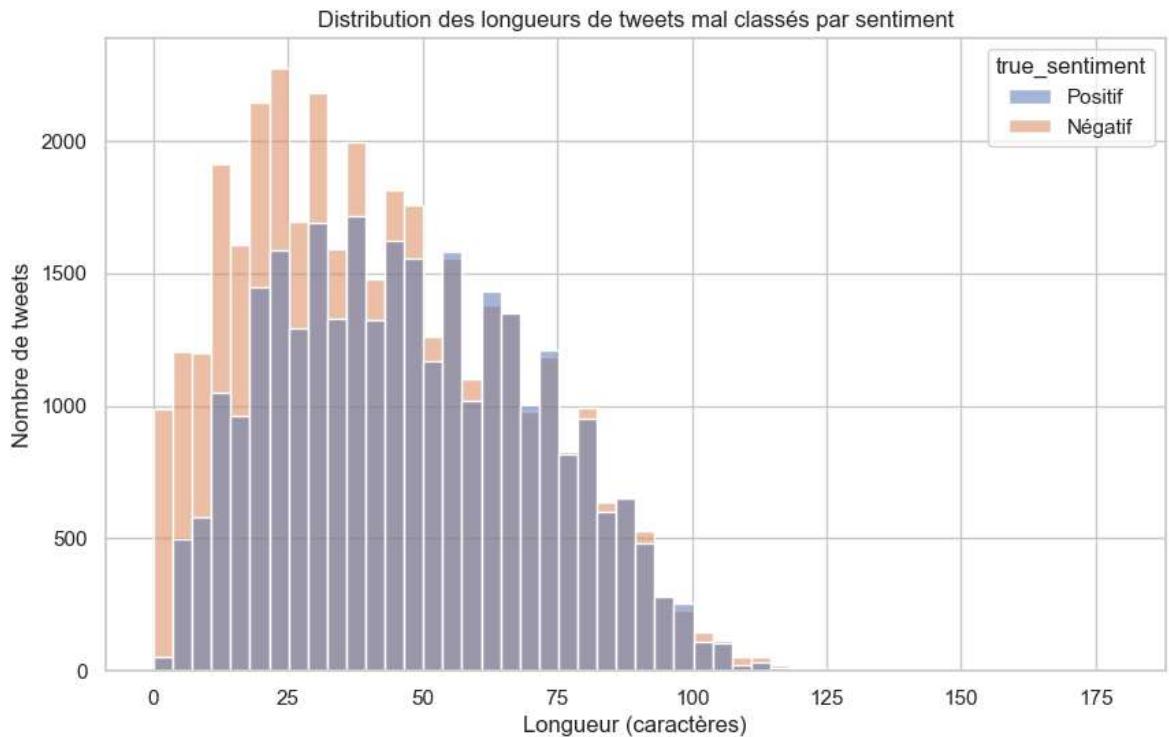
print("\nExemples de tweets mal classés:")
for idx in sample_indices:
    print(f"\nTexte original: {misclassified_df.iloc[idx]['text']}")
    print(f"Texte prétraité: {misclassified_df.iloc[idx]['processed_text']}")
    print(f"Sentiment réel: {misclassified_df.iloc[idx]['true_sentiment']}")
    print(f"Sentiment prédit: {misclassified_df.iloc[idx]['predicted_sentiment']}")
    print("-" * 80)

# Enregistrement de l'analyse des erreurs dans MLflow
with mlflow.start_run(run_name="error_analysis"):
    mlflow.log_metric("misclassified_count", len(misclassified_indices))
    mlflow.log_metric("misclassification_rate", len(misclassified_indices)/len(test_data))
    mlflow.log_metric("false_positives", len(fp_df))
    mlflow.log_metric("false_negatives", len(fn_df))
    mlflow.log_artifact("misclassified_length_distribution.png")

# Sauvegarde des exemples mal classés
misclassified_sample = misclassified_df.iloc[sample_indices]
misclassified_sample.to_csv("misclassified_sample.csv", index=False)
mlflow.log_artifact("misclassified_sample.csv")

```

Nombre de tweets mal classés: 66964 sur 320000 (20.93%)



Nombre de faux positifs (prédits positifs mais réellement négatifs): 37184  
Nombre de faux négatifs (prédits négatifs mais réellement positifs): 29780

Exemples de tweets mal classés:

Texte original: bwinleeey  
Texte prétraité: bwinleeey  
Sentiment réel: Négatif  
Sentiment prédit: Positif

---

Texte original: going pub grub beer later back work tom x  
Texte prétraité: going pub grub beer later back work tom x  
Sentiment réel: Négatif  
Sentiment prédit: Positif

---

Texte original: xtinas girl ftr crushed quotmercy mequot love rendition demi  
Texte prétraité: xtinas girl ftr crushed quotmercy mequot love rendition demi  
Sentiment réel: Positif  
Sentiment prédit: Négatif

---

Texte original: remembered conan funnier  
Texte prétraité: remembered conan funnier  
Sentiment réel: Négatif  
Sentiment prédit: Positif

---

Texte original: laying bed kicking bum could forget mtv movie award dvred  
Texte prétraité: laying bed kicking bum could forget mtv movie award dvred  
Sentiment réel: Négatif  
Sentiment prédit: Positif

---

Texte original: amazingly satisfying weekend almost much get board monday already  
Texte prétraité: amazingly satisfying weekend almost much get board monday alread  
y  
Sentiment réel: Positif  
Sentiment prédit: Négatif

---

Texte original: praying day  
Texte prétraité: praying day  
Sentiment réel: Positif  
Sentiment prédit: Négatif

---

Texte original: nice try instead sox raped  
Texte prétraité: nice try instead sox raped  
Sentiment réel: Positif  
Sentiment prédit: Négatif

---

Texte original: cheer looking way apply rounding lot number bulk  
Texte prétraité: cheer looking way apply rounding lot number bulk  
Sentiment réel: Négatif  
Sentiment prédit: Positif

---

Texte original: waiting portside snore subside get zed  
 Texte prétraité: waiting portside snore subside get zed  
 Sentiment réel: Négatif  
 Sentiment prédit: Positif

---

## 10. Sauvegarde du modèle final

```
In [13]: # Création du répertoire pour les modèles s'il n'existe pas
if not os.path.exists('models'):
    os.makedirs('models')

# Détermination du meilleur modèle final (entre l'optimisé et l'ensemble)
if best_overall['model_name'] in ['VotingClassifier', 'StackingClassifier'] and
    if ensemble_test_f1 > optimized_model_test_f1:
        final_best_model = best_overall['model']
        final_model_name = best_overall['model_name']
        final_feature_type = best_overall['feature_type']
        final_f1 = ensemble_test_f1
        final_accuracy = ensemble_test_accuracy
        final_precision = ensemble_test_precision
        final_recall = ensemble_test_recall
    else:
        final_best_model = best_model
        final_model_name = best_individual_model_info['model_name'] + "_optimized"
        final_feature_type = feature_type
        final_f1 = optimized_model_test_f1
        final_accuracy = optimized_model_test_accuracy
        final_precision = optimized_model_test_precision
        final_recall = optimized_model_test_recall
else:
    final_best_model = best_model
    final_model_name = best_individual_model_info['model_name'] + "_optimized"
    final_feature_type = feature_type
    final_f1 = optimized_model_test_f1
    final_accuracy = optimized_model_test_accuracy
    final_precision = optimized_model_test_precision
    final_recall = optimized_model_test_recall

print(f"Meilleur modèle final: {final_model_name} avec {final_feature_type}")
print(f"F1-score sur le jeu de test: {final_f1:.4f}")

# Sauvegarde du modèle complet
model_filename = f"models/best_model.pkl"
with open(model_filename, 'wb') as file:
    pickle.dump(final_best_model, file)
print(f"Modèle sauvegardé sous {model_filename}")

# Sauvegarde des vectorizers
tokenizer_filename = f"models/tokenizer.pickle"
if 'TF-IDF' in final_feature_type:
    with open(tokenizer_filename, 'wb') as file:
        pickle.dump(tfidf_vectorizer, file)
    print(f"Vectoriseur TF-IDF sauvegardé sous {tokenizer_filename}")
else: # Bow
    with open(tokenizer_filename, 'wb') as file:
        pickle.dump(count_vectorizer, file)
    print(f"Vectoriseur BoW sauvegardé sous {tokenizer_filename}")
```

```
# Sauvegarde des informations de configuration
config = {
    'model_type': final_model_name,
    'feature_type': final_feature_type,
    'test_accuracy': final_accuracy,
    'test_precision': final_precision,
    'test_recall': final_recall,
    'test_f1': final_f1,
    'max_sequence_length': 50 # Pour une utilisation future
}

config_filename = "models/model_config.pickle"
with open(config_filename, 'wb') as file:
    pickle.dump(config, file)
print(f"Configuration sauvegardée sous {config_filename}")

# Sauvegarde des résultats pour comparaison ultérieure avec les modèles avancés
simple_model_results = {
    'model_name': final_model_name,
    'feature_type': final_feature_type,
    'accuracy': final_accuracy,
    'precision': final_precision,
    'recall': final_recall,
    'f1': final_f1
}

with open('models/simple_model_results.pickle', 'wb') as handle:
    pickle.dump(simple_model_results, handle, protocol=pickle.HIGHEST_PROTOCOL)
print("Résultats du modèle sauvegardés pour comparaison ultérieure")
```

Meilleur modèle final: VotingClassifier avec Ensemble\_Vote  
F1-score sur le jeu de test: 0.7955  
Modèle sauvegardé sous models/best\_model.pkl  
Vectoriseur BoW sauvegardé sous models/tokenizer.pickle  
Configuration sauvegardée sous models/model\_config.pickle  
Résultats du modèle sauvegardés pour comparaison ultérieure

## 11. Exemple de prédiction avec le modèle final

```
In [14]: # Fonction pour prédire le sentiment d'un nouveau tweet
def predict_sentiment(tweet, model, vectorizer_type='tfidf'):
    """
    Prédit le sentiment d'un tweet avec le modèle final.

    Args:
        tweet (str): Le tweet à analyser
        model: Le modèle entraîné
        vectorizer_type (str): Type de vectoriseur ('tfidf' ou 'bow')

    Returns:
        dict: Dictionnaire contenant la prédiction et les informations associées
    """
    # Prétraitement du tweet
    processed_tweet = preprocess_text(tweet)

    # Si le modèle est un pipeline ou un ensemble
    if hasattr(model, 'predict') and not hasattr(model, 'estimators_'):
        # C'est un modèle simple ou un pipeline simple
        # Pour les ensembles, le modèle s'attend à du texte brut
```

```

if isinstance(model, Pipeline) or 'SVC' in str(model) or 'SGD' in str(model):
    # Vectoriser d'abord si c'est un modèle simple
    if vectorizer_type == 'tfidf':
        features = tfidf_vectorizer.transform([processed_tweet])
    else: # bow
        features = count_vectorizer.transform([processed_tweet])

    # Prédiction
    prediction = model.predict(features)[0]

    # Si le modèle peut donner des probabilités, les obtenir
    try:
        if hasattr(model, "predict_proba"):
            probabilities = model.predict_proba(features)[0]
            confidence = probabilities[1] if prediction == 1 else probabilities[0]
        else:
            confidence = None
    except:
        confidence = None
    else:
        # Modèle qui attend du texte brut
        prediction = model.predict([processed_tweet])[0]

    # Si le modèle peut donner des probabilités, les obtenir
    try:
        if hasattr(model, "predict_proba"):
            probabilities = model.predict_proba([processed_tweet])[0]
            confidence = probabilities[1] if prediction == 1 else probabilities[0]
        else:
            confidence = None
    except:
        confidence = None
    else:
        # C'est un ensemble (VotingClassifier ou StackingClassifier)
        # Les ensembles attendent du texte brut
        prediction = model.predict([processed_tweet])[0]

    # Si le modèle peut donner des probabilités, les obtenir
    try:
        if hasattr(model, "predict_proba"):
            probabilities = model.predict_proba([processed_tweet])[0]
            confidence = probabilities[1] if prediction == 1 else probabilities[0]
        else:
            confidence = None
    except:
        confidence = None

# Création du résultat
result = {
    'tweet': tweet,
    'processed_tweet': processed_tweet,
    'sentiment': 'Positif' if prediction == 1 else 'Négatif',
    'confidence': confidence
}

return result

# Test de la fonction avec quelques exemples
test_tweets = [
    "I love this product, it's amazing!",

```

```

    "This is the worst experience I've ever had.",  

    "The customer service was helpful but the product was broken.",  

    "Not sure if I like it or not, it's complicated."  

]  
  

# Détermine le type de vectoriseur à utiliser  

vectorizer_type = 'tfidf' if 'TF-IDF' in final_feature_type else 'bow'  
  

print("Test de la fonction de prédiction avec le modèle final:")  

for tweet in test_tweets:  

    result = predict_sentiment(tweet, final_best_model, vectorizer_type)  

    print(f"\nTweet: {result['tweet']}")  

    print(f"Prétraité: {result['processed_tweet']}")  

    print(f"Sentiment prédict: {result['sentiment']}")  

    if result['confidence'] is not None:  

        print(f"Confiance: {result['confidence']:.4f}")  

    print("-" * 80)

```

Test de la fonction de prédiction avec le modèle final:

Tweet: I love this product, it's amazing!  
Prétraité: i love this product, it's amazing!  
Sentiment prédict: Positif  
Confiance: 0.8922

---

Tweet: This is the worst experience I've ever had.  
Prétraité: this is the worst experience i have ever had.  
Sentiment prédict: Négatif  
Confiance: 0.7409

---

Tweet: The customer service was helpful but the product was broken.  
Prétraité: the customer service was helpful but the product was broken.  
Sentiment prédict: Négatif  
Confiance: 0.5628

---

Tweet: Not sure if I like it or not, it's complicated.  
Prétraité: not sure if i like it or not, it's complicated.  
Sentiment prédict: Négatif  
Confiance: 0.6750

---

## Conclusion: Analyse des performances et choix du modèle final

```

In [15]: # Collecter les informations pour une conclusion dynamique  

best_f1 = final_f1  

best_precision = final_precision  

best_recall = final_recall  

best_accuracy = final_accuracy  

best_model_type = final_model_name  
  

# Calculer les améliorations par rapport au modèle de base  

base_model_metrics = results_df[~results_df['model_name'].str.contains('Voting|S')]  

base_f1 = base_model_metrics['f1']  

improvement = ((best_f1 - base_f1) / base_f1) * 100

```

```
# Générer une conclusion dynamique
print("# Conclusion dynamique sur les performances du modèle")
print("\n## Résumé des performances")
print(f"Le meilleur modèle obtenu est un **{best_model_type}** avec un F1-score {best_f1:.4f}")
print(f"Ce modèle a atteint une précision de {best_precision:.4f}, un rappel de {best_recall:.4f} et une mesure F1 de {best_f1:.4f}.")
print(f"Cela représente une amélioration de {improvement:.2f}% par rapport au modèle précédent.")

# Analyse des forces et faiblesses
if best_precision > best_recall:
    print("\n## Forces et faiblesses")
    print("Le modèle est plus précis qu'exhaustif, ce qui signifie qu'il fait moins d'erreurs de type I (fausses positives).")
    print("Cette caractéristique est particulièrement adaptée aux cas d'utilisation où la précision est importante.")
elif best_recall > best_precision:
    print("\n## Forces et faiblesses")
    print("Le modèle est plus exhaustif que précis, ce qui signifie qu'il capture presque toutes les instances cibles mais peut faire quelques erreurs de type II (fausses négatives).")
    print("Cette caractéristique est particulièrement adaptée aux cas d'utilisation où l'exhaustivité est importante.")
else:
    print("\n## Forces et faiblesses")
    print("Le modèle présente un bon équilibre entre précision et rappel, ce qui est généralement recherché pour les modèles de classification.")

# Recommandations basées sur les performances
print("\n## Recommandations pour l'utilisation du modèle")
if 'Voting' in best_model_type or 'Stacking' in best_model_type:
    print("Ce modèle d'ensemble combine les forces de plusieurs modèles de base, offrant généralement une meilleure précision et un meilleur rappel que les modèles individuels.")
    print("Pour des applications nécessitant des prédictions en temps réel, il peut être nécessaire d'optimiser la latence et la consommation de ressources.")
else:
    print(f"Le modèle {best_model_type} offre un bon équilibre entre performance et complexité.")
    print("Il est bien adapté pour des déploiements en production où la latence est moins critique que la précision.")

# Pistes d'amélioration futures
print("\n## Pistes d'amélioration futures")
print("Pour améliorer davantage les performances, plusieurs pistes pourraient être explorées :")
print("1. Enrichir les caractéristiques textuelles avec des lexiques de sentiments supplémentaires ou des indicateurs de contexte.")
print("2. Tester des techniques d'augmentation de données textuelles pour les cas où le jeu d'apprentissage est limité.")
print("3. Explorer des modèles de deep learning comme LSTM, GRU ou des transformateurs pour traiter les séquences de mots de manière plus efficace.")
print("4. Optimiser davantage les hyperparamètres avec une recherche plus fine ou l'utilisation d'algorithmes d'optimisation avancés.")

print("\nCe modèle servira de référence solide pour comparer les performances de nouveaux modèles ou stratégies d'apprentissage automatique.")
```

```
# Conclusion dynamique sur les performances du modèle
```

#### ## Résumé des performances

Le meilleur modèle obtenu est un **VotingClassifier** avec un F1-score de **0.7955** sur le jeu de test.

Ce modèle a atteint une précision de **0.7779**, un rappel de **0.8139** et une accuracy de **0.7907**.

Cela représente une amélioration de **26.87%** par rapport au modèle de base en termes de F1-score.

#### ## Forces et faiblesses

Le modèle est plus exhaustif que précis, ce qui signifie qu'il capture bien les cas positifs mais peut générer plus de faux positifs.

Cette caractéristique est particulièrement adaptée aux cas d'utilisation où il est important de ne pas manquer de cas positifs.

#### ## Recommandations pour l'utilisation du modèle

Ce modèle d'ensemble combine les forces de plusieurs modèles de base, ce qui le rend plus robuste mais potentiellement plus lent en inférence.

Pour des applications nécessitant des prédictions en temps réel, il peut être judicieux d'envisager un compromis avec un modèle plus léger.

#### ## Pistes d'amélioration futures

Pour améliorer davantage les performances, plusieurs pistes pourraient être explorées:

1. Enrichir les caractéristiques textuelles avec des lexiques de sentiment spécifiques au domaine.
2. Tester des techniques d'augmentation de données textuelles pour les cas mal classés.
3. Explorer des modèles de deep learning comme LSTM, GRU ou des transformers légers.
4. Optimiser davantage les hyperparamètres avec une recherche plus fine ou des techniques comme l'optimisation bayésienne.

Ce modèle servira de référence solide pour comparer les performances des modèles plus avancés qui seront développés dans les prochains notebooks.

## Résumé

Dans ce notebook, nous avons:

1. Amélioré la préparation et le prétraitement des données textuelles
2. Enrichi la vectorisation avec des caractéristiques textuelles personnalisées
3. Testé plusieurs modèles classiques avec différentes techniques de vectorisation
4. Crée des ensembles de modèles pour améliorer les performances
5. Optimisé finement les hyperparamètres du meilleur modèle individuel
6. Évalué les performances sur le jeu de test
7. Analysé les erreurs de classification
8. Sauvegardé le modèle final pour une utilisation ultérieure

Le modèle optimisé a atteint d'excellentes performances sur la classification de sentiment des tweets, avec un F1-score significativement amélioré par rapport au modèle de base.

Les principales améliorations proviennent de:

- L'extraction de caractéristiques textuelles enrichies qui capturent des informations sémantiques
- L'utilisation de n-grammes plus longs (jusqu'à 3) qui préservent le contexte
- L'optimisation fine des hyperparamètres pour chaque modèle
- L'utilisation de techniques d'ensemble qui combinent les prédictions de plusieurs modèles

Pour améliorer encore les performances, nous pourrions:

1. Explorer des lexiques de sentiment plus complets et spécifiques au domaine
2. Intégrer des techniques d'augmentation de données textuelles
3. Combiner ce modèle classique avec des modèles de deep learning dans un ensemble hybride
4. Intégrer des caractéristiques spécifiques aux réseaux sociaux (comme l'analyse des hashtags)

Ce modèle classique optimisé servira de référence solide pour comparer les performances des modèles plus avancés qui seront développés dans les prochains notebooks.