COMP9727 Assignment 25T2: Content-Based Music Recommendation

Written by Jiacheng Nan (z5497519) on 11/06/2025

This assignment is to aim to build a classic content-based music recommender from scratch, which uses the user's personal music interests and preferences to recommend a personalized playlist.

Environment and Data Setup

In [4]:
```python
# General and text processing imports
import pandas as pd
import nltk
import re
import numpy as np
import matplotlib.pyplot as plt

# NLTK data setup
nltk.download('punkt')
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

# Features
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

# Models
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
from sklearn.svm import LinearSVC

# Validation
from sklearn.model_selection import StratifiedKFold, cross_validate

# Metrics
from sklearn.metrics import accuracy_score,f1_score, make_scorer
from sklearn.metrics.pairwise import cosine_similarity
```

```
[nltk_data] Downloading package punkt to /home/jiacheng/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/jiacheng/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Part 1 Topic Classification

1. Regex Preprocessing Comparison and Cross-Validation

In the tutorial, the regular expression regex `[^\w\s]` was used for text preprocessing. This pattern removes all punctuation and special symbols, preserving only alphanumeric characters and whitespace. While effective for producing normalized tokens, this method may be overly aggressive for certain natural language processing tasks.

In particular, punctuation marks such as `&`, `=`, and `?` can carry important semantic meaning, especially in informal or creative text such as song lyrics. The removal of these symbols may degrade classification performance, as such tokens can contribute to the uniqueness of an artist's style or lyrical theme. Moreover, since text is lowercased during preprocessing, including uppercase characters in the allowed token set is redundant and may introduce unnecessary noise.

To assess the impact of punctuation retention, two preprocessing strategies were implemented:

- Remove all punctuation, remaining the same as the tutorial's setup
- Retain key punctuation marks and only include lower case letters

Both strategies were evaluated using two classification models: **Multinomial Naive Bayes (MNB)** and **Bernoulli Naive Bayes (BNB)**. Performance was assessed using standard metrics: `accuracy`, `macro-f1`, and `micro-f1`.

Unlike the tutorial, which relied on a single `train_test_split()`, this evaluation used **Stratified 5-Fold Cross-Validation** via `StratifiedKFold()`. This method was chosen to better accommodate potential class imbalance across the five topic labels, and to yield more stable and generalizable performance estimates. Stratification ensures that each fold maintains similar class distributions, while cross-validation reduces the risk of performance fluctuation due to random splits, which is common with fixed train-test divisions.

1.1 Preprocessing Setup

In [5]:
```python
# Load dataset and combine the coloumns into a single text column
df = pd.read_csv("dataset.tsv", sep='\t')
df = df.drop_duplicates().dropna()
categories = ['artist_name', 'track_name', 'release_date', 'genre', 'lyrics', 't
df['content'] = df[categories].astype(str).agg(' '.join, axis=1)

# Load stopwords and initialize stemmer
stop_words_NLTK = set(stopwords.words('english'))
ps = PorterStemmer()

# Define preprocessing function
def preprocess_text(text, regex, stop_words, lowercase=True, stemming=True):
    if lowercase:
        text = text.lower()
    text = re.sub(regex, '', text)
    tokens = word_tokenize(text)
    tokens = [word for word in tokens if word not in stop_words]
    if stemming:
        tokens = [ps.stem(word) for word in tokens]
    return ' '.join(tokens)
```

1.2 Remove all punctuations

```
In [6]: df['content_A'] = df['content'].apply(lambda c: preprocess_text(c, regex=r"[^\w\
```

1.3 Retain key punctuation marks

```
In [7]: df['content_B'] = df['content'].apply(lambda c:preprocess_text(c, regex = r'[^a-
```

1.4 Applying Preprocessing and Run Evaluation

```
In [8]: # Define evaluation function for cross-validation
def evaluate_model(texts, labels, models, token_pattern, max_features):
    # Vectorize
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    vectorizer = CountVectorizer(max_features=max_features, token_pattern=token_
    X = vectorizer.fit_transform(texts)
    y = labels
    records = []

    scorers = {
        'Accuracy': make_scorer(accuracy_score),
        'Macro-F1': make_scorer(f1_score, average='macro'),
        'Micro-F1': make_scorer(f1_score, average='micro')
    }

    for model_name, model in models.items():
        # Cross-validate the model
        scores = cross_validate(model, X, y, cv=cv, scoring=scorers)
        row = {'Model': model_name}
        for metric in scorers:
            row[f'{metric} mean'] = scores[f'test_{metric}'].mean()
            row[f'{metric} std']  = scores[f'test_{metric}'].std()
        records.append(row)

    print(pd.DataFrame(records),"\n")

# Run comparison
models = {'MNB': MultinomialNB(),'BNB': BernoulliNB()}
token_pattern = r"(?u)\b\w\w+\b"

print("No Punctuation:")
evaluate_model(df['content_A'], df['topic'], models, token_pattern, max_features

print("Keeping Punctuations and lower case:")
evaluate_model(df['content_B'], df['topic'], models, token_pattern, max_features
No Punctuation:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.008705       0.852283      0.015367
1   BNB       0.913514      0.008705       0.760820      0.011860

   Micro-F1 mean  Micro-F1 std
0       0.877027      0.008705
1       0.913514      0.008705

Keeping Punctuations and lower case:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.012926       0.854101      0.019455
1   BNB       0.913514      0.006266       0.761430      0.009896

   Micro-F1 mean  Micro-F1 std
0       0.877027      0.012926
1       0.913514      0.006266
```

1.5 Results

Retaining selected punctuation produced a slight but consistent improvement in `macro-F1` scores across both models:

- **MNB**: from 0.852283 to 0.854101
- **BNB**: from 0.760820 to 0.761430

In contrast, `accuracy` and `micro-F1` scores remained unchanged, with overlapping standard deviations indicating negligible effect. Although the improvements in `macro-F1` were modest and within potential variance, the retention of punctuation did not negatively impact performance.

Based on these findings, the second preprocessing strategy of **retaining key punctuation and limiting input to lowercase letter** was selected for all further experiments. This approach preserves potentially meaningful symbols in the lyrical data while maintaining a consistent token structure for vectorization and classification.

2. Developing a Multinomial Naive Bayes Model and Further Tuning

Following the baseline implementation of both MNB and BNB models, the next objective was to enhance the pipeline by optimizing its preprocessing components. The experiement focused on varying key factors to assess their influence on model performance, which are:

- **Special character removal**: varying the regex to control which symbols are stripped or retained (extending the earlier comparison to include more variations)
- **Token definition**: defining what a “word” is
- **Stopword list selection**: comparing the performance of NLTK’s stop-word list, scikit-learn’s built-in list, or no stop-words at all
- **Case normalization and stemming**: evaluating the effect of converting all text to lowercase and applying Porter stemming

Each combination of preprocessing choices was assessed using 5-fold stratified cross-validation, with performance evaluated based on `accuracy`, `macro-f1`, and `micro-f1` scores.

2.1 Further Special Character Removal

Although retaining key punctuation provided a modest improvement, not all symbols contribute equally to classification performance. Many special characters occur so infrequently in the dataset that preserving them increases complexity without yielding benefits. To identify the most impactful symbols, each punctuation mark was removed individually, and the corresponding change in model performance was recorded. This experiment guided the construction of a regular expression that retains only those characters demonstrating a positive effect on the accuracy in classification.

In [9]:
```
df['content'] = df['content'].apply(lambda c:preprocess_text(c, regex = r"[^a-z\
evaluate_model(df['content'], df['topic'], models, token_pattern, max_features=3
```

```
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.012926       0.854101      0.019455   
1   BNB       0.913514      0.006266       0.761430      0.009896   

   Micro-F1 mean  Micro-F1 std  
0       0.877027      0.012926  
1       0.913514      0.006266  
```

After systematically evaluating the impact of each punctuation mark, only those whose removal led to a measurable performance drop, namely `.`, `'`, `/`, `-` and alphabet `a-z`, were retained in the final regular expression. All other symbols were excluded to simplify the preprocessing pipeline and reduce noise. Although the overall metrics remain unchanged from Section 1, this optimized regex yields a cleaner and more efficient preprocessing.

2.2 Token Definition

Tokenization divides text into discrete units called tokens, forming the basis of any NLP pipeline. In scikit-learn's `CountVectorizer()`, the `token_pattern` parameter, defined by a regular expression, determines which character sequences are recognized as valid tokens. The choice of pattern can substantially affect model performance.

To determine the most effective token definition, several patterns were compared:

- **Only letters**: exclude digits entirely, retaining only alphabetic characters
- **Punctuation handling**
  - **Default**: use scikit-learn's standard pattern `(\b\w\w+\b)`, which accepts words of length two or more, including digits and underscores. (this excludes single digits)
  - **Keep key punctuations**: allow selected symbols, `&`, `?`, `!`, `'`, `.`, within tokens

    - **Hyphen words**: treat sequences like `well-known` or `state-of-the-art` as single tokens by permitting internal hyphens
    - **Full punctuation**: keeping a broader range of symbols within tokens
- **Minimum token length**: lower the minimum token length to one character, which includs digits

Each variant was applied in turn, and the resulting classification performance of `accuracy`, `macro-F1`, and `micro-F1` scores was recorded using **stratified 5-fold cross-validation**. The pattern yielding the highest scores was selected for the final pipeline.

In [10]:
```
token_patterns = {
    'only_alphabet': r"(?u)\b[a-zA-Z]+\b",
    'default': r"(?u)\b\w\w+\b",
    'keep_key_punct': r"(?u)\b[\w'&!?\.]+\b",
    'hyphen_words': r"(?u)\b[a-zA-Z]+(?:-[a-zA-Z]+)*\b",
    'full_punct': r"(?u)\b[\w!\"#$%&'()*+,\-./:;<=>?@\[\]\\^_`{|}~]+\b",
    'allowing_single_char': r"(?u)\b\w+\b",
}

for name, pattern in token_patterns.items():
    print(f"Pattern {name}:")
    evaluate_model(df['content'], df['topic'], models, pattern, max_features=300
Pattern only_alphabet:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.876351      0.012926       0.852908      0.020423   
1   BNB       0.913514      0.006957       0.761101      0.011814   

   Micro-F1 mean  Micro-F1 std  
0       0.876351      0.012926  
1       0.913514      0.006957  

Pattern default:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.012926       0.854101      0.019455   
1   BNB       0.913514      0.006266       0.761430      0.009896   

   Micro-F1 mean  Micro-F1 std  
0       0.877027      0.012926  
1       0.913514      0.006266  

Pattern keep_key_punct:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.013913       0.854947      0.021507   
1   BNB       0.913514      0.006266       0.761375      0.009835   

   Micro-F1 mean  Micro-F1 std  
0       0.877703      0.013913  
1       0.913514      0.006266  

Pattern hyphen_words:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.011819       0.854196      0.019828   
1   BNB       0.914189      0.007584       0.761536      0.011782   

   Micro-F1 mean  Micro-F1 std  
0       0.877027      0.011819  
1       0.914189      0.007584  

Pattern full_punct:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877027      0.010811       0.854187      0.019202   
1   BNB       0.913514      0.006957       0.761325      0.011082   

   Micro-F1 mean  Micro-F1 std  
0       0.877027      0.010811  
1       0.913514      0.006957  

Pattern allowing_single_char:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.876351      0.012926       0.852908      0.020423   
1   BNB       0.913514      0.006957       0.761101      0.011814   

   Micro-F1 mean  Micro-F1 std  
0       0.876351      0.012926  
1       0.913514      0.006957  
```

The results indicate that the `keep_key_punct` pattern provided the most consistent benefit for MNB, increasing accuracy from **0.8770** to **0.8777**, with corresponding small gains in both macro-F1 and micro-F1. In contrast, the `hyphenated-word` pattern yielded the highest accuracy for BNB, improving from **0.9135** to **0.9142**, while having negligible impact on MNB. All other patterns either produced no improvement or slightly degraded performance.

A combined token pattern (retaining both key punctuation and hyphens) did not enhance either classifier further. To select a single pattern for subsequent experiments, the accuracy means of both MNB and BNB under each pattern were summed. The `keep_key_punct` pattern achieved the highest aggregate score and was therefore adopted for all future tuning in this assignment.

2.3 Stopword List Selection

In natural language text, very common words, such as articles, prepositions, and pronouns, often carry little semantic value and are therefore filtered out as **stopwords**. Removing stopwords can reduce noise and shrink the feature space, improving model efficiency. However, in tasks involving short or specialized texts, excessive removal may discard informative tokens and harm performance.

To determine the optimal stopword configuration, three options were compared:

- **No stopwords**: keep every single token
- **NLTK stopwords**: use the comprehensive list provided by NLTK, which includes common English words as well as several colloquial and dialectal terms
- **scikit-learn stopwords**: use the built-in list from scikit-learn, which is smaller and more conservative, focusing on standard grammatical words and excluding some colloquial terms

Each configuration was evaluated via **stratified 5-fold cross-validation** using `accuracy`, `macro-F1`, and `micro-F1` metrics. The stopword list yielding the highest score was selected for the final pipeline.

In [11]:
```
# Applying updates from previous section to originial content
df['content'] = df['content'].apply(lambda c:preprocess_text(c, regex = r"[^a-z\

#  Creating raw content for further processing
df['raw_content'] = df['content']
stopword_lists = {
    'no_stopwords': set(),
    'nltk': set(stopwords.words('english')),
    'sklearn': set(ENGLISH_STOP_WORDS)
}
regex = r"[^a-z\s.'\/-]"
token_pattern = r"(?u)\b[\w'&!?\.]+\b"

for name, stop_word in stopword_lists.items():
    df['raw_content'] = df['content'].apply(lambda c: preprocess_text(c, regex=r
    print(f"Stopword List: {name}")
    evaluate_model(df['raw_content'], df['topic'], models, token_pattern, max_fe
Stopword List: no_stopwords
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.011585       0.854195      0.018993
1   BNB       0.913514      0.008164       0.761449      0.010366

   Micro-F1 mean  Micro-F1 std
0       0.877703      0.011585
1       0.913514      0.008164

Stopword List: nltk
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.010978       0.854084      0.016864
1   BNB       0.913514      0.008164       0.761473      0.010367

   Micro-F1 mean  Micro-F1 std
0       0.877703      0.010978
1       0.913514      0.008164

Stopword List: sklearn
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.870946      0.009165       0.835782      0.012011
1   BNB       0.911486      0.008913       0.760498      0.010589

   Micro-F1 mean  Micro-F1 std
0       0.870946      0.009165
1       0.911486      0.008913
```

From the results, it is evident that omitting stopword removal entirely and using NLTK's stopword list yielded identical performance for both classifiers. In contrast, applying scikit-learn's stopword list reduced performance.

Because scikit-learn's stopword set degraded both models and NLTK's list maintained equivalent results to keeping all tokens, **NLTK's stopword list** was selected as the optimal choice. This list will be used in the final preprocessing pipeline.

2.4 Case Normalization & Stemming

Two final preprocessing steps were evaluated:

1. **Case normalization**: converting all text to lowercase and collapsing same words into the same token, reducing sparsity, but may lose useful distinctions in proper nouns or acronyms
2. **Stemming**: applying a Porter stemmer reduces words to their root form, which can improve recall by grouping inflected forms, though it may over-simplify certain tokens

Four pipeline configurations were tested:

- **Scenario A: Lowercase and Stemming**
- **Scenario B: Lowercase only**
- **Scenario C: Stemming only**
- **Scenario D: No lowercase, no stemming**

Each scenario was evaluated using **stratified 5-fold cross-validation** and the metrics `accuracy`, `macro-F1`, and `micro-F1`. The configuration achieving the highest score was adopted for all subsequent experiments.

In [12]:
```
scenarios = [
    ('A: lowercase_and_stemming', True,  True),
    ('B: lowercase_only', True,  False),
    ('C: stemming_only', False, True),
    ('D: nothing', False, False)
]

df['raw_content'] = df['content']
stop_word = stop_words_NLTK

for label, lower, stem in scenarios:
    processed = df['raw_content'].apply(
        lambda c: preprocess_text(
            c,
            regex=regex,
            stop_words=stop_word,
            lowercase=lower,
            stemming=stem
        )
    )

    print(f"Scenario {label}: ")
    evaluate_model(processed, df['topic'], models, token_pattern, max_features=3
Scenario A: lowercase_and_stemming:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.010978       0.854084      0.016864   
1   BNB       0.913514      0.008164       0.761473      0.010367   

   Micro-F1 mean  Micro-F1 std  
0       0.877703      0.010978  
1       0.913514      0.008164  

Scenario B: lowercase_only:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.878378      0.010468       0.853753      0.017911   
1   BNB       0.913514      0.006620       0.761719      0.010495   

   Micro-F1 mean  Micro-F1 std  
0       0.878378      0.010468  
1       0.913514      0.006620  

Scenario C: stemming_only:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.010978       0.854084      0.016864   
1   BNB       0.913514      0.008164       0.761473      0.010367   

   Micro-F1 mean  Micro-F1 std  
0       0.877703      0.010978  
1       0.913514      0.008164  

Scenario D: nothing:
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.878378      0.010468       0.853753      0.017911   
1   BNB       0.913514      0.006620       0.761719      0.010495   

   Micro-F1 mean  Micro-F1 std  
0       0.878378      0.010468  
1       0.913514      0.006620  
```

From the results above, the following observations were made:

- **Scenarios B and D** yielded identical, lower performance metrics
- **Scenarios A and C** achieved the highest scores, matching the best results from Section 2.3

Although Scenario C (stemming only) matched Scenario A in performance, leaving out lowercase conversion retains redundant uppercase tokens, which adds noise and sparsity in a bag-of-words model. Lowercasing is an inexpensive normalization step that unifies token variants without harming accuracy. Accordingly, **Scenario A (lowercase and stemming)** was selected for the final preprocessing pipeline.

3. Compare BNB and MNB with 5-Fold Cross-Validation

This section presents a performance comparison between **BernoulliNB** and **MultinomialNB** using the fully tuned preprocessing pipeline from Sections 1 and 2. To obtain robust and unbiased estimates on the **imbalanced** topic dataset, **stratified 5-fold cross-validation** was employed.

Models were evaluated according to **accuracy** and **F1 score** metrics. Although accuracy provides an overall measure of correctness, it may be inflated by the majority classes in imbalanced datasets. Consequently, **macro–average F1** was selected as the primary metric, since it equally weights all classes and emphasizes performance on minority categories. By analyzing these metrics, the superior classifier for topic prediction in this assignment will be identified.

In [13]:
```
regex = r"[^a-z\s.'\/-]"
token_pattern = r"(?u)\b[\w'&!?\.]+\b"
stop_word = stop_words_NLTK

df['content'] = df['content'].apply(
    lambda c: preprocess_text(
        c,
        regex=regex,
        stop_words=stop_word,
        lowercase=True,
        stemming=True
    )
)

evaluate_model(df['content'], df['topic'], models, token_pattern, max_features=3
```

```
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.010978       0.854084      0.016864   
1   BNB       0.913514      0.008164       0.761473      0.010367   

   Micro-F1 mean  Micro-F1 std  
0       0.877703      0.010978  
1       0.913514      0.008164  
```

3.1 Results

BNB encodes each token as a binary feature (present or absent), while MNB additionally leverages term frequencies. The stratified 5-fold cross-validation results reveal:

- **Accuracy**: BNB achieved higher overall `accuracy`, since its binary representation often over-predicts majority classes and thus records more correct predictions in raw counts
- **Macro-F1**: MNB outperformed BNB on `macro-F1`. By modeling word counts, MNB more effectively distinguishes minority topic patterns and reduces majority class bias.

Given the imbalanced distribution of the five topics, **macro-F1** provides a more equitable measure of performance, as it assigns equal weight to each class. Therefore, **MultinomialNB** was chosen as the preferred classifier for all subsequent topic classification experiments.

4. Number of Features

The size of feature words in `CountVectorizer()` is a critical hyperparameter. Too few features may exclude discriminative terms essential for topic separation, and too many features can introduce noise, increase sparsity, and risk overfitting.

To identify the optimal balance, both MNB and BNB were evaluated over a sequence of feature word sizes, ranging from **1** to **10,000** features in increments of **1,000**. For each setting, models were trained and assessed using **stratified 5-fold cross-validation**, and performance was recorded in terms of **accuracy** and **micro-F1**.

The resulting performance curves were analyzed to select the feature words size that maximized generalization while avoiding unnecessary complexity.

In [14]:
```
num_of_features = range(1, 10001, 1000)
df['raw_content'] = df['content']

for max_features in num_of_features:
    print(f"Max Features: {max_features}")
    evaluate_model(df['raw_content'], df['topic'], models, token_pattern, max_fe
Max Features: 1
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.329054      0.001655       0.099034      0.000375
1   BNB       0.325000      0.005812       0.123347      0.029882

   Micro-F1 mean  Micro-F1 std
0       0.329054      0.001655
1       0.325000      0.005812

Max Features: 1001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.908108      0.013067       0.890328      0.018687
1   BNB       0.968243      0.009698       0.934940      0.025051

   Micro-F1 mean  Micro-F1 std
0       0.908108      0.013067
1       0.968243      0.009698

Max Features: 2001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.886486      0.010598       0.865110      0.011559
1   BNB       0.925676      0.006410       0.797859      0.012435

   Micro-F1 mean  Micro-F1 std
0       0.886486      0.010598
1       0.925676      0.006410

Max Features: 3001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.010978       0.854084      0.016864
1   BNB       0.913514      0.008164       0.761473      0.010367

   Micro-F1 mean  Micro-F1 std
0       0.877703      0.010978
1       0.913514      0.008164

Max Features: 4001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.877703      0.008385       0.848938      0.010999
1   BNB       0.897297      0.012568       0.737599      0.012614

   Micro-F1 mean  Micro-F1 std
0       0.877703      0.008385
1       0.897297      0.012568

Max Features: 5001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.874324      0.008108       0.843615      0.008836
1   BNB       0.879054      0.006891       0.714524      0.012786

   Micro-F1 mean  Micro-F1 std
0       0.874324      0.008108
1       0.879054      0.006891

Max Features: 6001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.870270      0.004054       0.832933      0.004097
1   BNB       0.857432      0.012532       0.681526      0.017146

   Micro-F1 mean  Micro-F1 std
```

```
0       0.870270      0.004054
1       0.857432      0.012532

Max Features: 7001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.871622      0.007704       0.831312      0.007183
1   BNB       0.828378      0.013412       0.636557      0.013106

   Micro-F1 mean  Micro-F1 std
0       0.871622      0.007704
1       0.828378      0.013412

Max Features: 8001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.866216      0.006957       0.816944      0.006492
1   BNB       0.809459      0.010811       0.598937      0.013805

   Micro-F1 mean  Micro-F1 std
0       0.866216      0.006957
1       0.809459      0.010811

Max Features: 9001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.862838      0.007584       0.808370      0.015049
1   BNB       0.799324      0.009930       0.581751      0.001734

   Micro-F1 mean  Micro-F1 std
0       0.862838      0.007584
1       0.799324      0.009930
```

4.1 Narrowing Feature Size

The results indicated that performance peaked at a size of approximately **1,001** features. Beyond **2,001** features, all evaluation metrics began to decline gradually. This pattern suggests that the optimal size lies within the range **1,001 to 2,001**. To pinpoint the exact range, the feature range was narrowed to **1,001 to 2,001**, using increments of **100** for a more specific search.

In [15]:
```
num_of_features = range(1, 2002, 100)
df['raw_content'] = df['content']

for max_features in num_of_features:
    print(f"Max Features: {max_features}")
    evaluate_model(df['raw_content'], df['topic'], models, token_pattern, max_fe
Max Features: 1
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.329054      0.001655       0.099034      0.000375   
1   BNB       0.325000      0.005812       0.123347      0.029882   

   Micro-F1 mean  Micro-F1 std  
0       0.329054      0.001655  
1       0.325000      0.005812  

Max Features: 101
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.863514      0.016770       0.824916      0.025341   
1   BNB       0.931081      0.008705       0.831388      0.030741   

   Micro-F1 mean  Micro-F1 std  
0       0.863514      0.016770  
1       0.931081      0.008705  

Max Features: 201
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.913514      0.009698       0.888709      0.018658   
1   BNB       0.979054      0.011585       0.955639      0.026281   

   Micro-F1 mean  Micro-F1 std  
0       0.913514      0.009698  
1       0.979054      0.011585  

Max Features: 301
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.929730      0.008653       0.918843      0.013858   
1   BNB       0.990541      0.007524       0.987201      0.010750   

   Micro-F1 mean  Micro-F1 std  
0       0.929730      0.008653  
1       0.990541      0.007524  

Max Features: 401
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.932432      0.006410       0.917102      0.012442   
1   BNB       0.985135      0.010158       0.977985      0.016154   

   Micro-F1 mean  Micro-F1 std  
0       0.932432      0.006410  
1       0.985135      0.010158  

Max Features: 501
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.937838      0.008164       0.918929      0.013443   
1   BNB       0.985811      0.010768       0.978482      0.016669   

   Micro-F1 mean  Micro-F1 std  
0       0.937838      0.008164  
1       0.985811      0.010768  

Max Features: 601
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.925000      0.008108       0.905378      0.019874   
1   BNB       0.981081      0.012568       0.971204      0.019544   

   Micro-F1 mean  Micro-F1 std  
```

```
0       0.925000      0.008108
1       0.981081      0.012568

Max Features: 701
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.922973      0.008108        0.90656      0.012762   
1   BNB       0.980405      0.013581        0.97050      0.020418   

   Micro-F1 mean  Micro-F1 std  
0       0.922973      0.008108  
1       0.980405      0.013581  

Max Features: 801
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.916216      0.012162       0.899457      0.017174   
1   BNB       0.974324      0.013614       0.958796      0.021670   

   Micro-F1 mean  Micro-F1 std  
0       0.916216      0.012162  
1       0.974324      0.013614  

Max Features: 901
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.910135      0.013781       0.892588      0.020363   
1   BNB       0.972297      0.011973       0.951152      0.020031   

   Micro-F1 mean  Micro-F1 std  
0       0.910135      0.013781  
1       0.972297      0.011973  

Max Features: 1001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.908108      0.013067       0.890328      0.018687   
1   BNB       0.968243      0.009698       0.934940      0.025051   

   Micro-F1 mean  Micro-F1 std  
0       0.908108      0.013067  
1       0.968243      0.009698  

Max Features: 1101
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.901351      0.012348       0.882639      0.015333   
1   BNB       0.962838      0.009314       0.922294      0.026272   

   Micro-F1 mean  Micro-F1 std  
0       0.901351      0.012348  
1       0.962838      0.009314  

Max Features: 1201
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.904730      0.014076       0.890171      0.017526   
1   BNB       0.958108      0.007277       0.912960      0.018855   

   Micro-F1 mean  Micro-F1 std  
0       0.904730      0.014076  
1       0.958108      0.007277  

Max Features: 1301
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.899324      0.016328       0.884284      0.021999   
```

```
1   BNB       0.953378      0.006891       0.899091      0.020891

   Micro-F1 mean  Micro-F1 std
0       0.899324      0.016328
1       0.953378      0.006891

Max Features: 1401
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.899324      0.015614       0.883086      0.022076
1   BNB       0.949324      0.007402       0.884957      0.034854

   Micro-F1 mean  Micro-F1 std
0       0.899324      0.015614
1       0.949324      0.007402

Max Features: 1501
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.900676      0.016356       0.885564      0.021529
1   BNB       0.945270      0.006891       0.875759      0.028296

   Micro-F1 mean  Micro-F1 std
0       0.900676      0.016356
1       0.945270      0.006891

Max Features: 1601
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.897297      0.014428       0.880318      0.016986
1   BNB       0.943243      0.006891       0.870783      0.036023

   Micro-F1 mean  Micro-F1 std
0       0.897297      0.014428
1       0.943243      0.006891

Max Features: 1701
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.893243      0.011225       0.873690      0.012653
1   BNB       0.939865      0.006891       0.859643      0.030166

   Micro-F1 mean  Micro-F1 std
0       0.893243      0.011225
1       0.939865      0.006891

Max Features: 1801
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.889189      0.010554       0.869486      0.014553
1   BNB       0.934459      0.005056       0.831365      0.022167

   Micro-F1 mean  Micro-F1 std
0       0.889189      0.010554
1       0.934459      0.005056

Max Features: 1901
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.887838      0.011184       0.865881      0.014144
1   BNB       0.931081      0.006266       0.817022      0.012544

   Micro-F1 mean  Micro-F1 std
0       0.887838      0.011184
1       0.931081      0.006266
```

```
Max Features: 2001
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.886486      0.010598       0.865110      0.011559
1   BNB       0.925676      0.006410       0.797859      0.012435

   Micro-F1 mean  Micro-F1 std
0       0.886486      0.010598
1       0.925676      0.006410
```

4.2 Results

As the feature word size decreased to **1 to 500**, both models showed consistent performance gains. The peak `macro-F1` scores occurred at 500 features, with BNB score of **0.9784** and MNB of **0.9189**.

Beyond 500 features, all metrics began to decline. Therefore, a **maximum feature size of 500** was selected for all further experiments, as it maximizes model performance while minimizing extraneous features.

5. Support Vector Machines (LinearSVM)

Support Vector Machines are supervised, margin-based classifiers that identify a hyperplane in high dimensional feature space to maximally separate classes. In text classification, each token corresponds to one dimension, resulting in very sparse vectors. A **linear SVM** often outperforms Naive Bayes models by emphasizing discriminative features and reducing the influence of noisy features.

Given the lyrics dataset, represented as a 500-dimensional bag-of-words, linear SVM is well suited to this high dimensional, sparse setting. It offers greater flexibility than Naive Bayes in weighting combinations of tokens and has a strong record in NLP tasks.

The `LinearSVC()` implementation requires tuning two hyperparameters. One of them is the `max_iter`, which is the maximum number of optimization iterations. The default of 1,000 iterations in scikit-learn may not achieve convergence, and triggering a warning. To ensure convergence, `max_iter` was increased in increments of **500** until no warnings occurred. The other hyperpartameter is the `dual`, which is a boolean flag that selects between solving the dual or the primal optimization problem. Even knowing that `dual=False` is typically faster when **n_samples > n_features**, both settings were evaluated to identify the best trade-off between efficiency and accuracy.

The evaluation procedure mirrored Section 3, employing **stratified 5-fold cross-validation** and the metrics `accuracy` and `macro-F1`. It was hypothesized that linear SVM would match or exceed the performance of both BNB and MNB.

```
In [16]: df['content'] = df['content'].apply(
             lambda c: preprocess_text(
                 c,
                 regex=regex,
                 stop_words=stop_word,
                 lowercase=True,
                 stemming=True
```

```
    )
)

# Define models for evaluation
models = {
    'MNB': MultinomialNB(),
    'BNB': BernoulliNB(),
    'SVM': LinearSVC(max_iter=2500, dual=False)
}

evaluate_model(df['content'], df['topic'], models, token_pattern, max_features=5
```

```
  Model  Accuracy mean  Accuracy std  Macro-F1 mean  Macro-F1 std  \
0   MNB       0.935811      0.005653       0.916603      0.011561   
1   BNB       0.986486      0.009791       0.978959      0.016404   
2   SVM       0.970946      0.010598       0.957399      0.016039   

   Micro-F1 mean  Micro-F1 std  
0       0.935811      0.005653  
1       0.986486      0.009791  
2       0.970946      0.010598  
```

5.1 Results

Through systematic tuning, convergence warnings disappeared once `max_iter` was set to **2,500**, and neither higher nor lower values improved its performance. Similarly, setting `dual=False` consistently outperformed `dual=True`. Therefore, the final SVM configuration used `max_iter=2500` and `dual=False`.

When benchmarked alongside our Naive Bayes classifiers, the linear SVM did outperform MNB on all metrics but did not surpass BNB. This result diverges from our initial hypothesis and from the findings in Section 3, where MNB had demonstrated higher `macro-F1` score than BNB. The strong performance of BNB in this round is likely attributable to the reduced feature words size to 500, which produces a binary feature signal that BNB exploits more effectively.

In conclusion, **Bernoulli Naive Bayes**, paired with the optimized preprocessing pipeline, was selected as the final classifier for all remaining experiments.

Part 2 Recommendation Methods

1. Building Recommendation System

Having established and tuned the Bernoulli Naive Bayes classifier in Part 1, the next phase simulates a content-based recommendation system. This system leverages the topic predictions generated by BNB to suggest new songs tailored to individual user preferences.

1.1 Preprocessing Setup

The preprocessing steps in this part is similar to Part 1. The only difference is that the dataset is divided into four, represented by weeks. Week 1 to 3 will be the training set and week 4 will be the test set. Details of the preprocessing steps (e.g., regex, token pattern, stopwords, stemming) are inherited from the final pipeline established in Part 1.

In [17]:
```
# Load dataset and combine the coloumns into a single text column
df = pd.read_csv("dataset.tsv", sep='\t')
df = df.drop_duplicates().dropna()
categories = ['artist_name', 'track_name', 'release_date', 'genre', 'lyrics']
df['content'] = df[categories].astype(str).agg(' '.join, axis=1)

# Split dataset into training and testing sets
train = df.iloc[:750].copy()
test = df.iloc[750:].copy()

# Load best preprocessing parameters
chosen_stop_words = set(stopwords.words('english'))
ps = PorterStemmer()
chosen_regex = r"[^a-z\s.'\/-]"
chosen_token_pattern = r"(?u)\b[\w'&!?\.]+\b"
chosen_max_features = 500

# Define preprocessing function
def preprocess_text_recommender(text):
    text = text.lower()
    text = re.sub(chosen_regex, '', text)
    tokens = word_tokenize(text)
    tokens = [word for word in tokens if word not in chosen_stop_words]
    tokens = [ps.stem(word) for word in tokens]
    return ' '.join(tokens)

train['content'] = train['content'].apply(preprocess_text_recommender)
test['content'] = test['content'].apply(preprocess_text_recommender)

# Vectorize and train set
vectorizer = TfidfVectorizer(max_features=chosen_max_features, token_pattern=cho
X_train = vectorizer.fit_transform(train['content'])
y_train = train['topic']
classifier = BernoulliNB().fit(X_train, y_train)

# predict topics on train and test
X_test = vectorizer.transform(test['content'])
train['pred_topic'] = classifier.predict(X_train)
test['pred_topic'] = classifier.predict(X_test)
```

1.2 Building User Profile

We will be building the user profile according to their music preferences in this section. To do so we will first take all the training songs from week 1 to 3 in given topics and mark as liked if the user's keywords for that topic matches the lyrics. Then we will fit the topic-specific TF-IDF model on all the songs in that topic, and transform them into one large document containing the lyrics of the liked songs concatenating into a single TF-IDF vector. With this process, the vector then become the user profile for that topic, highligthing the words occuring most often.

Each user’s profile is constructed from their liked songs in Weeks 1–3 as follows:

1. For each topic, select training songs whose predicted topic matches and whose lyrics contain at least one of the user's topic-specific keywords.
2. For each topic, train a `TfidfVectorizer()` on the preprocessed lyrics of all training songs predicted under that topic. This captures the term distributions characteristic of each topic.
3. Connect the preprocessed lyrics of the user's liked songs for each topic into a single document.
4. Use the corresponding topic's TF-IDF model to convert the document into a single profile vector. High weighted terms in this vector represent the words most characteristic of that user's preferences within the topic.

The resulting set of topic-specific TF-IDF vectors constitutes the user's content-based profile.

In [18]:
```python
# Function to build user profile based on liked songs
def build_profile(df, user_keywords):
    profile = {}
    for topic, keyword in user_keywords.items():
        liked_songs = df[
            (df['pred_topic'] == topic) &
            (df['lyrics']
               .str.lower()
               .apply(lambda t: any(word.lower() in t for word in keyword)))
        ]
        # Make record of all words of this topic
        all_words_of_topic = df[df['pred_topic'] == topic]['content'].tolist()
        if not all_words_of_topic:
            continue
        # fit tfidf on all songs predicted as this topic
        vectorizer = TfidfVectorizer(token_pattern=chosen_token_pattern, min_df=
        vectorizer.fit(all_words_of_topic)
        # build one big document containing all of liked songs
        all_liked_songs = " ".join(liked_songs['content'].tolist())
        profile_vector = vectorizer.transform([all_liked_songs])
        profile[topic] = (vectorizer, profile_vector)
    return profile

# Load user preferences from files
def load_user(file):
    processed_user = {}
    for line in open(file):
        topic, words = line.strip().split('\t')
        processed_user[topic] = words.split()
    return processed_user

user1 = load_user('user1.tsv')
user2 = load_user('user2.tsv')
user3 = {
    'dark': ['night','shadow','blood','fear','death'],
    'lifestyle': ['relax','fun','city','moment','freedom'],
    'personal': ['life','dream','heart','soul','goal'],
    'sadness': ['unrelevant','happy','cheerful','unrelated','recommender']
}

# build profiles
profile1 = build_profile(train, user1)
```

```python
profile2 = build_profile(train, user2)
profile3 = build_profile(train, user3)

# Function to print top 20 words in each topic of the user profile
def print_top_words(profile, label):
    print(f"\nTop words for {label}:")
    for topic, (tfidf, vector) in profile.items():
        array = vector.toarray().ravel()
        index = np.argsort(array)[::-1][:20]
        words = np.array(tfidf.get_feature_names_out())[index]
        print(f"\n{topic}:")
        print(", ".join(words))

print_top_words(profile1, 'User1')
print_top_words(profile2, 'User2')
print_top_words(profile3, 'User3')
```

```
Top words for User1:

dark:
fight, blood, come, fall, wall, like, know, tell, stand, na, hear, gon, peopl, st
art, steadi, follow, riot, home, kill, yeah

sadness:
regret, greater, think, leav, place, want, blame, hold, lord, word, chang, caus,
mind, trust, space, away, dream, suitcas, consequ, loos

personal:
abus, zayn, youth, younger, young, yesterday, yellow, year, yeah, wrong, write, a
maz, american, anderson, angel, answer, anybodi, anymor, apart, apolog

lifestyle:
sing, rhythm, song, like, feel, strong, radio, kingdom, wheel, girl, come, think,
tower, dial, midnight, letter, loud, write, know, eye

emotion:
good, feel, touch, miss, feelin, want, luck, go, hold, know, look, light, na, li
p, like, control, caus, right, babi, kiss

Top words for User2:

sadness:
break, heart, wave, silenc, crash, fall, like, spin, go, fade, away, leav, na, sc
ar, dark, learn, come, echo, flood, save

emotion:
lip, eas, fade, away, kiss, like, freez, dream, blow, burn, final, sink, pour, sh
ine, hop, breath, rain, hand, forget, blue

Top words for User3:

dark:
blood, fight, come, know, hear, fear, like, feel, na, time, live, look, light, go
n, hand, rais, stand, away, night, need

lifestyle:
sing, come, want, blue, right, wait, sky, play, know, rhythm, feel, song, tune, l
ike, think, listen, hear, need, music, light

personal:
life, live, na, world, wan, know, chang, yeah, reason, like, lord, time, thank, c
ome, good, teach, grind, dream, go, day

sadness:
ach, z, youth, young, yesterday, yellow, year, yeah, wrong, write, wreck, applau
s, archiv, arm, around, ask, asleep, attract, awak, away
```

1.3 User Profile Inspection

The top 20 terms in each topic profile generally align well with the intended themes. Despite the overall accuracy, there are some interesting trends and behaviours.

- User 1:
  - Dark: *fight*, *blood*, *follow*, and *kill*

    - Personal: terms such as *life*, *youth*, and *dream* dominate the list, despite minor noise from artist names like *abus* and *zayn*. Overall, the profile captures words of strong connection to the theme
- User 3:
    - For User 3, its **sadness** keywords were constructed to have unrelevant and contradicting words in which the algorithm searched Weeks 1–3 training songs for those whose predicted topic was sadness and whose lyrics contained at least one of these keywords. As expected, no songs satisfied both conditions and the code instead fit the TF-IDF vectorizer on all training songs whose predicted topic was sadness
    - The resulting top 20 words revealed high frequency sadness tokens like *youth*, *young*, *yesterday* which align with common lyrical themes of reflection and loss

To further reduce noise, each `TfidfVectorizer()` was configured with `min_df=2`, eliminating tokens that appear in fewer than two songs. This filter removed single letter noises and rare tokens, improving profile accuracy without sacrificing meaningful terms.

In conclusion the topic-specific TF-IDF profiles successfully highlight each user's interests. Despite minor noises, the main themes are clearly represented, and the profile well built for future recommendation.

2. Evaluating Performance with Chosen Metrics

With the user profiles constructed, the next step was to generate recommendations and assess their quality.

At the heart of the recommendation strategy is the **cosine similarity** metric. For each user, their profile is represented as a TF-IDF vector derived from the songs they previously liked. Cosine similarity was then calculated between this profile vector and the TF-IDF vector of every Week 4 song. Cosine similarity measures the angle between two high-dimensional vectors, so higher scores indicate greater content alignment with the user's established preferences.

Once all similarities are computed, the songs were sorted in descending order of similarity. The top N songs were designated as recommendations.

To evaluate how well these recommendations align with actual user interests, we use two standard metrics in recommender systems: **Precision@N** and **Recall@N**, which one focuses on relevance, and the other focuses on coverage:

Two standard metrics were employed to assess recommendation quality:

- **Precision@N** measures the proportion of recommended songs that were actually liked by the user. A high precision means the top N list is highly relevant
- **Recall@N** measures the proportion of all liked songs that were successfully captured in the top N. High recall signifies comprehensive coverage of the user's interests

Selecting **N** involves balancing relevance and coverage. A small N may yield high precision but low recall, and a large N can weaken relevance and overwhelm the user. To achieve a practical balance, **N = 10** was chosen.

```
In [19]: # Calculate similarity between songs and user profiles
records = []
for user_label, profiles in [
        ('User1', profile1),
        ('User2', profile2),
        ('User3', profile3)
    ]:
    for index, song in test.iterrows():
        topic = song['pred_topic']
        if topic not in profiles:
            continue
        tfidf_model, profile_vector = profiles[topic]
        song_vector = tfidf_model.transform([song['content']])
        similarity = cosine_similarity(song_vector, profile_vector)[0,0]
        records.append((user_label, index, similarity))

# Function to get top N recommendations for each user (preference to tutorial's
def get_top_n_recommendations(records, n=10):
    top_n = {}

    for uid, iid, score in records:
        top_n.setdefault(uid, []).append((iid, score))

    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = [iid for iid, _ in user_ratings[:n]]
    return top_n

top_n = get_top_n_recommendations(records, n=10)

# Function to get actual likes based on user profiles
actual_likes = {}
for user_label, keywords in [
        ('User1', user1),
        ('User2', user2),
        ('User3', user3)
    ]:
    for topic, keyword in keywords.items():
        # Select songs from the test set that match the predicted topic
        # and contain at least one of the user's keywords in the lyrics
        liked_songs = test[
            (test['pred_topic'] == topic) &
            (test['lyrics']
            .str.lower()
            .apply(lambda t: any(word.lower() in t for word in keyword)))
        ]
        actual_likes.setdefault(user_label, set()).update(liked_songs.index)

# Function to calculate precision and recall at N
def precision_recall_evaluation(top_n, actual_likes, n=10):
    rows = []
    for uid, user_ratings in top_n.items():
        true_ids = actual_likes.get(uid, set())
        true_likes = len(set(user_ratings) & true_ids)
        # The number of liked songs in the top N recommendations
        precision = true_likes / n
        # The number of liked songs in the actual likes
        if true_ids:
            recall = true_likes / len(true_ids)
        else:
            recall = 0.0
        rows.append({'User': uid, 'Precision@10': precision, 'Recall@10': recall
    return pd.DataFrame(rows)

print(precision_recall_evaluation(top_n, actual_likes, n=10))
```

```
    User  Precision@10  Recall@10
0  User1           0.6   0.153846
1  User2           0.1   0.125000
2  User3           1.0   0.043668
```

2.1 Evaluation of Final Performance

The table above reports Precision@10 and Recall@10 for each user:

- User 1: **Precision@10 = 0.60**, which means that 6 out of 10 recommended songs were liked. **Recall@10 = 0.1538**, meaning that approximately 15.4% of all liked songs appeared in the top-10 list. These results indicate a balanced trade-off between relevance and coverage.
- User 2: **Precision@10 = 0.10, Recall@10 = 0.1250**. The low scores reflect the limited keyword set in this user's profile, which reduced the system's ability to identify relevant songs.
- User 3: **Precision@10 = 1.00, Recall@10 = 0.0437**. This trend is interesting and means that all ten recommendations were liked, but they covered only a small fraction of the user's full set of liked songs. This pattern might have been resulted from the sadness keyword assignment in Section 1.2, which forced the model to fall back on all songs predicted as sadness, yielding very precise but narrowly focused recommendations. In other words the model was confident in what a typical sadness song look like and therefore recommending extremely relevant songs. However, because the keywords did not match any of the actual lyrics, almost no test songs were considered truly liked.

These outcomes demonstrate that recommendation quality focuses on profile specificity. Well-aligned profiles produce both high precision and reasonable recall, whereas vague or misleading profiles yield high precision but poor coverage. The cosine-similarity matching of TF-IDF profile vectors, evaluated through Precision@10 and Recall@10, provides a consistent and interpretable framework across diverse user scenarios.

Part 3 User Evaluation

A user study was conducted to assess the content-based recommender developed in Part 2. The dataset was divided into four chronological weeks similar to Part 2:

- Weeks 1–3 served as the training set
- Week 4 used for evaluation of recommendations

For each of Weeks 1–3, the subject was presented with **N = 10** randomly selected songs and asked to indicate which ones she liked. The liked songs from Weeks 1–3 were used to build the user profile through the **TF-IDF** and **cosine-similarity** pipeline described in Section 2. Finally, Week 4 recommendations, which are the top 10 by similarity, were generated and the subject again selected liked songs.

The subject is a friend, who has no knowledge of recommendation algorithms, to ensure unbiased, perception-driven feedback. **Precision@10** and **Recall@10** were computed, with precision serving as the primary metric. This is because that with real user, the subject is only exposed to recommended songs generated from Week 4 and has no access to other sons. Therefore, she isn't able to reflect on how many other songs from the dataset she might have liked and the recall value will always be 1. Nonetheless, qualitative feedback on satisfaction, diversity, and alignment with preferences was also collected.

1. Selecting Songs and Gathering User Feedbacks

In [20]:
```
# Load dataset and combine the coloumns into a single text column
df = pd.read_csv("dataset.tsv", sep='\t')
df = df.drop_duplicates().dropna()
categories = ['artist_name', 'track_name', 'release_date', 'genre', 'lyrics']
df['content'] = df[categories].astype(str).agg(' '.join, axis=1)

# Define sample size and sample songs for three weeks
N = 10
week1 = df.iloc[0:250].sample(N, random_state=42)
week2 = df.iloc[250:500].sample(N, random_state=42)
week3 = df.iloc[500:750].sample(N, random_state=42)

# Show user the sampled songs for Week 1-3
print("Week 1 Songs:")
print(week1.reset_index()[['index', 'track_name', 'artist_name']])
print("\nWeek 2 Songs:")
print(week2.reset_index()[['index', 'track_name', 'artist_name']])
print("\nWeek 3 Songs:")
print(week3.reset_index()[['index', 'track_name', 'artist_name']])
```

```
Week 1 Songs:
   index                         track_name        artist_name
0    142                   vivo hip hop (live)          skool 77
1      6                            trap door        rebelution
2     97                      outrunning karma      alec benjamin
3     60  we are come to outlive our brains             phish
4    112                    shout sister shout  madeleine peyroux
5    181                       what i could do     janiva magness
6    197                    if you met me first      eric ethridge
7    184                               natural    imagine dragons
8      9                            never land     eli young band
9    104                    john the revelator         larkin poe

Week 2 Songs:
   index                 track_name         artist_name
0    392                stupid deep         jon bellion
1    256                  black tar           the kills
2    347            the good doctor              haken
3    310              american tune      allen toussaint
4    362            i have this hope  tenth avenue north
5    431  heaven falls / fall on me            surfaces
6    447        devils got you beat      blues saraceno
7    434                 this is it   the wood brothers
8    259                 tesselation      mild high club
9    354              snake charmer         parov stelar

Week 3 Songs:
   index                  track_name     artist_name
0    643                    the fear       the score
1    506            buy my own drinks    runaway june
2    597                    get happy    goodbye june
3    560             head above water   avril lavigne
4    613                   tie me up        shenseea
5    683          times won’t change me     circa waves
6    699                ...a livication       iya terra
7    686                      church    fall out boy
8    509      you’re worthy of it all   shane & shane
9    605  the dark horse always wins  blues saraceno
```

After the user listening to the songs shown above, she chose the song she liked and marked down the index number of the song in following dictionary.

In [21]:
```python
# Create a dictionary to store user feedback
user_liked_songs = {
    'Week1': [97, 184],
    'Week2': [310, 447, 259],
    'Week3': [643, 699, 686]
}
```

2. Building User Profile

In [22]:
```python
# Combine liked song from Weeks 1-3
liked_song_combined = (
    user_liked_songs['Week1'] +
    user_liked_songs['Week2'] +
    user_liked_songs['Week3']
)
liked_songs_df = df.loc[liked_song_combined].copy()
```

```
# Preprocess the liked songs
liked_songs_df['content'] = liked_songs_df['content'].apply(preprocess_text_reco
train = df.iloc[:750].copy()
train['content'] = train['content'].apply(preprocess_text_recommender)
vectorizer = TfidfVectorizer(max_features=chosen_max_features, token_pattern=cho
vectorizer.fit(train['content'])

# Combine all liked song content into one document
liked_text = ' '.join(liked_songs_df['content'].tolist())
user_profile = vectorizer.transform([liked_text])

# Preprocess the test set and calculate similarities
test = df.iloc[750:1000].copy()
test['content'] = test['content'].apply(preprocess_text_recommender)
X_test  = vectorizer.transform(test['content'])
similarities = cosine_similarity(X_test, user_profile).ravel()
test['similarity'] = similarities

# Get top 10 recommended songs
top_n_recs = test.sort_values(by='similarity', ascending=False).head(10)
print(top_n_recs[['track_name', 'artist_name']])
```

```
                track_name       artist_name
881          boy in the bubble      alec benjamin
823    the devil don't sleep  brantley gilbert
942            whiskey fever           dorothy
913       pressure under fire         gov't mule
926              sit awhile    the band steele
1003                 hayati         tinariwen
799            do your worst        rival sons
879                our town         tyler farr
795          it's only right            wallows
902              moon river       nicole henry
```

Then again, the songs above were shown to the user and her liked songs were marked and recorded below.

In [23]:
```
liked_songs_recommended = [881, 902, 1003, 799]
```

3. Evaluating Recommendations

In [24]:
```
# Calculate precision and recall for the top 10 recommendations
predicted_top10 = top_n_recs.index.tolist()
actual_likes = set(liked_songs_recommended)
true_likes = len(set(predicted_top10) & actual_likes)
precision = true_likes / 10
if actual_likes:
    recall = true_likes / len(actual_likes)
else:
    recall = 0.0

print(f"Precision@10: {precision}")
print(f"Recall@10: {recall}")
```

```
Precision@10: 0.4
Recall@10: 1.0
```

4. Conclusion and Real User Feedbacks

The user evaluation yielded a **Precision@10 of 0.40**, indicating that 4 out of the 10 recommended songs were actually liked. This level of relevance is encouraging for a purely content-based system operating on a small data size. The **Recall@10 reached 1.0**, which was anticipated given that the user could only evaluate the ten songs presented. Unlike Part 2, where recall was computed against a broader set of liked songs using keyword matching, this live test had a constrained recommendation set, therfore inflating recall.

Throughout the experiment, there were some interesting behaviors obeserved from the subject. During Week 1, the user instantly recognized “Outrunning Karma” by Alec Benjamin as which was her favorate artist and song. In Week 4, another Alec Benjamin track appeared among the top recommendations and was likewise liked. Although no explicit artist data was used, the TF-IDF representation might implicitly captured patterns associated with that artist, suggesting emergent personalization beyond topic matching.

During an small interview of the subject, when asked whether lyrics alone would be enough for selection and no audio provided, the user expressed uncertainty. This underscores an important limitation on content-based recommendation system. Musical attributes like play a critical role in listener preference and are not captured in content-based models. So therefore, it is clear that in real user interview, the results could be biased or highly driven by external sources.

Overall, the content-based recommender demonstrated strong precision given limited feedback, while the qualitative feedback highlighted both its strengths and its weaknesses. Future enhancements, such as integrating acoustic features, user collaborative signals, or even larger and more colorful datasets, could further improve recommendation quality and user satisfaction.