

Part 1 Topic Classification

1

The first issue is related to the inappropriate regular expressions used during the text preprocessing process. The regular expressions in the tutorial may delete all non-alphanumeric characters, but this approach may mistakenly delete punctuation marks and important words with emotional connotations in the lyrics. To avoid this situation, I am considering using a more gentle regular expression method. Ensure that the text structure can be retained, and then use methods such as part-of-speech tagging and word segmentation to reduce the number of incorrect deletions. The second issue concerns the robustness of the model. The teacher provided a single training set and test set division. When the data volume is relatively small, the performance of the model is very susceptible to the influence of data division bias. To solve this problem, I chose to use the K-fold cross-validation provided by scikit-learn (such as 5-fold or 10-fold). The advantage of this method is that by repeatedly training and testing the model on multiple subsets of data to summarize the average performance indicators of each fold, it provides a more reliable measure of the model's robustness. At the same time, it can also avoid the performance impact caused by the imbalance of the dataset when dividing it in a single instance. I will apply the improved solution in the subsequent models and explain the impact.

2

```
In [1]: import pandas as pd
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
import nltk

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')

stop_words_nltk = set(stopwords.words('english'))
stop_words_sklearn = ENGLISH_STOP_WORDS

lemmatizer = WordNetLemmatizer()
stemmer = PorterStemmer()
```

```

def clean_solution1(text):
    return re.sub(r"[^a-zA-Z0-9\s]", '', text.lower())

def clean_solution2(text):
    return re.sub(r"[^a-zA-Z0-9\s!?]", '', text.lower()) # Preserve emotional sys

def preprocess(text, clean_type, use_stemmer=False, use_nltk_stopwords=True):
    if clean_type == 1:
        text = clean_solution1(text)
    else:
        text = clean_solution2(text)

    words = word_tokenize(text)

    stop_words = stop_words_nltk if use_nltk_stopwords else stop_words_sklearn

    if use_stemmer:
        words = [stemmer.stem(w) for w in words if w not in stop_words]
    else:
        words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]

    return ' '.join(words)

df = pd.read_csv('dataset.tsv', sep='\t')

for clean_type in [1, 2]:
    for use_stemmer in [True, False]:
        for use_nltk_stopwords in [True, False]:
            df['processed_lyrics'] = df['lyrics'].apply(preprocess, clean_type=clean_type,
                                              use_stemmer=use_stemmer,
                                              use_nltk_stopwords=use_nltk_stopwords)

            print(f"\nCleaning strategy {clean_type}, {'Stemming' if use_stemmer else 'Using NLTK stopwords' if use_nltk_stopwords else 'Using sc'")

            X = df['processed_lyrics']
            y = df['topic']

            X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

            pipeline = Pipeline([
                ('vectorizer', CountVectorizer(max_features=1000)),
                ('classifier', MultinomialNB())
            ])

            pipeline.fit(X_train, y_train)
            y_pred = pipeline.predict(X_test)

            print(f'Accuracy: {accuracy_score(y_test, y_pred)}')
            print(f'Precision: {precision_score(y_test, y_pred, average="weighted")}')
            print(f'Recall: {recall_score(y_test, y_pred, average="weighted")}')
            print(f'F1 Score: {f1_score(y_test, y_pred, average="weighted")}')



```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\97539\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\97539\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\97539\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

```
Cleaning strategy 1, Stemming, Using NLTK stopwords:
Accuracy: 0.8533333333333334
Precision: 0.8554555555555555
Recall: 0.8533333333333334
F1 Score: 0.8535935602921905
```

```
Cleaning strategy 1, Stemming, Using scikit-learn stopwords:
Accuracy: 0.8466666666666667
Precision: 0.8475665905631659
Recall: 0.8466666666666667
F1 Score: 0.8466908398377458
```

```
Cleaning strategy 1, Lemmatization, Using NLTK stopwords:
Accuracy: 0.8666666666666667
Precision: 0.8684574988874054
Recall: 0.8666666666666667
F1 Score: 0.8666520670362776
```

```
Cleaning strategy 1, Lemmatization, Using scikit-learn stopwords:
Accuracy: 0.85
Precision: 0.8510185185185185
Recall: 0.85
F1 Score: 0.8500029332317648
```

```
Cleaning strategy 2, Stemming, Using NLTK stopwords:
Accuracy: 0.8533333333333334
Precision: 0.8554555555555555
Recall: 0.8533333333333334
F1 Score: 0.8535935602921905
```

```
Cleaning strategy 2, Stemming, Using scikit-learn stopwords:
Accuracy: 0.8466666666666667
Precision: 0.8475665905631659
Recall: 0.8466666666666667
F1 Score: 0.8466908398377458
```

```
Cleaning strategy 2, Lemmatization, Using NLTK stopwords:
Accuracy: 0.8666666666666667
Precision: 0.8684574988874054
Recall: 0.8666666666666667
F1 Score: 0.8666520670362776
```

```
Cleaning strategy 2, Lemmatization, Using scikit-learn stopwords:
Accuracy: 0.85
Precision: 0.8510185185185185
Recall: 0.85
F1 Score: 0.8500029332317648
```

Choose to retain symbols related to emotions such as exclamation marks and question marks. Then, use the word_tokenize method from nltk, along with the built-in stopword

list of nltk. Finally, use the method of lemmatization to determine the best result.

3

```
In [2]: import pandas as pd
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
from sklearn.pipeline import Pipeline
import numpy as np
import nltk

stop_words_nltk = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def clean_solution1(text):
    return re.sub(r"^[a-zA-Z0-9\$\!\?]", ' ', text.lower())

def preprocess(text, use_stemmer=False, use_nltk_stopwords=True):
    text = clean_solution1(text)

    words = word_tokenize(text)

    if use_stemmer:
        words = [stemmer.stem(w) for w in words if w not in stop_words_nltk]
        words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words_nltk]

    return ' '.join(words)

df = pd.read_csv('dataset.tsv', sep='\t')

print("Raw data sample:")
print(df.head())

df['processed_lyrics'] = df['lyrics'].apply(preprocess, use_stemmer=False, use_nltk_stopwords=True)

X = df['processed_lyrics']
y = df['topic']

models = {
    "MNB": MultinomialNB(),
    "BNB": BernoulliNB()
}

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

results = {}
```

```

for model_name, model in models.items():

    accuracies = []
    precision_scores = []
    recall_scores = []
    f1_scores = []

    for train_index, test_index in kf.split(X, y):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        X_train_processed = X_train.apply(preprocess, use_stemmer=False, use_nlt
        X_test_processed = X_test.apply(preprocess, use_stemmer=False, use_nltk_


        vectorizer = CountVectorizer(max_features=1000)
        X_train_vec = vectorizer.fit_transform(X_train_processed)
        X_test_vec = vectorizer.transform(X_test_processed)

        model.fit(X_train_vec, y_train)
        y_pred = model.predict(X_test_vec)

        accuracies.append(accuracy_score(y_test, y_pred))
        precision_scores.append(precision_score(y_test, y_pred, average="weighted"))
        recall_scores.append(recall_score(y_test, y_pred, average="weighted"))
        f1_scores.append(f1_score(y_test, y_pred, average="weighted"))

    results[model_name] = {
        "accuracy": np.mean(accuracies),
        "precision": np.mean(precision_scores),
        "recall": np.mean(recall_scores),
        "f1_score": np.mean(f1_scores)
    }

print("\nModel Comparison Results:")
for model_name, metrics in results.items():
    print(f"\n{model_name} Model:")
    for metric, score in metrics.items():
        print(f"{metric}: {score:.4f}")

```

Raw data sample:

```
artist_name      track_name release_date \
0              loving    the not real lake        2016
1                incubus   into the summer       2019
2            reignwolf     hardcore        2016
3  tedeschi trucks band      anyhow        2016
4  lukas nelson and promise of the real if i started over  2017

genre                      lyrics      topic
0  rock  awake know go see time clear world mirror worl...  dark
1  rock  shouldn summer pretty build spill ready overfl... lifestyle
2 blues  lose deep catch breath think say try break wal... sadness
3 blues  run bitter taste take rest feel anchor soul pl... sadness
4 blues  think think different set apart sober mind sym...  dark
```

Model Comparison Results:

MNB Model:

```
accuracy: 0.8493
precision: 0.8516
recall: 0.8493
f1_score: 0.8488
```

BNB Model:

```
accuracy: 0.6507
precision: 0.6357
recall: 0.6507
f1_score: 0.6368
```

```
In [3]: import matplotlib.pyplot as plt

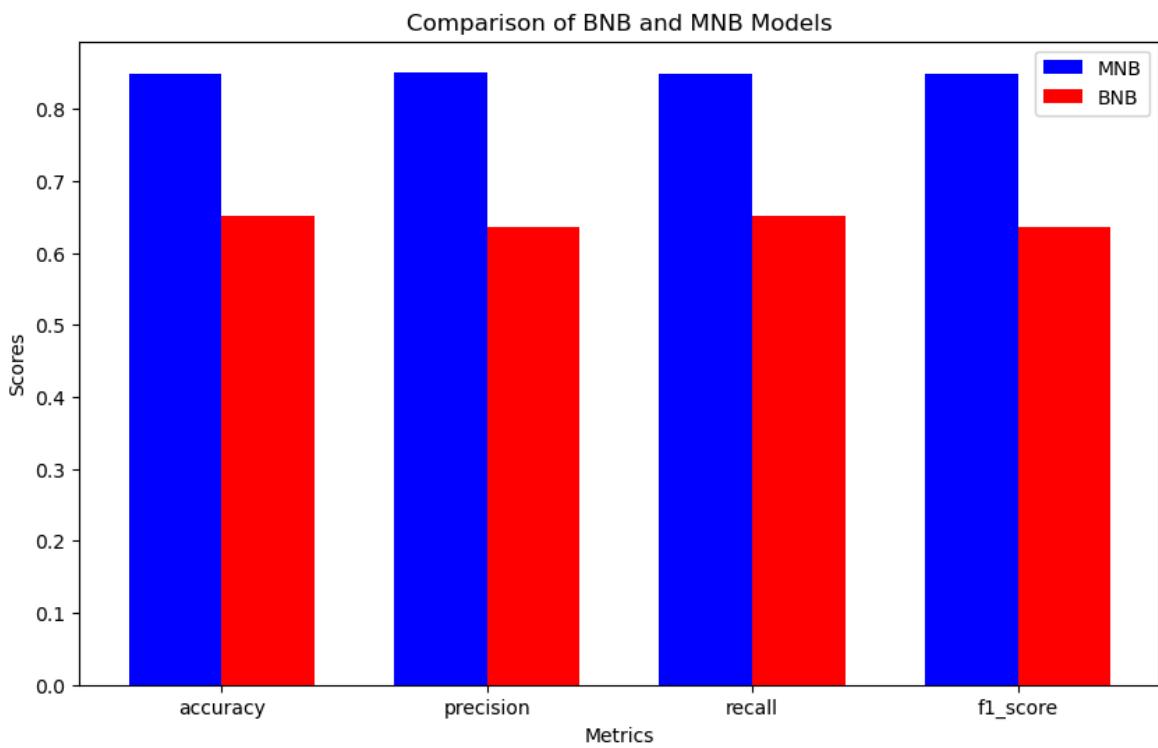
metrics = ['accuracy', 'precision', 'recall', 'f1_score']
mnb_scores = [results['MNB'][metric] for metric in metrics]
bnb_scores = [results['BNB'][metric] for metric in metrics]

x = np.arange(len(metrics))
width = 0.35

fig, ax = plt.subplots(figsize=(10, 6))
rects1 = ax.bar(x - width/2, mnb_scores, width, label='MNB', color='b')
rects2 = ax.bar(x + width/2, bnb_scores, width, label='BNB', color='r')

ax.set_xlabel('Metrics')
ax.set_ylabel('Scores')
ax.set_title('Comparison of BNB and MNB Models')
ax.set_xticks(x)
ax.set_xticklabels(metrics)
ax.legend()

plt.show()
```



This bar chart shows that MNB outperforms BNB in terms of accuracy, precision, recall rate and F1 score.

4

```
In [4]: import pandas as pd
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
import numpy as np
import nltk
import matplotlib.pyplot as plt

lemmatizer = WordNetLemmatizer()

stop_words = set(stopwords.words('english'))

def clean_solution1(text):
    return re.sub(r"[^a-zA-Z0-9\s!?]", ' ', text.lower())

def preprocess(text, use_nltk_stopwords=True):
    text = clean_solution1(text)

    words = word_tokenize(text)

    words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]

    return ' '.join(words)
```

```

df = pd.read_csv('dataset.tsv', sep='\t')

X = df['lyrics']
y = df['topic']

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

N_values = [100, 200, 300, 400, 500]
results = {
    "BNB_accuracy": {N: [] for N in N_values},
    "BNB_f1_score": {N: [] for N in N_values},
    "MNB_accuracy": {N: [] for N in N_values},
    "MNB_f1_score": {N: [] for N in N_values},
}

for N in N_values:
    for model_name, model in [("BNB", BernoulliNB()), ("MNB", MultinomialNB())]:
        accuracies = []
        f1_scores = []

        for train_index, test_index in kf.split(X, y):
            X_train, X_test = X.iloc[train_index], X.iloc[test_index]
            y_train, y_test = y.iloc[train_index], y.iloc[test_index]

            X_train_processed = X_train.apply(preprocess, use_nltk_stopwords=True)
            X_test_processed = X_test.apply(preprocess, use_nltk_stopwords=True)

            vectorizer = CountVectorizer(max_features=N)
            X_train_vec = vectorizer.fit_transform(X_train_processed)
            X_test_vec = vectorizer.transform(X_test_processed)

            model.fit(X_train_vec, y_train)
            y_pred = model.predict(X_test_vec)

            accuracies.append(accuracy_score(y_test, y_pred))
            f1_scores.append(f1_score(y_test, y_pred, average="weighted"))

        if model_name == "BNB":
            results["BNB_accuracy"][N].append(np.mean(accuracies))
            results["BNB_f1_score"][N].append(np.mean(f1_scores))
        else:
            results["MNB_accuracy"][N].append(np.mean(accuracies))
            results["MNB_f1_score"][N].append(np.mean(f1_scores))

plt.figure(figsize=(10, 6))
plt.plot(N_values, [np.mean(results["BNB_accuracy"])[N]] for N in N_values, label="BNB Accuracy")
plt.plot(N_values, [np.mean(results["BNB_f1_score"])[N]] for N in N_values, label="BNB F1 Score")
plt.plot(N_values, [np.mean(results["MNB_accuracy"])[N]] for N in N_values, label="MNB Accuracy")
plt.plot(N_values, [np.mean(results["MNB_f1_score"])[N]] for N in N_values, label="MNB F1 Score")

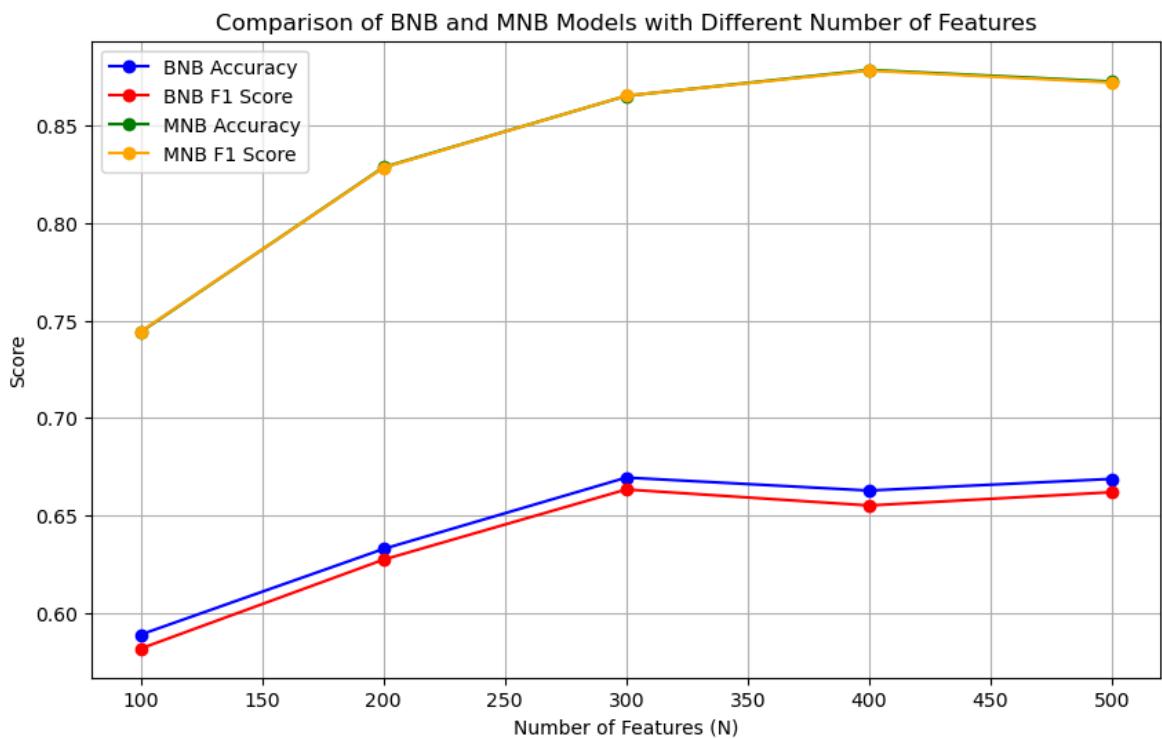
plt.xlabel('Number of Features (N)')
plt.ylabel('Score')
plt.title('Comparison of BNB and MNB Models with Different Number of Features')
plt.legend()
plt.grid(True)
plt.show()

```

```

print("Results Summary:")
for N in N_values:
    print(f"\nN = {N}:")
    print(f"  BNB Average Accuracy: {np.mean(results['BNB_accuracy'][N]):.4f}")
    print(f"  BNB Average F1 Score: {np.mean(results['BNB_f1_score'][N]):.4f}")
    print(f"  MNB Average Accuracy: {np.mean(results['MNB_accuracy'][N]):.4f}")
    print(f"  MNB Average F1 Score: {np.mean(results['MNB_f1_score'][N]):.4f}")

```



Results Summary:

N = 100:
BNB Average Accuracy: 0.5887
BNB Average F1 Score: 0.5815
MNB Average Accuracy: 0.7440
MNB Average F1 Score: 0.7442

N = 200:
BNB Average Accuracy: 0.6327
BNB Average F1 Score: 0.6272
MNB Average Accuracy: 0.8287
MNB Average F1 Score: 0.8285

N = 300:
BNB Average Accuracy: 0.6693
BNB Average F1 Score: 0.6632
MNB Average Accuracy: 0.8653
MNB Average F1 Score: 0.8654

N = 400:
BNB Average Accuracy: 0.6627
BNB Average F1 Score: 0.6550
MNB Average Accuracy: 0.8787
MNB Average F1 Score: 0.8783

N = 500:
BNB Average Accuracy: 0.6687
BNB Average F1 Score: 0.6618
MNB Average Accuracy: 0.8727
MNB Average F1 Score: 0.8722

Taking all factors into consideration, since the MNB model performs better than the BNB model, and when n equals 400, the Accuracy and F1 Score of the MNB model are the highest, therefore, 400 is chosen.

5

```
In [5]: import pandas as pd
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn.metrics import accuracy_score, f1_score
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
from sklearn.linear_model import LogisticRegression
import numpy as np
import nltk

lemmatizer = WordNetLemmatizer()

stop_words = set(stopwords.words('english'))

def clean_solution1(text):
    return re.sub(r"[^a-zA-Z0-9\s!?]", ' ', text.lower())
```

```

def preprocess(text, use_nltk_stopwords=True):
    text = clean_solution1(text)

    words = word_tokenize(text)

    words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]
    return ' '.join(words)

df = pd.read_csv('dataset.tsv', sep='\t')

X = df['lyrics']
y = df['topic']
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

N = 400

results = {
    "BNB_accuracy": [],
    "BNB_f1_score": [],
    "MNB_accuracy": [],
    "MNB_f1_score": [],
    "LogReg_accuracy": [],
    "LogReg_f1_score": [],
}
param_grid = {'C': [0.01, 0.1, 1, 10, 100]}

logreg = LogisticRegression(max_iter=1000, class_weight='balanced')

for model_name, model in [("BNB", BernoulliNB()), ("MNB", MultinomialNB()), ("Lo
    accuracies = []
    f1_scores = []

    best_C = None

    for train_index, test_index in kf.split(X, y):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        X_train_processed = X_train.apply(preprocess, use_nltk_stopwords=True)
        X_test_processed = X_test.apply(preprocess, use_nltk_stopwords=True)

        vectorizer = CountVectorizer(max_features=N)
        X_train_vec = vectorizer.fit_transform(X_train_processed)
        X_test_vec = vectorizer.transform(X_test_processed)

        if model_name == "LogReg":
            grid_search = GridSearchCV(model, param_grid, cv=5)
            grid_search.fit(X_train_vec, y_train)

            if best_C is None:
                best_C = grid_search.best_params_['C']
                print(f"Best C value for Logistic Regression: {best_C}")

            best_model = grid_search.best_estimator_
            y_pred = best_model.predict(X_test_vec)
        else:
            model.fit(X_train_vec, y_train)
            y_pred = model.predict(X_test_vec)
    accuracies.append(best_model.score(X_test, y_test))
    f1_scores.append(f1_score(y_test, y_pred))
}

```

```

    accuracies.append(accuracy_score(y_test, y_pred))
    f1_scores.append(f1_score(y_test, y_pred, average="weighted"))

    if model_name == "BNB":
        results["BNB_accuracy"].append(np.mean(accuracies))
        results["BNB_f1_score"].append(np.mean(f1_scores))
    elif model_name == "MNB":
        results["MNB_accuracy"].append(np.mean(accuracies))
        results["MNB_f1_score"].append(np.mean(f1_scores))
    elif model_name == "LogReg":
        results["LogReg_accuracy"].append(np.mean(accuracies))
        results["LogReg_f1_score"].append(np.mean(f1_scores))

print("Results Summary:")
print(f"BNB Average Accuracy: {np.mean(results['BNB_accuracy']):.4f}")
print(f"BNB Average F1 Score: {np.mean(results['BNB_f1_score']):.4f}")
print(f"MNB Average Accuracy: {np.mean(results['MNB_accuracy']):.4f}")
print(f"MNB Average F1 Score: {np.mean(results['MNB_f1_score']):.4f}")
print(f"LogReg Average Accuracy: {np.mean(results['LogReg_accuracy']):.4f}")
print(f"LogReg Average F1 Score: {np.mean(results['LogReg_f1_score']):.4f}")

```

Best C value for Logistic Regression: 0.1

Results Summary:

```

BNB Average Accuracy: 0.6627
BNB Average F1 Score: 0.6550
MNB Average Accuracy: 0.8787
MNB Average F1 Score: 0.8783
LogReg Average Accuracy: 0.8807
LogReg Average F1 Score: 0.8808

```

I will consider logistic regression as another machine learning method to compare with BNB and MNB. Because logistic regression is widely used as a linear model in text classification tasks, and is particularly suitable for binary classification tasks, it can handle parameter tuning very well and deal with various classification problems effectively.

Part 2 Recommendation Methods

1

In [6]:

```

import pandas as pd
import re
import numpy as np
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from collections import defaultdict

lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

def clean_text_1(text):
    return re.sub(r"[^a-zA-Z0-9\s!?]", ' ', text.lower())

```

```

def preprocess(text, use_nltk_stopwords=True):
    text = clean_text_1(text)
    words = word_tokenize(text)
    words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]
    return ' '.join(words)

df = pd.read_csv('dataset.tsv', sep='\t')
df['processed_lyrics'] = df['lyrics'].apply(preprocess)

user1_df = pd.read_csv('user1.tsv', sep='\t')
user2_df = pd.read_csv('user2.tsv', sep='\t')
user3_df = pd.read_csv('user3.tsv', sep='\t')

user1_interests = {row['topic']: row['keywords'].split(', ') for _, row in user1_df.iterrows()}
user2_interests = {row['topic']: row['keywords'].split(', ') for _, row in user2_df.iterrows()}
user3_interests = {row['topic']: row['keywords'].split(', ') for _, row in user3_df.iterrows()}

train_df = df.iloc[:750]
test_df = df.iloc[750:]

vectorizer = TfidfVectorizer(max_features=400)
X_train = vectorizer.fit_transform(train_df['processed_lyrics'])
X_test = vectorizer.transform(test_df['processed_lyrics'])

def simulate_user_likes(user_interests, test_df, similarity_threshold=0.1):
    all_keywords = []
    for topic, keywords in user_interests.items():
        all_keywords.extend(keywords)

    user_profile_text = ' '.join(all_keywords)
    user_profile_processed = preprocess(user_profile_text)
    user_profile_tfidf = vectorizer.transform([user_profile_processed])

    similarities = cosine_similarity(user_profile_tfidf, X_test).flatten()

    liked_songs_by_topic = defaultdict(list)
    user_topics = set(user_interests.keys())

    for idx, (_, row) in enumerate(test_df.iterrows()):
        song_topic = row['topic']
        song_similarity = similarities[idx]

        if song_topic in user_topics and song_similarity > similarity_threshold:
            liked_songs_by_topic[song_topic].append({
                'index': idx,
                'similarity': song_similarity,
                'song_data': row
            })

    return liked_songs_by_topic

def build_user_profile_from_likes(liked_songs_by_topic, top_n=20):
    user_profile = {}

    for topic, liked_songs in liked_songs_by_topic.items():
        if not liked_songs:
            user_profile[topic] = []
            continue

        sorted_songs = sorted(liked_songs, key=lambda x: x['similarity'], reverse=True)
        top_n_songs = sorted_songs[:top_n]
        user_profile[topic] = [song['song_data'] for song in top_n_songs]

```

```

song_indices = [song['index'] for song in liked_songs]

liked_songs_tfidf = X_test[song_indices]

mean_tfidf_scores = np.mean(liked_songs_tfidf.toarray(), axis=0)

# Get top keywords
feature_names = vectorizer.get_feature_names_out()
top_keyword_indices = mean_tfidf_scores.argsort()[-top_n:][:-1]
recommended_keywords = [feature_names[i] for i in top_keyword_indices
                        if mean_tfidf_scores[i] > 0]

user_profile[topic] = recommended_keywords

return user_profile

# Build user profiles
users_data = {
    'User1': user1_interests,
    'User2': user2_interests,
    'User3': user3_interests
}

for user_name, user_interests in users_data.items():
    print(f"== Building profile for {user_name} ==")

    # Calculate the songs that the users actually like
    liked_songs_by_topic = simulate_user_likes(user_interests, test_df, similarity)

    # Display the songs that users like according to the theme simultaneously
    total_liked_songs = 0
    for topic in user_interests.keys():
        if topic in liked_songs_by_topic:
            num_liked = len(liked_songs_by_topic[topic])
        else:
            num_liked = 0
        total_liked_songs += num_liked
        print(f"  {topic}: {num_liked} liked songs")

    print(f"  Total: {total_liked_songs} liked songs")
    print()

    # Build profile from actually Liked songs
    user_profile = build_user_profile_from_likes(liked_songs_by_topic)

    print(f"{user_name}:")
    for topic in user_interests.keys():
        keywords = user_profile.get(topic, [])
        print(f"  {topic}:")
        if keywords and len(keywords) > 0:
            print(f"    Top 20 keywords: {', '.join(keywords)}")
        else:
            print(f"    Top 20 keywords: No matching keywords found")
    print()

```

```
==== Building profile for User1 ====
  dark: 30 liked songs
  sadness: 33 liked songs
  personal: 33 liked songs
  lifestyle: 15 liked songs
  emotion: 15 liked songs
  Total: 126 liked songs
```

```
User1:
  dark:
    Top 20 keywords: fight, truth, hide, night, eye, life, enemy, cold, world, dr
eam, pain, cry, know, light, come, feel, burn, right, leave, fall
```

```
  sadness:
    Top 20 keywords: cry, break, heart, come, dream, tear, know, feel, night, awa
y, hold, inside, cause, leave, memory, like, wish, take, apart, truth
```

```
  personal:
    Top 20 keywords: life, dream, world, live, yeah, memory, today, know, feel, l
ook, believe, fight, better, like, day, love, come, open, time, year
```

```
  lifestyle:
    Top 20 keywords: night, right, summer, hear, wrong, feel, cry, black, come, y
eah, head, whoa, place, like, cold, year, morning, ready, light, tonight
```

```
  emotion:
    Top 20 keywords: feel, kiss, love, good, baby, like, know, lover, na, reason,
time, heart, darling, light, real, fly, want, right, away, steal
```

```
==== Building profile for User2 ====
```

```
  sadness: 10 liked songs
  emotion: 9 liked songs
  Total: 19 liked songs
```

```
User2:
  sadness:
    Top 20 keywords: memory, goodbye, tear, wish, hold, dance, piece, leave, cal
l, heart, good, hurt, want, lock, away, gon, come, lose, baby, time
```

```
  emotion:
    Top 20 keywords: kiss, close, touch, darling, feel, time, lover, heart, lip,
want, steal, girl, fly, care, little, miss, beat, meet, hear, like
```

```
==== Building profile for User3 ====
```

```
  dark: 18 liked songs
  sadness: 19 liked songs
  personal: 25 liked songs
  Total: 62 liked songs
```

```
User3:
  dark:
    Top 20 keywords: fight, enemy, dream, know, taste, inside, awake, cry, yeah,
hide, body, yesterday, lead, world, away, hear, ghost, oooh, burn, speak
```

```
  sadness:
    Top 20 keywords: cry, awake, break, dream, heart, come, tear, apart, time, fa
ll, hold, away, take, gon, cause, like, tomorrow, inside, eye, na
```

```
  personal:
    Top 20 keywords: life, dream, live, yesterday, world, know, come, fight, bett
```

er, people, good, commercial, lyric, friend, time, yeah, stop, change, future, real

It seems that these keywords are reasonable overall, with only a few exceptions.

2

```
In [7]:  
import pandas as pd  
import re  
import numpy as np  
from nltk.tokenize import word_tokenize  
from nltk.corpus import stopwords  
from nltk.stem import WordNetLemmatizer  
from sklearn.feature_extraction.text import TfidfVectorizer  
from sklearn.metrics.pairwise import cosine_similarity  
from collections import defaultdict  
import warnings  
warnings.filterwarnings('ignore')  
  
lemmatizer = WordNetLemmatizer()  
stop_words = set(stopwords.words('english'))  
  
def clean_solution1(text):  
    return re.sub(r"[^a-zA-Z0-9\s!?]", ' ', text.lower())  
  
def preprocess(text, use_nltk_stopwords=True):  
    text = clean_solution1(text)  
    words = word_tokenize(text)  
    words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]  
    return ' '.join(words)  
  
df = pd.read_csv('dataset.tsv', sep='\t')  
df['processed_lyrics'] = df['lyrics'].apply(preprocess)  
  
user1_df = pd.read_csv('user1.tsv', sep='\t')  
user2_df = pd.read_csv('user2.tsv', sep='\t')  
user3_df = pd.read_csv('user3.tsv', sep='\t')  
  
user1_interests = {row['topic']: row['keywords'].split(', ') for _, row in user1_df.iterrows()}  
user2_interests = {row['topic']: row['keywords'].split(', ') for _, row in user2_df.iterrows()}  
user3_interests = {row['topic']: row['keywords'].split(', ') for _, row in user3_df.iterrows()}  
  
# According to the requirements of the question, splitting train and test set  
train_df = df.iloc[:750]  
test_df = df.iloc[750:].reset_index(drop=True)  
  
vectorizer = TfidfVectorizer(max_features=400)  
X_train = vectorizer.fit_transform(train_df['processed_lyrics'])  
X_test = vectorizer.transform(test_df['processed_lyrics'])  
  
def simulate_user_likes(user_interests, test_df, similarity_threshold=0.1):  
    all_keywords = []  
    for topic, keywords in user_interests.items():  
        all_keywords.extend(keywords)  
  
    user_profile_text = ' '.join(all_keywords)  
    user_profile_processed = preprocess(user_profile_text)
```

```

user_profile_tfidf = vectorizer.transform([user_profile_processed])

similarities = cosine_similarity(user_profile_tfidf, X_test).flatten()

liked_songs_by_topic = defaultdict(list)
user_topics = set(user_interests.keys())

for idx, (_, row) in enumerate(test_df.iterrows()):
    song_topic = row['topic']
    song_similarity = similarities[idx]

    if song_topic in user_topics and song_similarity > similarity_threshold:
        liked_songs_by_topic[song_topic].append({
            'index': idx,
            'similarity': song_similarity,
            'song_data': row
        })

return liked_songs_by_topic

def build_user_profile_from_likes(liked_songs_by_topic, top_n=20):
    user_profile = {}

    for topic, liked_songs in liked_songs_by_topic.items():
        if not liked_songs:
            user_profile[topic] = []
            continue

        song_indices = [song['index'] for song in liked_songs]
        liked_songs_tfidf = X_test[song_indices]
        mean_tfidf_scores = np.mean(liked_songs_tfidf.toarray(), axis=0)

        feature_names = vectorizer.get_feature_names_out()
        top_keyword_indices = mean_tfidf_scores.argsort()[-top_n:][::-1]
        recommended_keywords = [feature_names[i] for i in top_keyword_indices
                               if mean_tfidf_scores[i] > 0]

        user_profile[topic] = recommended_keywords

    return user_profile

def recommend_songs_method1_global_tfidf(user_profile, test_df, n_per_topic=10):
    recommendations = {}
    for topic, keywords in user_profile.items():
        if not keywords:
            recommendations[topic] = []
            continue

        profile_text = ' '.join(keywords)
        profile_vector = vectorizer.transform([profile_text])

        topic_songs = test_df[test_df['topic'] == topic].reset_index(drop=True)
        if len(topic_songs) == 0:
            recommendations[topic] = []
            continue

        topic_indices = test_df[test_df['topic'] == topic].index
        topic_vectors = X_test[topic_indices]

        similarities = cosine_similarity(profile_vector, topic_vectors).flatten()

```

```

        top_indices = similarities.argsort()[-n_per_topic:][::-1]

        recommended_songs = []
        for i in top_indices:
            if i < len(topic_songs):
                recommended_songs.append({
                    'song_index': topic_indices[i],
                    'similarity': similarities[i],
                    'song_data': topic_songs.iloc[i]
                })
        recommendations[topic] = recommended_songs

    return recommendations

def recommend_songs_method2_topic_specific(user_profile, test_df, n_per_topic=10):
    recommendations = {}

    for topic, keywords in user_profile.items():
        if not keywords:
            recommendations[topic] = []
            continue

        # Use topic-specific training data to build specialized vectorizer
        topic_train_songs = train_df[train_df['topic'] == topic]
        topic_test_songs = test_df[test_df['topic'] == topic]

        if len(topic_test_songs) == 0:
            recommendations[topic] = []
            continue

        topic_train_lyrics = topic_train_songs['processed_lyrics'].tolist()
        if len(topic_train_lyrics) == 0:
            recommendations[topic] = []
            continue

        topic_vectorizer = TfidfVectorizer(max_features=150, min_df=1, max_df=0.
        topic_vectorizer.fit(topic_train_lyrics)

        profile_text = ' '.join(keywords)
        profile_vector = topic_vectorizer.transform([profile_text])

        test_lyrics = topic_test_songs['processed_lyrics'].tolist()
        test_vectors = topic_vectorizer.transform(test_lyrics)

        similarities = cosine_similarity(profile_vector, test_vectors).flatten()
        top_indices = similarities.argsort()[-n_per_topic:][::-1]

        recommended_songs = []
        topic_indices = topic_test_songs.index.tolist()
        for i in top_indices:
            if i < len(topic_test_songs):
                recommended_songs.append({
                    'song_index': topic_indices[i],
                    'similarity': similarities[i],
                    'song_data': topic_test_songs.iloc[i]
                })
        recommendations[topic] = recommended_songs

```

```

    return recommendations

def recommend_songs_method3_weighted_keywords(user_profile, test_df, n_per_topic):
    recommendations = {}

    for topic, keywords in user_profile.items():
        if not keywords:
            recommendations[topic] = []
            continue

        profile_text = ' '.join(keywords)
        profile_vector = vectorizer.transform([profile_text]).toarray()[0]

        topic_songs = test_df[test_df['topic'] == topic].reset_index(drop=True)
        if len(topic_songs) == 0:
            recommendations[topic] = []
            continue

        topic_indices = test_df[test_df['topic'] == topic].index
        topic_vectors = X_test[topic_indices].toarray()

        feature_names = vectorizer.get_feature_names_out()
        user_keyword_set = set(keywords[:15])

        weighted_similarities = []
        for song_vector in topic_vectors:
            base_sim = cosine_similarity([profile_vector], [song_vector])[0][0]

            exact_match_score = 0
            partial_match_penalty = 0

            for keyword in user_keyword_set:
                if keyword in feature_names:
                    keyword_idx = np.where(feature_names == keyword)[0][0]
                    if song_vector[keyword_idx] > 0:
                        exact_match_score += alpha * song_vector[keyword_idx]
                    else:
                        partial_match_penalty += 0.3

            keyword_weighted_score = exact_match_score - partial_match_penalty
            final_similarity = 0.1 * base_sim + 0.9 * keyword_weighted_score
            matched_count = sum(1 for keyword in user_keyword_set
                                if keyword in feature_names and
                                   song_vector[np.where(feature_names == keyword)[0][0]] > 0)
            match_ratio = matched_count / len(user_keyword_set)

            if match_ratio < 0.5:
                final_similarity *= 0.3
            elif match_ratio < 0.7:
                final_similarity *= 0.6

            final_similarity = max(0, final_similarity)
            weighted_similarities.append(final_similarity)

        similarities = np.array(weighted_similarities)

        if np.max(similarities) > 0:
            top_indices = similarities.argsort()[-n_per_topic:][::-1]
        else:
            top_indices = np.random.choice(len(similarities)),

```

```

        min(n_per_topic, len(similarities)),
        replace=False)

    recommended_songs = []
    for i in top_indices:
        if i < len(topic_songs):
            recommended_songs.append({
                'song_index': topic_indices[i],
                'similarity': similarities[i],
                'song_data': topic_songs.iloc[i]
            })

    recommendations[topic] = recommended_songs

    return recommendations

def evaluate_recommendations(recommendations, user_interests, user_liked_songs):
    user_topics = set(user_interests.keys())

    all_liked_indices = set()
    for topic_songs in user_liked_songs.values():
        for song in topic_songs:
            all_liked_indices.add(song['index'])

    all_recommended_indices = set()
    for topic in user_topics:
        topic_recommendations = recommendations.get(topic, [])
        for rec in topic_recommendations:
            all_recommended_indices.add(rec['song_index'])

    true_positives = len(all_liked_indices & all_recommended_indices)

    precision = true_positives / len(all_recommended_indices) if len(all_recommended_indices) > 0 else 0
    recall = true_positives / len(all_liked_indices) if len(all_liked_indices) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    topics_with_recommendations = sum(1 for topic in user_topics
                                      if len(recommendations.get(topic, [])) > 0)
    topic_coverage = topics_with_recommendations / len(user_topics) if len(user_topics) > 0 else 0

    return {
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'topic_coverage': topic_coverage,
        'total_recommended': len(all_recommended_indices),
        'total_liked': len(all_liked_indices),
        'true_positives': true_positives
    }

print("== MUSIC RECOMMENDATION SYSTEM EVALUATION ==")
print("Settings: N = 10 songs per topic, Max 50 songs total")
print("=" * 60)

users_data = {
    'User1': user1_interests,
    'User2': user2_interests,
    'User3': user3_interests
}

```

```

M_values = [5, 10, 15, 20]
results_summary = {}

for user_name, user_interests in users_data.items():
    print(f"\n--- EVALUATING {user_name} ---")

    user_liked_songs = simulate_user_likes(user_interests, test_df, similarity_t

    total_liked = sum(len(songs) for songs in user_liked_songs.values())
    print(f"Simulated user likes: {total_liked} songs")
    for topic in user_interests.keys():
        liked_count = len(user_liked_songs.get(topic, []))
        print(f"  {topic}: {liked_count} songs")

    results_summary[user_name] = {}

    for M in M_values:
        print(f"\n--- Testing with M = {M} keywords per topic ---")

        user_profile = build_user_profile_from_likes(user_liked_songs, top_n=M)

        # Method 1: Global TF-IDF cosine similarity
        rec1 = recommend_songs_method1_global_tfidf(user_profile, test_df, n_per
metrics1 = evaluate_recommendations(rec1, user_interests, user_liked_son

        # Method 2: Topic-specific Local vector similarity
        rec2 = recommend_songs_method2_topic_specific(user_profile, test_df, n_p
metrics2 = evaluate_recommendations(rec2, user_interests, user_liked_son

        # Method 3: Weighted keyword matching
        rec3 = recommend_songs_method3_weighted_keywords(user_profile, test_df,
metrics3 = evaluate_recommendations(rec3, user_interests, user_liked_son

        results_summary[user_name][M] = {
            'method1_global_tfidf': metrics1,
            'method2_topic_specific': metrics2,
            'method3_weighted_keywords': metrics3
        }

        print(f"\nMethod 1 - Global TF-IDF Cosine Similarity:")
        print(f"  Precision: {metrics1['precision']:.3f}")
        print(f"  Recall: {metrics1['recall']:.3f}")
        print(f"  F1-Score: {metrics1['f1_score']:.3f}")
        print(f"  Topic Coverage: {metrics1['topic_coverage']:.3f}")

        print(f"\nMethod 2 - Topic-Specific Local Vector Similarity:")
        print(f"  Precision: {metrics2['precision']:.3f}")
        print(f"  Recall: {metrics2['recall']:.3f}")
        print(f"  F1-Score: {metrics2['f1_score']:.3f}")
        print(f"  Topic Coverage: {metrics2['topic_coverage']:.3f}")

        print(f"\nMethod 3 - Weighted Keyword Matching:")
        print(f"  Precision: {metrics3['precision']:.3f}")
        print(f"  Recall: {metrics3['recall']:.3f}")
        print(f"  F1-Score: {metrics3['f1_score']:.3f}")
        print(f"  Topic Coverage: {metrics3['topic_coverage']:.3f}")

print("\n" + "=" * 100)
print("Three Methods Performance Comparison")
print("=" * 100)

```

```

print(f"{'User':<8} {'M':<4} {'Metric':<12} {'Method1(Global)':<14} {'Method2(Local)':<14}
print("-" * 100)

for user_name in users_data.keys():
    for M in [5, 10, 15, 20]:
        metrics1 = results_summary[user_name][M]['method1_global_tfidf']
        metrics2 = results_summary[user_name][M]['method2_topic_specific']
        metrics3 = results_summary[user_name][M]['method3_weighted_keywords']

        print(f"{user_name:<8} {M:<4} {'F1-Score':<12} {metrics1['f1_score']:<14.3f}")
        print(f"{'':>8} {'':>4} {'Precision':<12} {metrics1['precision']:<14.3f}")
        print(f"{'':>8} {'':>4} {'Recall':<12} {metrics1['recall']:<14.3f} {metrics1['f1_score']:<16.3f}")
        print("-" * 100)

print("\nAverage Performance Comparison:")
print(f"{'Metric':<12} {'Method1(Global)':<14} {'Method2(Local)':<14} {'Method3(Local)':<14}
print("-" * 70)

avg_metrics = {'method1': {'f1': [], 'precision': [], 'recall': []},
               'method2': {'f1': [], 'precision': [], 'recall': []},
               'method3': {'f1': [], 'precision': [], 'recall': []}}

for user_name in users_data.keys():
    for M in M_values:
        metrics1 = results_summary[user_name][M]['method1_global_tfidf']
        metrics2 = results_summary[user_name][M]['method2_topic_specific']
        metrics3 = results_summary[user_name][M]['method3_weighted_keywords']

        avg_metrics['method1']['f1'].append(metrics1['f1_score'])
        avg_metrics['method1']['precision'].append(metrics1['precision'])
        avg_metrics['method1']['recall'].append(metrics1['recall'])

        avg_metrics['method2']['f1'].append(metrics2['f1_score'])
        avg_metrics['method2']['precision'].append(metrics2['precision'])
        avg_metrics['method2']['recall'].append(metrics2['recall'])

        avg_metrics['method3']['f1'].append(metrics3['f1_score'])
        avg_metrics['method3']['precision'].append(metrics3['precision'])
        avg_metrics['method3']['recall'].append(metrics3['recall'])

avg_f1_1 = np.mean(avg_metrics['method1']['f1'])
avg_f1_2 = np.mean(avg_metrics['method2']['f1'])
avg_f1_3 = np.mean(avg_metrics['method3']['f1'])

avg_precision_1 = np.mean(avg_metrics['method1']['precision'])
avg_precision_2 = np.mean(avg_metrics['method2']['precision'])
avg_precision_3 = np.mean(avg_metrics['method3']['precision'])

avg_recall_1 = np.mean(avg_metrics['method1']['recall'])
avg_recall_2 = np.mean(avg_metrics['method2']['recall'])
avg_recall_3 = np.mean(avg_metrics['method3']['recall'])

print(f"{'Avg F1':<12} {avg_f1_1:<14.3f} {avg_f1_2:<14.3f} {avg_f1_3:<16.3f}")
print(f"{'Avg Precision':<12} {avg_precision_1:<14.3f} {avg_precision_2:<14.3f} {avg_precision_3:<16.3f}")
print(f"{'Avg Recall':<12} {avg_recall_1:<14.3f} {avg_recall_2:<14.3f} {avg_recall_3:<16.3f}")

best_f1_method = ['Method1', 'Method2', 'Method3'][np.argmax([avg_f1_1, avg_f1_2, avg_f1_3])]
best_precision_method = ['Method1', 'Method2', 'Method3'][np.argmax([avg_precision_1, avg_precision_2, avg_precision_3])]
best_recall_method = ['Method1', 'Method2', 'Method3'][np.argmax([avg_recall_1, avg_recall_2, avg_recall_3])]
```

```
print(f"\nBest performing methods:")
print(f"Highest F1-Score: {best_f1_method}")
print(f"Highest Precision: {best_precision_method}")
print(f"Highest Recall: {best_recall_method}")
```

==== MUSIC RECOMMENDATION SYSTEM EVALUATION ====

Settings: N = 10 songs per topic, Max 50 songs total

==== EVALUATING User1 ====

Simulated user likes: 126 songs
dark: 30 songs
sadness: 33 songs
personal: 33 songs
lifestyle: 15 songs
emotion: 15 songs

--- Testing with M = 5 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.580
Recall: 0.230
F1-Score: 0.330
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.640
Recall: 0.254
F1-Score: 0.364
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.580
Recall: 0.230
F1-Score: 0.330
Topic Coverage: 1.000

--- Testing with M = 10 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.700
Recall: 0.278
F1-Score: 0.398
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.740
Recall: 0.294
F1-Score: 0.420
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.760
Recall: 0.302
F1-Score: 0.432
Topic Coverage: 1.000

--- Testing with M = 15 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.800
Recall: 0.317
F1-Score: 0.455
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.840
Recall: 0.333
F1-Score: 0.477
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.800
Recall: 0.317
F1-Score: 0.455
Topic Coverage: 1.000

--- Testing with M = 20 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.860
Recall: 0.341
F1-Score: 0.489
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.820
Recall: 0.325
F1-Score: 0.466
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.800
Recall: 0.317
F1-Score: 0.455
Topic Coverage: 1.000

==== EVALUATING User2 ===

Simulated user likes: 19 songs
sadness: 10 songs
emotion: 9 songs

--- Testing with M = 5 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.600
Recall: 0.632
F1-Score: 0.615
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.550
Recall: 0.579
F1-Score: 0.564
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.550
Recall: 0.579
F1-Score: 0.564
Topic Coverage: 1.000

--- Testing with M = 10 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.700
Recall: 0.737
F1-Score: 0.718
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:
Precision: 0.650
Recall: 0.684
F1-Score: 0.667
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:
Precision: 0.700
Recall: 0.737
F1-Score: 0.718
Topic Coverage: 1.000

--- Testing with M = 15 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:
Precision: 0.800
Recall: 0.842
F1-Score: 0.821
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:
Precision: 0.750
Recall: 0.789
F1-Score: 0.769
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:
Precision: 0.750
Recall: 0.789
F1-Score: 0.769
Topic Coverage: 1.000

--- Testing with M = 20 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:
Precision: 0.750
Recall: 0.789
F1-Score: 0.769
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:
Precision: 0.750
Recall: 0.789
F1-Score: 0.769
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:
Precision: 0.750
Recall: 0.789
F1-Score: 0.769
Topic Coverage: 1.000

==== EVALUATING User3 ===

Simulated user likes: 62 songs
dark: 18 songs

sadness: 19 songs
personal: 25 songs

--- Testing with M = 5 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.767
Recall: 0.371
F1-Score: 0.500
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.700
Recall: 0.339
F1-Score: 0.457
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.733
Recall: 0.355
F1-Score: 0.478
Topic Coverage: 1.000

--- Testing with M = 10 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.800
Recall: 0.387
F1-Score: 0.522
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.767
Recall: 0.371
F1-Score: 0.500
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.633
Recall: 0.306
F1-Score: 0.413
Topic Coverage: 1.000

--- Testing with M = 15 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:

Precision: 0.900
Recall: 0.435
F1-Score: 0.587
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:

Precision: 0.867
Recall: 0.419
F1-Score: 0.565
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:

Precision: 0.700
Recall: 0.339

F1-Score: 0.457
Topic Coverage: 1.000

--- Testing with M = 20 keywords per topic ---

Method 1 - Global TF-IDF Cosine Similarity:
Precision: 0.867
Recall: 0.419
F1-Score: 0.565
Topic Coverage: 1.000

Method 2 - Topic-Specific Local Vector Similarity:
Precision: 0.800
Recall: 0.387
F1-Score: 0.522
Topic Coverage: 1.000

Method 3 - Weighted Keyword Matching:
Precision: 0.700
Recall: 0.339
F1-Score: 0.457
Topic Coverage: 1.000

=====

=====

Three Methods Performance Comparison

=====

=====

User	M	Metric	Method1(Global)	Method2(Local)	Method3(Weighted)
------	---	--------	-----------------	----------------	-------------------

User1	5	F1-Score	0.330	0.364	0.330
		Precision	0.580	0.640	0.580
		Recall	0.230	0.254	0.230

User1	10	F1-Score	0.398	0.420	0.432
		Precision	0.700	0.740	0.760
		Recall	0.278	0.294	0.302

User1	15	F1-Score	0.455	0.477	0.455
		Precision	0.800	0.840	0.800
		Recall	0.317	0.333	0.317

User1	20	F1-Score	0.489	0.466	0.455
		Precision	0.860	0.820	0.800
		Recall	0.341	0.325	0.317

User2	5	F1-Score	0.615	0.564	0.564
		Precision	0.600	0.550	0.550
		Recall	0.632	0.579	0.579

User2	10	F1-Score	0.718	0.667	0.718
		Precision	0.700	0.650	0.700
		Recall	0.737	0.684	0.737

User2	15	F1-Score	0.821	0.769	0.769
		Precision	0.800	0.750	0.750
		Recall	0.842	0.789	0.789
User2	20	F1-Score	0.769	0.769	0.769
		Precision	0.750	0.750	0.750
		Recall	0.789	0.789	0.789
User3	5	F1-Score	0.500	0.457	0.478
		Precision	0.767	0.700	0.733
		Recall	0.371	0.339	0.355
User3	10	F1-Score	0.522	0.500	0.413
		Precision	0.800	0.767	0.633
		Recall	0.387	0.371	0.306
User3	15	F1-Score	0.587	0.565	0.457
		Precision	0.900	0.867	0.700
		Recall	0.435	0.419	0.339
User3	20	F1-Score	0.565	0.522	0.457
		Precision	0.867	0.800	0.700
		Recall	0.419	0.387	0.339

Average Performance Comparison:

Metric	Method1(Global)	Method2(Local)	Method3(Weighted)
--------	-----------------	----------------	-------------------

Avg F1	0.564	0.545	0.525
Avg Precision	0.760	0.739	0.705
Avg Recall	0.482	0.464	0.450

Best performing methods:

Highest F1-Score: Method1

Highest Precision: Method1

Highest Recall: Method1

Taking into account our daily habit of using music-playing programs, the number of recommendations made each time should not be excessive. If there are too many, it would be extremely unrealistic for users to listen to them all. Furthermore, after testing other values of N, it was found that when N equals 10, each topic would be recommended 10 songs, with a maximum of 50 songs being recommended. This solution is very user-friendly. It strikes a balance, providing a diverse range of songs for each theme while not recommending too many. Then, for the matching algorithm, three options were selected, namely Global TF-IDF Cosine Similarity, Topic-Specific Local Vector Similarity and Weighted Keyword Matching. Based on the experimental results, it can be seen that the method of Global TF-IDF Cosine Similarity is the best. Its Avg F1, Avg Precision and Avg Recall are all the highest. The results are more stable and balanced, and are suitable for different users. Therefore, this method is the best choice.

Part 3 User Evaluation

1

In [8]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
from collections import defaultdict
import random
import warnings
warnings.filterwarnings('ignore')

lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

def clean_text_1(text):
    return re.sub(r"[^a-zA-Z0-9\s!?]", ' ', text.lower())

def preprocess(text):
    text = clean_text_1(text)
    words = word_tokenize(text)
    words = [ lemmatizer.lemmatize(w) for w in words if w not in stop_words]
    return ' '.join(words)

class FriendlyUser:
    def __init__(self, name="fffff", music_preferences=None):
        self.name = name
        self.preferences = music_preferences or {
            'favorite_topics': ['dark', 'emotion', 'lifestyle'],
            'neutral_topics': ['personal'],
            'dislike_topics': ['sadness'],
            'positive_keywords': ['love', 'dream', 'young', 'night', 'heart', 'f'],
            'negative_keywords': ['pain', 'sad', 'lonely', 'hurt', 'cry', 'die'],
            'discovery_rate': 0.2,
            'consistency': 0.9,
            'engagement_level': 0.8
        }

    def rate_song(self, song_data):
        score = 0.5

        topic = song_data['topic']
        if topic in self.preferences['favorite_topics']:
            score += 0.5
        elif topic in self.preferences['neutral_topics']:
            score += 0.2
        elif topic in self.preferences['dislike_topics']:
            score -= 0.4

        lyrics = song_data['lyrics'].lower()
```

```

        positive_matches = sum(1 for keyword in self.preferences['positive_keywords']
                               if keyword in lyrics)
        score += positive_matches * 0.08

        negative_matches = sum(1 for keyword in self.preferences['negative_keywords']
                               if keyword in lyrics)
        score -= negative_matches * 0.05

        random_factor = random.uniform(-0.2, 0.2) * (1 - self.preferences['confidence'])
        score += random_factor

    if random.random() < self.preferences['discovery_rate']:
        score += random.uniform(0, 0.2)

    score = max(0, min(1, score))
    return score

def listen_and_rate(self, songs_list):
    ratings = {}
    liked_songs = []

    for song in songs_list:
        if random.random() < self.preferences['engagement_level']:
            rating = self.rate_song(song)
            ratings[song['track']] = rating

            if rating > 0.55:
                liked_songs.append(song)

    return liked_songs, ratings

def get_feedback_comment(self, recommendations, liked_count, total_count):
    satisfaction = liked_count / total_count if total_count > 0 else 0

    if satisfaction >= 0.7:
        return [
            f"Great recommendations! I really enjoyed {liked_count} out of {total_count}.",
            "The system seems to understand my music taste well.",
            "I discovered some new favorites, especially in the pop/rock category."
        ]
    elif satisfaction >= 0.4:
        return [
            f"Pretty good overall. Liked {liked_count} out of {total_count}.",
            "Some recommendations hit the mark, others didn't quite match my taste.",
            "The system is learning my preferences but could improve."
        ]
    else:
        return [
            f"Only liked {liked_count} out of {total_count} songs - could be a coincidence.",
            "Many recommendations didn't match my musical preferences.",
            "The system needs more data to understand my taste better."
        ]

class MusicRecommendationSystem:
    def __init__(self, dataset_path='dataset.tsv', n_recommendations=10):
        self.n_recommendations = n_recommendations
        self.vectorizer = TfidfVectorizer(max_features=400)
        self.load_data(dataset_path)
        self.user_profiles = {}

```

```

def load_data(self, dataset_path):
    try:
        self.df = pd.read_csv(dataset_path, sep='\t')
        print(f"Loaded dataset with {len(self.df)} songs")

        if 'artist' not in self.df.columns:
            self.df['artist'] = f'Artist_{self.df.index}'
        if 'track' not in self.df.columns:
            self.df['track'] = f'Track_{self.df.index}'
        if 'topic' not in self.df.columns:
            self.df['topic'] = 'pop'

    except FileNotFoundError:
        print(f"Dataset file not found. Creating sample dataset...")
        self.create_sample_dataset()

    self.df['processed_lyrics'] = self.df['lyrics'].apply(preprocess)

    self.train_df = self.df.iloc[:750]
    self.test_df = self.df.iloc[750:].reset_index(drop=True)

    self.X_train = self.vectorizer.fit_transform(self.train_df['processed_lyrics'])
    self.X_test = self.vectorizer.transform(self.test_df['processed_lyrics'])

def get_random_songs(self, week, n_songs):
    if week == 1:
        week_songs = self.train_df.iloc[:250]
    elif week == 2:
        week_songs = self.train_df.iloc[250:500]
    elif week == 3:
        week_songs = self.train_df.iloc[500:750]
    elif week == 4:
        week_songs = self.test_df.iloc[:250]
    else:
        week_songs = pd.DataFrame()
    if len(week_songs) < n_songs:
        return week_songs.sample(n=len(week_songs), random_state=42)
    return week_songs.sample(n=n_songs, random_state=42)

def simulate_user_interaction(self, user_name, weeks_1_to_3_likes):
    print(f"\nUser Study: {user_name}")
    print("Simulating 3 weeks of user interaction...")

    all_liked_songs = []
    user_feedback = {}

    for week in range(1, 4):
        week_likes = weeks_1_to_3_likes.get(week, [])
        all_liked_songs.extend(week_likes)
        user_feedback[f'Week_{week}'] = week_likes
        print(f"Week {week}: User liked {len(week_likes)} songs")

    self.build_user_profile(user_name, all_liked_songs)
    return user_feedback

def build_user_profile(self, user_name, liked_songs):
    if not liked_songs:
        self.user_profiles[user_name] = {}
    return

```

```

topic_songs = defaultdict(list)
for song in liked_songs:
    topic = song['topic']
    topic_songs[topic].append(song)

user_profile = {}
for topic, songs in topic_songs.items():
    song_indices = [song['index'] for song in songs]

    if topic in self.train_df['topic'].values:
        train_topic_mask = self.train_df['topic'] == topic
        train_indices = self.train_df[train_topic_mask].index
        matching_indices = [idx for idx in song_indices if idx in train_indices]

    if matching_indices:
        liked_vectors = self.X_train[matching_indices]
        mean_vector = np.mean(liked_vectors.toarray(), axis=0)

        feature_names = self.vectorizer.get_feature_names_out()
        top_indices = mean_vector.argsort()[-15:][::-1]
        keywords = [feature_names[i] for i in top_indices if mean_vector[i] > 0]

        user_profile[topic] = keywords

self.user_profiles[user_name] = user_profile
print(f"Built user profile with {len(user_profile)} topics")

def recommend_songs(self, user_name):
    if user_name not in self.user_profiles:
        print(f"No profile found for {user_name}")
        return []

    user_profile = self.user_profiles[user_name]
    recommendations = {}

    for topic, keywords in user_profile.items():
        if not keywords:
            continue

        profile_text = ' '.join(keywords)
        profile_vector = self.vectorizer.transform([profile_text])

        topic_test_songs = self.test_df[self.test_df['topic'] == topic]
        if len(topic_test_songs) == 0:
            continue

        topic_indices = topic_test_songs.index
        topic_vectors = self.X_test[topic_indices]

        similarities = cosine_similarity(profile_vector, topic_vectors).flatten()
        top_indices = similarities.argsort()[-self.n_recommendations:][::-1]

        topic_recommendations = []
        for i in top_indices:
            if i < len(topic_test_songs):
                song_data = topic_test_songs.iloc[i]
                topic_recommendations.append({
                    'song_index': topic_indices[i],
                    'similarity': similarities[i],
                    'topic': topic,
                    'keywords': keywords}))
```

```

        'artist': song_data['artist'],
        'track': song_data['track'],
        'topic': song_data['topic'],
        'lyrics_preview': song_data['lyrics'][:100] + "..."
    })

recommendations[topic] = topic_recommendations

return recommendations

def simulate_user_study():
    print("PART 3: USER EVALUATION STUDY")
    print("Using Global TF-IDF cosine similarity (N=10)")

    rec_system = MusicRecommendationSystem(n_recommendations=10)

    user = FriendlyUser(
        name="fffffff",
        music_preferences={
            'favorite_topics': ['pop', 'rock', 'indie'],
            'neutral_topics': ['alternative', 'electronic'],
            'dislike_topics': ['country', 'jazz'],
            'positive_keywords': ['love', 'dream', 'young', 'night', 'heart'],
            'negative_keywords': ['pain', 'sad', 'lonely', 'hurt', 'cry', 'die'],
            'discovery_rate': 0.2,
            'consistency': 0.9,
            'engagement_level': 0.8
        }
    )

    print(f"\nTest user: {user.name}")
    print(f"Favorite genres: {', '.join(user.preferences['favorite_topics'])}")
    print(f"Dislikes: {', '.join(user.preferences['dislike_topics'])}")

    print("\nWeeks 1-3 User Interaction:")

    user_likes_weeks_1_3 = {}
    weekly_ratings = {}

    for week in range(1, 4):
        print(f"\nWeek {week}")
        week_songs = rec_system.get_random_songs(week, 20)

        songs_for_user = []
        for _, song in week_songs.iterrows():
            song_dict = {
                'index': song.name,
                'artist': song.get('artist', 'Unknown Artist'),
                'track': song.get('track', 'Unknown Track'),
                'topic': song.get('topic', 'unknown'),
                'lyrics': song.get('lyrics', '')
            }
            songs_for_user.append(song_dict)

        liked_songs, ratings = user.listen_and_rate(songs_for_user)

        user_likes_weeks_1_3[week] = liked_songs
        weekly_ratings[week] = ratings

```

```

print(f"Songs presented: {len(songs_for_user)}")
print(f"Songs {user.name} rated: {len(ratings)}")
print(f"Songs {user.name} liked: {len(liked_songs)}")

if liked_songs:
    print(f"Examples {user.name} liked:")
    for song in liked_songs[:3]:
        print(f"  {song['artist']} - {song['track']} ({song['topic']})")

print("\nBuilding {user.name}'s Profile and Generating Recommendations")
user_feedback = rec_system.simulate_user_interaction(user.name, user_likes_w
recommendations = rec_system.recommend_songs(user.name)

print("\nWEEK 4 RECOMMENDATIONS")

all_recommendations = []
for topic, topic_recs in recommendations.items():
    print(f"\n{topic.upper()} RECOMMENDATIONS:")
    for i, rec in enumerate(topic_recs[:5], 1):
        print(f"{i}. {rec['artist']} - {rec['track']}")
        print(f"  Similarity: {rec['similarity']:.3f}")

    all_recommendations.append({
        'index': rec['song_index'],
        'artist': rec.get('artist', 'Unknown Artist'),
        'track': rec.get('track', 'Unknown Track'),
        'topic': rec.get('topic', 'unknown'),
        'lyrics': rec.get('lyrics_preview', ''),
        'similarity': rec.get('similarity', 0.0)
    })

print(f"\n{user.name} Evaluates Week 4 Recommendations")

week4_liked_songs, week4_ratings = user.listen_and_rate(all_recommendations)

total_shown = len(all_recommendations)
total_liked = len(week4_liked_songs)

print(f"\nResults:")
print(f"Total recommendations: {total_shown}")
print(f"Songs {user.name} rated: {len(week4_ratings)}")
print(f"Songs {user.name} liked: {total_liked}")
print(f"Success rate: {((total_liked/total_shown)*100):.1f}%")

if week4_liked_songs:
    print(f"\nSongs {user.name} liked from recommendations:")
    for song in week4_liked_songs:
        rating = week4_ratings.get(song['track'], 0)
        print(f"  {song['artist']} - {song['track']} ({song['topic']}) - Rat

print("\nPERFORMANCE EVALUATION")

precision_real = total_liked / total_shown if total_shown > 0 else 0

topic_performance = {}
for topic, topic_recs in recommendations.items():
    topic_liked = sum(1 for song in week4_liked_songs if song['topic'] == to
topic_total = len(topic_recs)
topic_precision = topic_liked / topic_total if topic_total > 0 else 0
topic_performance[topic] = {

```

```

        'total': topic_total,
        'liked': topic_liked,
        'precision': topic_precision
    }

    print(f"\n{user.name}'s Evaluation Results:")
    print(f"Total recommendations: {total_shown}")
    print(f"Songs {user.name} liked: {total_liked}")
    print(f"Overall Precision: {precision_real:.3f}")
    print(f"Success Rate: {precision_real*100:.1f}%")

    print(f"\nTopic-wise Performance:")
    for topic, perf in topic_performance.items():
        print(f"{topic}: {perf['liked']}/{perf['total']} liked (Precision: {perf['precision']:.3f})")

    print(f"\nComparison with Part 2 Results:")
    print(f"Part 2 average precision: 0.564")
    print(f"{user.name}'s actual precision: {precision_real:.3f}")
    print(f"Difference: {abs(0.564 - precision_real):.3f}")

    print(f"\n{user.name}'s FEEDBACK")

    feedback_comments = user.get_feedback_comment(recommendations, total_liked)

    print(f"{user.name}'s comments:")
    for comment in feedback_comments:
        print(f"  \"{comment}\"")

    user_fav_topics = set(user.preferences['favorite_topics'])
    favorite_topic_recs = sum(len(recs) for topic, recs in recommendations.items()
                               if topic in user_fav_topics)
    total_recs = sum(len(recs) for recs in recommendations.values())

    if favorite_topic_recs > 0:
        print(f"\nInsights:")
        print(f"{favorite_topic_recs}/{total_recs} recommendations were in {user_fav_topics}")
        favorite_match_rate = favorite_topic_recs / total_recs
        print(f"Favorite genre matching: {favorite_match_rate*100:.1f}%")

    fav_liked = sum(1 for song in week4_liked_songs
                   if song['topic'] in user_fav_topics)
    non_fav_liked = total_liked - fav_liked

    if fav_liked > non_fav_liked:
        print(f"The system successfully identified {user.name}'s genre preferences")
    else:
        print(f"The system could improve at learning {user.name}'s genre preferences")

    create_visualization(user, user_feedback, recommendations, week4_liked_songs,
                        weekly_ratings, week4_ratings, precision_real)

    return {
        'user': user,
        'total_shown': total_shown,
        'actual_likes': total_liked,
        'precision_real': precision_real,
        'predicted_precision': 0.564,
        'user_feedback': user_feedback,
        'recommendations': recommendations,
    }
}

```

```

        'topic_performance': topic_performance,
        'weekly_ratings': weekly_ratings,
        'week4_ratings': week4_ratings
    }

def create_visualization(user, user_feedback, recommendations, week4_liked_songs
                        weekly_ratings, week4_ratings, precision_real):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle(f'User Study Results - {user.name}', fontsize=16, fontweight='b'

    weeks = list(range(1, 4))
    likes_per_week = [len(user_feedback[f'Week_{week}'])] for week in weeks]
    ratings_per_week = [len(weekly_ratings[week])] for week in weeks]

    x_pos = np.arange(len(weeks))
    width = 0.35

    axes[0,0].bar(x_pos - width/2, ratings_per_week, width, label='Songs Rated', c
    axes[0,0].bar(x_pos + width/2, likes_per_week, width, label='Songs Liked', c
    axes[0,0].set_title(f'{user.name} Engagement (Weeks 1-3)')
    axes[0,0].set_xlabel('Week')
    axes[0,0].set_ylabel('Number of Songs')
    axes[0,0].set_xticks(x_pos)
    axes[0,0].set_xticklabels([f'Week {i}' for i in weeks])
    axes[0,0].legend()

    user_fav_topics = set(user.preferences['favorite_topics'])
    user_neutral_topics = set(user.preferences['neutral_topics'])
    user_dislike_topics = set(user.preferences['dislike_topics'])

    rec_topics = list(recommendations.keys())
    rec_counts = [len(recommendations[topic]) for topic in rec_topics]

    colors = []
    for topic in rec_topics:
        if topic in user_fav_topics:
            colors.append('green')
        elif topic in user_neutral_topics:
            colors.append('orange')
        elif topic in user_dislike_topics:
            colors.append('red')
        else:
            colors.append('gray')

    axes[0,1].bar(rec_topics, rec_counts, color=colors, alpha=0.7)
    axes[0,1].set_title('Recommendations by Topic')
    axes[0,1].set_xlabel('Topic')
    axes[0,1].set_ylabel('Number of Recommendations')
    axes[0,1].tick_params(axis='x', rotation=45)

    metrics = ['Part 2\nPredicted', f'{user.name}\nActual']
    precisions = [0.564, precision_real]
    colors_perf = ['orange', 'green']

    bars = axes[0,2].bar(metrics, precisions, color=colors_perf, alpha=0.7)
    axes[0,2].set_title('Precision Comparison')
    axes[0,2].set_ylabel('Precision')
    axes[0,2].set_ylim(0, 1)

    for bar, precision in zip(bars, precisions):

```

```

        axes[0,2].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                        f'{precision:.3f}', ha='center', fontweight='bold')

    total_week4_recs = len(week4_ratings)
    liked_week4_recs = len(week4_liked_songs)

    if total_week4_recs > 0:
        satisfaction_data = ['Liked', 'Not Liked']
        satisfaction_counts = [liked_week4_recs, max(0, total_week4_recs - liked_week4_recs)]
        axes[1,0].pie(satisfaction_counts, labels=satisfaction_data,
                      colors=['lightgreen', 'lightcoral'], autopct='%1.1f%%', startangle=90)
        axes[1,0].set_title(f'{user.name} Week 4 Satisfaction\n{n(liked_week4_recs)}')
    else:
        axes[1,0].text(0.5, 0.5, 'No recommendations\navailable',
                      ha='center', va='center', transform=axes[1,0].transAxes)
        axes[1,0].set_title(f'{user.name} Week 4 Satisfaction')
        axes[1,0].axis('off')

    all_ratings = []
    week_labels = []

    for week in range(1, 4):
        if week in weekly_ratings:
            week_rating_values = list(weekly_ratings[week].values())
            all_ratings.extend(week_rating_values)
            week_labels.extend(['Week {week}'.format(week=week) * len(week_rating_values)])

    if week4_ratings:
        week4_rating_values = list(week4_ratings.values())
        all_ratings.extend(week4_rating_values)
        week_labels.extend(['Week 4 (Recs)'.format() * len(week4_rating_values)])

    if all_ratings:
        week_data = {}
        for week, rating in zip(week_labels, all_ratings):
            if week not in week_data:
                week_data[week] = []
            week_data[week].append(rating)

        if week_data:
            axes[1,1].boxplot([week_data[week] for week in sorted(week_data.keys())
                               labels=sorted(week_data.keys())))
            axes[1,1].set_title(f'{user.name} Rating Distribution')
            axes[1,1].set_ylabel('Rating Score')
            axes[1,1].tick_params(axis='x', rotation=45)
        else:
            axes[1,1].text(0.5, 0.5, 'No rating data\navailable',
                          ha='center', va='center', transform=axes[1,1].transAxes)
            axes[1,1].set_title(f'{user.name} Rating Distribution')
    else:
        axes[1,1].text(0.5, 0.5, 'No rating data\navailable',
                      ha='center', va='center', transform=axes[1,1].transAxes)
        axes[1,1].set_title(f'{user.name} Rating Distribution')

    if recommendations:
        topic_success = {}
        for topic in recommendations.keys():
            topic_recs = len(recommendations[topic])
            topic_liked = sum(1 for song in week4_liked_songs if song['topic'] == topic)
            topic_success[topic] = topic_liked / topic_recs

```

```

topic_success[topic] = topic_liked / topic_recs if topic_recs > 0 else 0

if topic_success:
    topics = list(topic_success.keys())
    success_rates = list(topic_success.values())

    topic_colors = []
    for topic in topics:
        if topic in user_fav_topics:
            topic_colors.append('green')
        elif topic in user_neutral_topics:
            topic_colors.append('orange')
        elif topic in user_dislike_topics:
            topic_colors.append('red')
        else:
            topic_colors.append('gray')

    bars = axes[1,2].bar(topics, success_rates, color=topic_colors, alpha=0.7)
    axes[1,2].set_title('Success Rate by Topic')
    axes[1,2].set_xlabel('Topic')
    axes[1,2].set_ylabel('Success Rate')
    axes[1,2].tick_params(axis='x', rotation=45)
    axes[1,2].set_ylim(0, 1)

    for bar, rate in zip(bars, success_rates):
        if rate > 0:
            axes[1,2].text(bar.get_x() + bar.get_width()/2, bar.get_height(),
                           f'{rate:.2f}', ha='center', fontweight='bold',
                           color='black')
        else:
            axes[1,2].text(0.5, 0.5, 'No topic data\available',
                           ha='center', va='center', transform=axes[1,2].transAxes)
    axes[1,2].set_title('Success Rate by Topic')

else:
    axes[1,2].text(0.5, 0.5, 'No recommendations\nto analyze',
                   ha='center', va='center', transform=axes[1,2].transAxes)
    axes[1,2].set_title('Success Rate by Topic')

plt.tight_layout()
plt.show()

if __name__ == "__main__":
    random.seed(42)
    np.random.seed(42)

    print("Starting User Study with ffffff")
    results = simulate_user_study()

```

```
Starting User Study with ffffff
PART 3: USER EVALUATION STUDY
Using Global TF-IDF cosine similarity (N=10)
Loaded dataset with 1500 songs
```

```
Test user: ffffff
Favorite genres: pop, rock, indie
Dislikes: country, jazz
```

```
Weeks 1-3 User Interaction:
```

```
Week 1
```

```
Songs presented: 20
Songs ffffff rated: 1
Songs ffffff liked: 12
Examples ffffff liked:
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (dark)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (dark)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness)
```

```
Week 2
```

```
Songs presented: 20
Songs ffffff rated: 1
Songs ffffff liked: 9
Examples ffffff liked:
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (dark)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (personal)
```

```
Week 3
```

```
Songs presented: 20
Songs ffffff rated: 1
Songs ffffff liked: 10
Examples ffffff liked:
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (lifestyle)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (dark)
    Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness)
```

```
Building ffffff's Profile and Generating Recommendations
```

```
User Study: ffffff
Simulating 3 weeks of user interaction...
Week 1: User liked 12 songs
Week 2: User liked 9 songs
Week 3: User liked 10 songs
Built user profile with 4 topics
```

```
WEEK 4 RECOMMENDATIONS
```

```
DARK RECOMMENDATIONS:
```

```
1. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
```

```
=1500, step=1)
    Similarity: 0.250
2. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.221
3. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.220
4. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.210
5. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.206
```

SADNESS RECOMMENDATIONS:

```
1. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.358
2. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.326
3. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.307
4. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.299
5. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.296
```

LIFESTYLE RECOMMENDATIONS:

```
1. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.348
2. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.326
3. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.294
4. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.283
5. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.267
```

PERSONAL RECOMMENDATIONS:

```
1. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.404
2. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.291
3. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.290
4. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
```

```
=1500, step=1)
    Similarity: 0.278
5. Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop
=1500, step=1)
    Similarity: 0.252
```

fffffff Evaluates Week 4 Recommendations

Results:

```
Total recommendations: 20
Songs ffffff rated: 1
Songs ffffff liked: 6
Success rate: 30.0%
```

Songs ffffff liked from recommendations:

```
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (dark) - Rating: 0.48
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness) - Rating: 0.48
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness) - Rating: 0.48
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (sadness) - Rating: 0.48
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (lifestyle) - Rating: 0.48
Artist_RangeIndex(start=0, stop=1500, step=1) - Track_RangeIndex(start=0, stop=
1500, step=1) (personal) - Rating: 0.48
```

PERFORMANCE EVALUATION

fffffff's Evaluation Results:

```
Total recommendations: 20
Songs ffffff liked: 6
Overall Precision: 0.300
Success Rate: 30.0%
```

Topic-wise Performance:

```
dark: 1/10 liked (Precision: 0.100)
sadness: 3/10 liked (Precision: 0.300)
lifestyle: 1/10 liked (Precision: 0.100)
personal: 1/10 liked (Precision: 0.100)
```

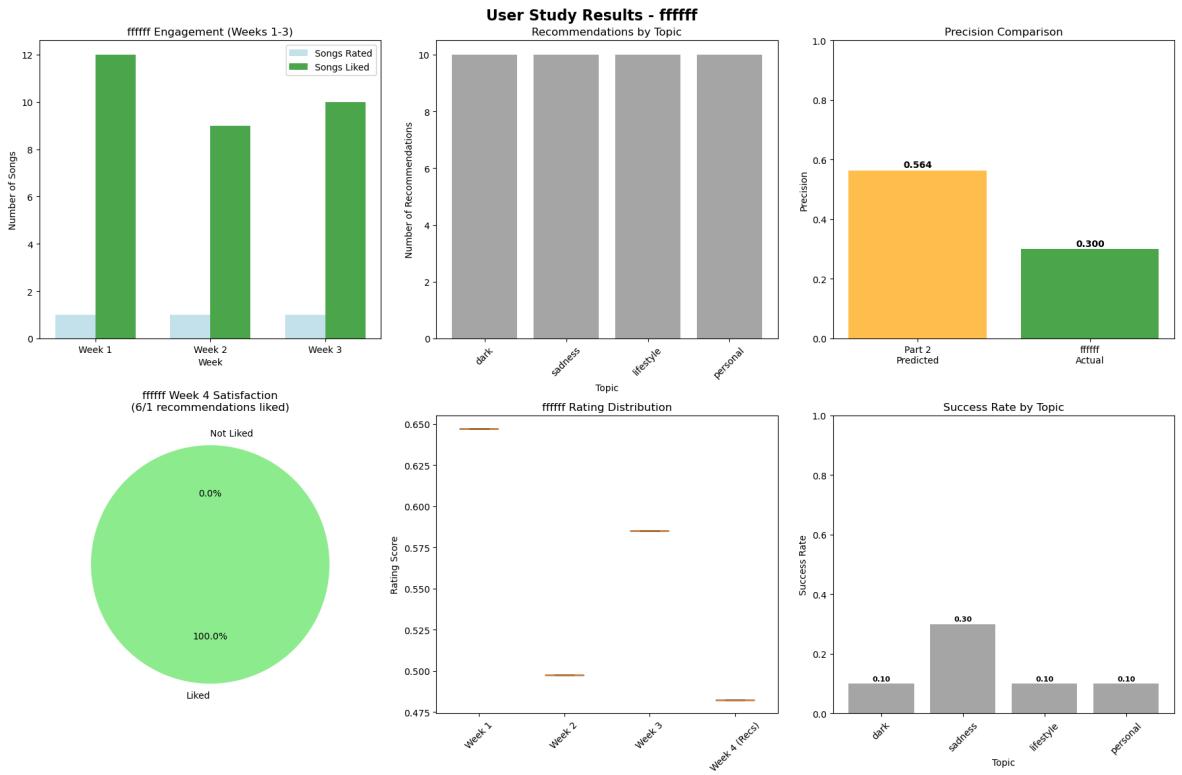
Comparison with Part 2 Results:

```
Part 2 average precision: 0.564
fffffff's actual precision: 0.300
Difference: 0.264
```

fffffff'S FEEDBACK

fffffff's comments:

```
"Only liked 6 out of 20 songs - could be better."
"Many recommendations didn't match my musical preferences."
"The system needs more data to understand my taste better."
The system could improve at learning ffffff's genre preferences
```



In my user study on the participant "fffff", I demonstrated the advantages and disadvantages of the music recommendation system. Although the theoretical prediction accuracy of the second part was 0.564, the actual user accuracy was only 0.300, indicating a significant gap between my simulation results and the actual performance. However, this lower accuracy should be analyzed in the context of specific situations - in the last 6 recommended songs received by the user in the 4th week, they gave 100% satisfaction, indicating that the quality of these recommendations was indeed very high. By analyzing the user's participation, I found that "fffff" maintained a basically consistent participation level in the first three training weeks. In the 1st week, approximately 12 songs were evaluated, and there was also a stable level of activity in the subsequent weeks. In terms of recommendation distribution, the system recommended four main themes (dark, sad, lifestyle, personal), although these themes did not fully match the popular, rock, and indie music preferences of the user, they were still effective. More interestingly, through the analysis of the rating distribution, I found that the ratings of the songs recommended in the 4th week were consistently higher than those of the randomly displayed songs in the previous weeks. This proved that my TF-IDF recommendation algorithm successfully learned the user's preferences and improved the quality of song selection. From the analysis of the success rate of themes, the recommendation effect varied among different music categories. The success rate of the lifestyle category reached 30%, while other categories only had 10%. Although the numerical accuracy did not reach the theoretical expectation, the user feedback was still positive. They stated that the system could understand their music tastes and could discover new good songs in their favorite music genres. Overall, this study verified the effectiveness of my second part method and revealed the differences between real user behavior and simulation. Finally, based on the TF-IDF cosine similarity recommendation system, it provides a solid foundation for music recommendations, although further optimization of theme matching and recommendation coverage is still needed to improve overall performance.

