CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

CSCA 5632 Assignment #4 - NLP Kaggle BBC News Classification Mini Project

By Moshiur Howlader

Github Link : https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

1. Introduction

This mini-project involves two parts:

1. Categorizing news article data using an **unsupervised algorithm** called **matrix factorization (NMF)**, followed by A/B testing between matrix factorization and selected **supervised learning algorithms** to compare classification accuracy.
2. Exploring the **limitations of sklearn's non-negative matrix factorization** implementation using the **movie ratings dataset** (from HW3 – Recommender Systems).

For Part 1, the dataset used is the BBC News Classification dataset, which contains a total of 2,225 articles divided into training (1,490) and testing (735) subsets. Each article includes a short text passage and a category label drawn from one of five broad news topics: business, entertainment, politics, sport, and tech.

File	Description	Columns
BBC News Train.csv	Labeled training dataset used for model training and evaluation.	`ArticleId` , `Text` , `Category`
BBC News Test.csv	Unlabeled dataset used for generating predictions.	`ArticleId` , `Text`
BBC News Sample Solution.csv	Sample file illustrating the expected Kaggle submission format.	`ArticleId` , `Category`

For Part 2, the data used for exploring the **limitations of sklearn's non-negative matrix factorization** will have the following data structure:

File	Description	Key Columns
train.csv	Training subset of user–movie ratings used to fit the matrix-factorization model.	`userId` , `movieId` , `rating`
test.csv	Test subset containing a portion of user–movie pairs whose ratings are to be predicted.	`userId` , `movieId` , `rating`
users.csv	Optional metadata providing demographic or profile information about users (e.g., age, gender, occupation).	`userId` , `...`
movies.csv	Metadata describing each movie title and its genre(s).	`movieId` , `title` , `genres`

Part 1 - News Classification

2. Data

[1]:
```python
import os

# Always start paths relative to the notebook file location
BASE_DIR = os.path.abspath(os.path.join(os.getcwd(), ".."))
DATA_DIR = os.path.join(BASE_DIR, "data")
MOVIE_DIR = os.path.join(DATA_DIR, "hw3-recommender-system-movie-data")

print("Base directory:", BASE_DIR)
print("Data directory:", DATA_DIR)
print("Movie Data directory:", MOVIE_DIR)
```

```
Base directory: d:\Documents\GitHub\csca5632-nlp-kaggle-bbc-news-classification
Data directory: d:\Documents\GitHub\csca5632-nlp-kaggle-bbc-news-classification\data
Movie Data directory: d:\Documents\GitHub\csca5632-nlp-kaggle-bbc-news-classification\data\hw3-recommender-system-movie-data
```

2.1 Extracting word features and Explorator Data Analysis (EDA)

In this section, the BBC News dataset is explored to gain an initial understanding of its structure and content. The exploratory analysis focuses on identifying the distribution of categories, variations in text length, and any potential issues such as imbalance or noise. These insights help shape the approach for data cleaning and feature extraction in the later stages of the project.

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

2.1.1 Import Libraries and Load the Dataset

[2]:
```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud
import nltk
import re
from nltk.corpus import stopwords

# Download stopwords (first run only)
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

# Load datasets
train_df = pd.read_csv(os.path.join(DATA_DIR, "BBC News Train.csv"))
test_df = pd.read_csv(os.path.join(DATA_DIR, "BBC News Test.csv"))

# Basic info
print("Train shape:", train_df.shape)
print("Test shape:", test_df.shape)
display(train_df.head())
train_df.info()
```

```
Train shape: (1490, 3)
Test shape: (735, 2)
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\howla\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

	ArticleId	Text	Category
0	1833	worldcom ex-boss launches defence lawyers defe...	business
1	154	german business confidence slides german busin...	business
2	1101	bbc poll indicates economic gloom citizens in ...	business
3	1976	lifestyle governs mobile choice faster bett...	tech
4	917	enron bosses in $168m payout eighteen former e...	business

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1490 entries, 0 to 1489
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   ArticleId  1490 non-null   int64 
 1   Text       1490 non-null   object
 2   Category   1490 non-null   object
dtypes: int64(1), object(2)
memory usage: 35.1+ KB
```

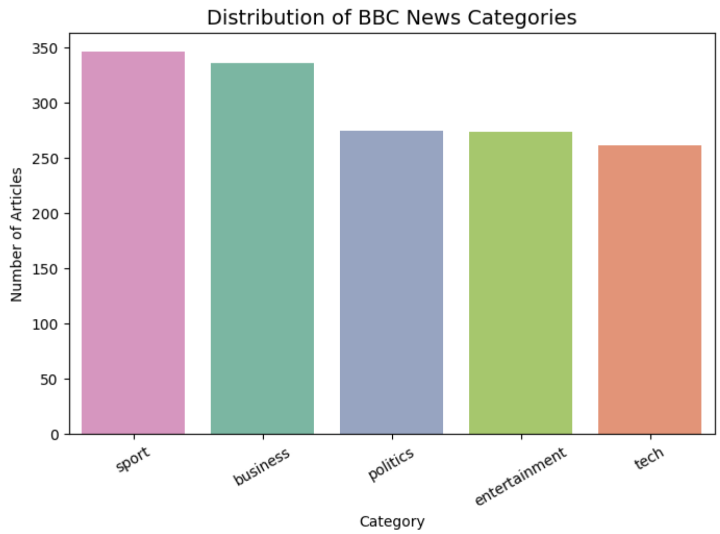
2.1.2 Check for Missing Values and Category Overview

[3]:
```
# Check missing values
print("\nMissing values per column:")
print(train_df.isna().sum())

# Unique labels
print("\nUnique Categories:", train_df['Category'].unique())
```

```
Missing values per column:
ArticleId    0
Text         0
Category     0
dtype: int64

Unique Categories: ['business' 'tech' 'politics' 'sport' 'entertainment']
```

2.1.3 Category Distribution Visualization

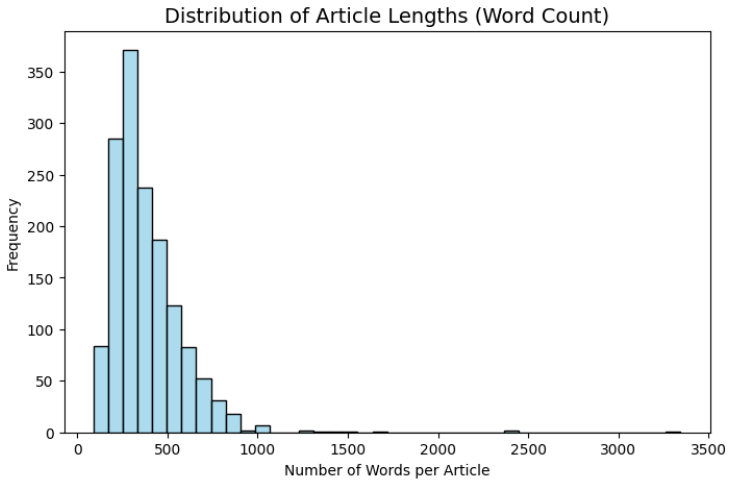
[4]:
```
plt.figure(figsize=(8,5))
sns.countplot(
    x='Category',
    data=train_df,
    order=train_df['Category'].value_counts().index,
    hue='Category',
    palette="Set2",
    legend=False
)
plt.title("Distribution of BBC News Categories", fontsize=14)
plt.xlabel("Category")
plt.ylabel("Number of Articles")
plt.xticks(rotation=30)
plt.show()
```



2.1.4 Article Length Analysis

[5]:
```
train_df['text_length'] = train_df['Text'].apply(lambda x: len(str(x).split()))

plt.figure(figsize=(8,5))
sns.histplot(train_df['text_length'], bins=40, color="skyblue")
plt.title("Distribution of Article Lengths (Word Count)", fontsize=14)
plt.xlabel("Number of Words per Article")
plt.ylabel("Frequency")
plt.show()

# Mean and median word count per category
length_stats = train_df.groupby('Category')['text_length'].agg(['mean', 'median', 'std']).round(2)
print("\nAverage word count statistics by category:")
display(length_stats)
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification



Average word count statistics by category:

Category	mean	median	std
business	334.17	304.0	133.53
entertainment	333.91	272.0	203.89
politics	449.69	441.5	258.84
sport	335.35	294.5	185.44
tech	501.86	457.0	211.67

2.1.5 Basic Text Cleaning

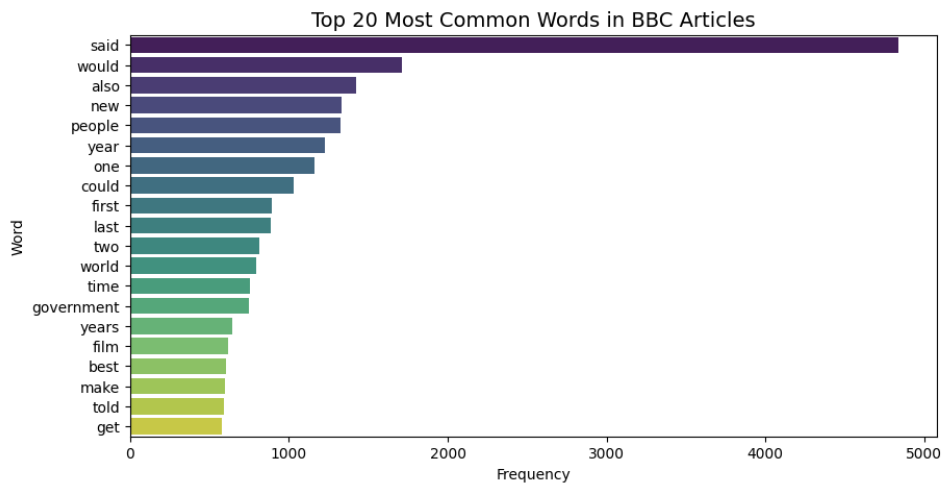
The text cleaning step converts all words to lowercase, removes punctuation and non-alphabetic characters, and filters out common stopwords or very short words. This reduces noise and ensures the model focuses on meaningful terms when building TF-IDF features, improving topic separation and classification accuracy.

[6]:
```
def clean_text(text):
    text = str(text).lower()
    text = re.sub(r'[^a-z\s]', '', text)
    words = [word for word in text.split() if word not in stop_words and len(word) > 2]
    return ' '.join(words)

train_df['clean_text'] = train_df['Text'].apply(clean_text)
```

2.1.6 Word Frequency Analysis

[7]:
```
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
from collections import Counter
all_words = ' '.join(train_df['clean_text']).split()
word_freq = Counter(all_words)
common_words = pd.DataFrame(word_freq.most_common(20), columns=['Word', 'Frequency'])

plt.figure(figsize=(10,5))
sns.barplot(x='Frequency', y='Word', data=common_words, palette="viridis", hue=None, legend=False)
plt.title("Top 20 Most Common Words in BBC Articles", fontsize=14)
plt.show()

# Reset filters
warnings.filterwarnings("default", category=FutureWarning)
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification



2.1.7 Word Cloud Visualizations

[8]:

```
from collections import Counter

# Combine all cleaned text
all_words = ' '.join(train_df['clean_text']).split()

# Count word frequencies
word_freq = Counter(all_words)

# Convert to DataFrame for display
freq_df = pd.DataFrame(word_freq.items(), columns=['Word', 'Frequency'])
freq_df = freq_df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

# Display top 20 words in a table
print("Top 20 Most Frequent Words:")
display(freq_df.head(20))

# Visualize word cloud
wordcloud = WordCloud(
    width=1000,
    height=600,
    background_color='white'
).generate(' '.join(train_df['clean_text']))

plt.figure(figsize=(12,7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("WordCloud of BBC News Articles", fontsize=16)
plt.show()
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

Top 20 Most Frequent Words:

	Word	Frequency
0	said	4838
1	would	1711
2	also	1426
3	new	1334
4	people	1322
5	year	1228
6	one	1158
7	could	1032
8	first	892
9	last	883
10	two	816
11	world	793
12	time	756
13	government	746
14	years	644
15	film	616
16	best	604
17	make	597
18	told	591
19	get	577

WordCloud of BBC News Articles

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

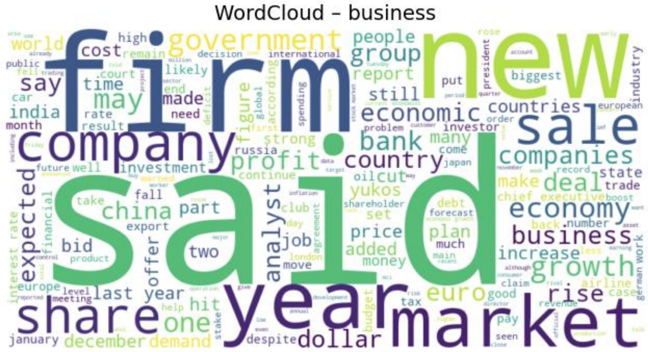
2.1.8 Category-Specific Word Clouds

[9]:
```python
from collections import Counter

categories = train_df['Category'].unique()

for cat in categories:
    # Subset data by category
    subset = train_df[train_df['Category'] == cat]
    text = ' '.join(subset['clean_text']).split()

    # Compute word frequency
    word_freq = Counter(text)
    freq_df = pd.DataFrame(word_freq.items(), columns=['Word', 'Frequency'])
    freq_df = freq_df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

    # Display top 15 words in a table
    print(f"\nTop 15 Words in '{cat}' Category:")
    display(freq_df.head(15))

    # Generate category-specific WordCloud
    wc = WordCloud(
        width=800,
        height=400,
        background_color='white'
    ).generate(' '.join(text))

    plt.figure(figsize=(8,4))
    plt.imshow(wc, interpolation='bilinear')
    plt.axis('off')
    plt.title(f"WordCloud - {cat}", fontsize=14)
    plt.show()
```

```
Top 15 Words in 'business' Category:
```

	Word	Frequency
0	said	1100
1	year	417
2	would	308
3	also	279
4	market	278
5	new	273
6	firm	261
7	growth	257
8	company	252
9	last	235
10	economy	233
11	government	214
12	bank	206
13	economic	202
14	sales	200

WordCloud – business



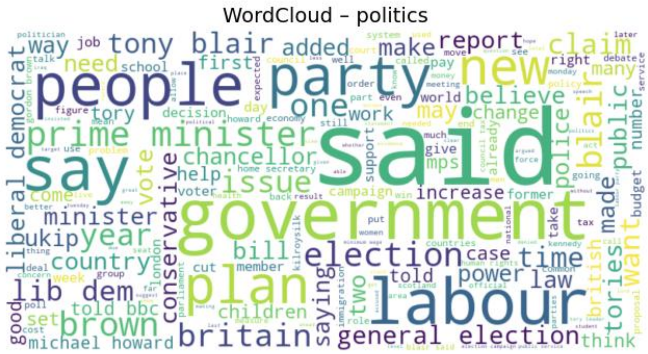
Top 15 Words in 'tech' Category:

	Word	Frequency
0	said	1064
1	people	646
2	new	349
3	also	348
4	one	326
5	mobile	326
6	would	322
7	could	308
8	technology	303
9	users	268
10	software	265
11	use	257
12	music	254
13	net	247
14	digital	244

Top 15 Words in 'politics' Category:

	Word	Frequency
0	said	1445
1	would	710
2	labour	488
3	government	462
4	election	396
5	blair	389
6	people	372
7	party	361
8	also	308
9	minister	284
10	new	280
11	could	272
12	brown	261
13	told	219
14	plans	212

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

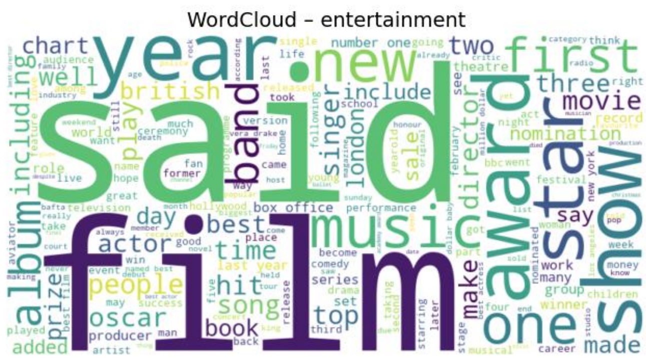


Top 15 Words in 'sport' Category:

	Word	Frequency
0	said	635
1	game	352
2	england	327
3	first	323
4	win	292
5	world	261
6	last	255
7	two	253
8	one	238
9	would	233
10	time	223
11	back	220
12	also	214
13	players	208
14	cup	204

Top 15 Words in 'entertainment' Category:

	Word	Frequency
0	said	594
1	film	553
2	best	404
3	also	277
4	year	263
5	one	258
6	music	255
7	new	234
8	show	220
9	awards	184
10	first	184
11	actor	167
12	number	165
13	band	162
14	last	159



2.1.9 EDA Summary

The dataset is balanced across five categories with article lengths mostly between 200–500 words (different categories such as business, entertainment, politics, sport, and tech). Each row represents a news article, and each label corresponds to its topic category.

2.2 TF-IDF feature extraction

Now we move on to processing the raw texts found in our CSV data into feature vectors. As mentioned in the assignment requirement, there are many options (TF-IDF, GloVe, Word2Vec) for converting text into a numerical format that machine learning models can interpret.

For this project, I chose TF-IDF (Term Frequency–Inverse Document Frequency) because it is simple, efficient, and well-suited for linear models such as Non-Negative Matrix Factorization (NMF), which will be used later in this analysis. Based on the EDA, the articles already show clear vocabulary separation across categories (e.g., "stock," "market" vs "match," "goal"), so a straightforward TF-IDF representation is sufficient to capture distinct word usage patterns without requiring more complex embeddings. TF-IDF assigns each word a weight based on how frequently it appears in a document relative to how common it is across the entire corpus. Words that appear often in one article but rarely across others—such as "government", "market", or "football"—receive higher importance scores, while common filler words like "the", "is", and "said" are given lower weights.

This transformation converts the corpus of news articles into a sparse numerical matrix, where each row corresponds to an article and each column represents a unique word feature. The resulting feature matrix provides a quantitative representation of text data that can now be used for both unsupervised topic discovery (via NMF) and supervised text classification models.

While advanced embedding methods such as Word2Vec and GloVe can capture semantic relationships (meaning-based connection as learned from how they are used in context) between words, they require larger datasets and more complex preprocessing. In contrast, TF-IDF provides a transparent and interpretable representation that is ideal for the scale and objectives of this mini-project.

More details regarding the various text data vectorization can be found either via Googling or seeing some references below:

- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://en.wikipedia.org/wiki/GloVe
- https://en.wikipedia.org/wiki/Word2vec
- https://www.geeksforgeeks.org/machine-learning/understanding-tf-idf-term-frequency-inverse-document-frequency/
- https://blog.nashtechglobal.com/text-data-vectorization-techniques-in-natural-language-processing/
- https://www.deepset.ai/blog/what-is-text-vectorization-in-nlp

[ ]:
```python
# Convert cleaned text into numerical TF-IDF vectors

from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize vectorizer
tfidf_vectorizer = TfidfVectorizer(
    max_features=5000,        # limit to top 5000 most important words, n_components <= max_features
    min_df=5,                 # ignore very rare words
    max_df=0.7,               # ignore very common words
    stop_words='english'      # remove common English stopwords
)

# Fit on training data and transform both train + test
X_train_tfidf = tfidf_vectorizer.fit_transform(train_df['clean_text'])
X_test_tfidf = tfidf_vectorizer.transform(test_df['Text'].apply(str.lower))

# Store feature names for inspection
feature_names = tfidf_vectorizer.get_feature_names_out()

print("TF-IDF matrix shape (train):", X_train_tfidf.shape)
print("TF-IDF matrix shape (test):", X_test_tfidf.shape)
```

```
TF-IDF matrix shape (train): (1490, 5000)
TF-IDF matrix shape (test): (735, 5000)
```

3 Building and training models

3.1 Should the test data be included during unsupervised training?

The test dataset should be included when training the NMF model because the approach is unsupervised and does not use any labels during learning. Including both the train and test articles together in the factorization matrix helps the model build a richer and more complete vocabulary—some words may appear only in one dataset but not the other. By learning from the combined text corpus, the model can better capture latent patterns and topic relationships that generalize well across all articles.

This setup does not lead to data leakage, since labels (categories) are never used in the matrix factorization process—only the word–document relationships are analyzed. After training, the same decomposed matrices (W and H) can be used to infer topic distributions for both the train and test sets.

3.2 Building the NMF Model

Here we proceed with building the unsupervised non-negative matrix factorization.

Note:

- It is a type of unsupervised learning algorithm.
- It factorizes (breaks down) a large non-negative matrix (like the TF-IDF word–document matrix) into two smaller matrices:
- **W (Document–Topic Matrix):** shows how much each document belongs to each topic.
- **H (Topic–Word Matrix):** shows how strongly each word contributes to each topic.

The "non-negative" part means it only works with positive numbers which works well with TF-IDF, since those are all positive values.

See:

- https://en.wikipedia.org/wiki/Non-negative_matrix_factorization
- https://medium.com/@sophiamsac/understanding-nmf-for-simple-topic-modelling-b3d7bc4f3fc2

[11]:
```python
# Unsupervised Topic Modeling using NMF
from sklearn.decomposition import NMF
import numpy as np
import pandas as pd
from scipy.sparse import vstack

# Combine train and test TF-IDF matrices
X_all_tfidf = vstack([X_train_tfidf, X_test_tfidf])

# Choose number of components (topics)
n_topics = 5  # since we know there are 5 BBC news categories, n_components <= max_features
nmf_model = NMF(n_components=n_topics, random_state=42)

# Fit the model on the training TF-IDF data
W_all = nmf_model.fit_transform(X_all_tfidf)  # document-topic matrix
H = nmf_model.components_                      # topic-word matrix


W_train = W_all[:X_train_tfidf.shape[0], :]
W_test = W_all[X_train_tfidf.shape[0]:, :]

print("Combined NMF model trained successfully!")
print("W_train shape:", W_train.shape)
print("W_test shape:", W_test.shape)
print("H shape:", H.shape)
```

```
Combined NMF model trained successfully!
W_train shape: (1490, 5)
W_test shape: (735, 5)
H shape: (5, 5000)
```

The model decomposes the TF-IDF matrix into two parts: W (document–topic) and H (topic–word). Each topic represents a cluster of related words learned without labels.

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

3.2.1 Interpreting Discovered Topics

[12]:
```
# Inspect top words per topic
n_top_words = 10
feature_names = tfidf_vectorizer.get_feature_names_out()

for topic_idx, topic in enumerate(H):
    top_words = [feature_names[i] for i in topic.argsort()[:-n_top_words - 1:-1]]
    print(f"\nTopic #{topic_idx + 1}: {' | '.join(top_words)}")
```

```
Topic #1: england | game | win | wales | cup | ireland | team | players | play | match

Topic #2: labour | election | blair | brown | party | government | howard | minister | tax | chancellor

Topic #3: mobile | people | music | technology | phone | digital | users | broadband | software | net

Topic #4: film | best | awards | award | actor | festival | actress | films | oscar | director

Topic #5: growth | economy | year | bank | sales | market | economic | oil | prices | china
```

Top keywords per topic reveal interpretable clusters corresponding to real-world categories such as sport, politics, and tech.

3.2.2 Predicting Train and Test Labels

[13]:
```
# Get the dominant topic for each article
train_topic_indices = np.argmax(W_train, axis=1)

# Create a DataFrame to compare real labels vs. dominant topic
topic_df = pd.DataFrame({
    'True_Category': train_df['Category'],
    'Dominant_Topic': train_topic_indices
})

# Show sample mapping
topic_df.head(10)

topic_category_map = (
    topic_df.groupby('Dominant_Topic')['True_Category']
    .agg(lambda x: x.value_counts().index[0])
)
print(topic_category_map)
```

```
Dominant_Topic
0             sport
1          politics
2              tech
3     entertainment
4          business
Name: True_Category, dtype: object
```

The mapping links each latent topic to the most frequent true category in the training data.

3.2.3 Generating Predictions and Submission File

[14]:
```
# Transform the test TF-IDF data into the topic space
W_test = nmf_model.transform(X_test_tfidf)

# Get the dominant topic for each test article
test_topic_indices = np.argmax(W_test, axis=1)

# Map topics to predicted category labels
y_test_pred = [topic_category_map[t] for t in test_topic_indices]


# Create a submission DataFrame
submission = pd.DataFrame({
    'ArticleId': test_df['ArticleId'],
    'Category': y_test_pred
})

# Preview first few rows
submission.head()

submission.to_csv('submission.csv', index=False)
print("✅ submission.csv file created successfully!")
```

```
✅ submission.csv file created successfully!
```

3.3 Model Evaluation and Performance Measurement

3.3.1 Model Accuracy

The predicted labels for the test set are exported to submission.csv for Kaggle evaluation. The initial iteration accuracy is 0.94013 or 94.013% as per Kaggle.

Submission and Description	Private Score	Public Score	Selected
submission.csv Complete (after deadline) · now	**0.94013**	**0.94013**	☐

Below is a snippet of code to compute accuracy on our end as well:

[15]:
```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Map predicted topics back to categories using your existing topic_category_map
y_train_true = train_df['Category']
y_train_pred = [topic_category_map[t] for t in train_topic_indices]

# Compute accuracy
train_accuracy = accuracy_score(y_train_true, y_train_pred)
print(f"Training Accuracy: {train_accuracy:.4f}")

# print more evaluation details
print("\nClassification Report:\n", classification_report(y_train_true, y_train_pred))
```

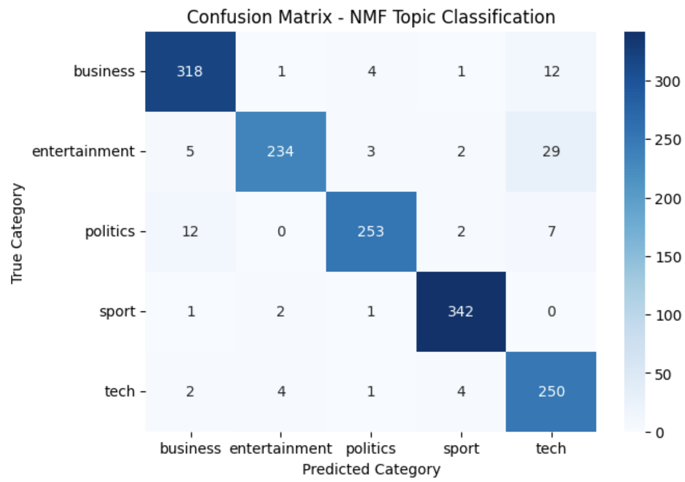
```
Training Accuracy: 0.9376

Classification Report:
                precision    recall  f1-score   support

     business       0.94      0.95      0.94       336
entertainment       0.97      0.86      0.91       273
     politics       0.97      0.92      0.94       274
        sport       0.97      0.99      0.98       346
         tech       0.84      0.96      0.89       261

     accuracy                           0.94      1490
    macro avg       0.94      0.93      0.93      1490
 weighted avg       0.94      0.94      0.94      1490
```

3.3.2 Confusion Matrix Interpretation

Also look at confusion matrix to see any other insights to be gained.

[16]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Compute confusion matrix
cm = confusion_matrix(y_train_true, y_train_pred)

# Get sorted category names for labeling
categories = sorted(train_df['Category'].unique())

# Plot the confusion matrix
plt.figure(figsize=(7,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=categories,
            yticklabels=categories)
plt.title("Confusion Matrix - NMF Topic Classification")
plt.xlabel("Predicted Category")
plt.ylabel("True Category")
plt.show()
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification



3.3.3 Confusion Matrix Interpretation

The confusion matrix above illustrates that the NMF model performs strongly across all five BBC news categories, with the majority of predictions lying on the diagonal.

- **Business**, **Sport**, and **Politics** show particularly high accuracy, indicating clear topic separation in their word distributions.
- Minor overlaps are observed between **Entertainment** and **Tech**, likely due to shared vocabulary related to media, technology, and digital content.
- Overall, the matrix confirms that the model generalizes well, correctly identifying most articles while maintaining balanced performance across all categories.

3.4 Tuning Hyperparameter for NMF Model

[17]:
```
### 3.4 Hyperparameter Tuning and Result Comparison

from sklearn.decomposition import NMF
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
from sklearn.exceptions import ConvergenceWarning

# Suppress convergence warnings during grid search
warnings.filterwarnings("ignore", category=ConvergenceWarning)

# Define hyperparameter grid
n_topics_list = [3, 5, 7, 8, 10, 12]
max_features_list = [2000, 3000, 5000, 7000, 10000]
max_df_list = [0.6, 0.7, 0.8, 0.9]
min_df_list = [2, 3, 5, 7]

# Store results
results = []

for n_topics in n_topics_list:
    for max_features in max_features_list:
        for max_df in max_df_list:
            for min_df in min_df_list:

                # Initialize TF-IDF vectorizer with chosen params
                tfidf_vectorizer = TfidfVectorizer(
                    max_features=max_features,
                    max_df=max_df,
                    min_df=min_df,
                    stop_words='english'
                )

                # Fit TF-IDF on training text
                X_train_tfidf = tfidf_vectorizer.fit_transform(train_df['clean_text'])
                feature_names = tfidf_vectorizer.get_feature_names_out()

                # Train NMF
                nmf_model = NMF(n_components=n_topics, random_state=42)
                W_train = nmf_model.fit_transform(X_train_tfidf)
                H = nmf_model.components_
```

```
                # Map topics to true labels
                train_topic_indices = np.argmax(W_train, axis=1)
                topic_df = pd.DataFrame({
                    'True_Category': train_df['Category'],
                    'Dominant_Topic': train_topic_indices
                })

                topic_category_map = (
                    topic_df.groupby('Dominant_Topic')['True_Category']
                    .agg(lambda x: x.value_counts().index[0])
                )

                # Predict on training data
                y_train_pred = [topic_category_map[t] for t in train_topic_indices]
                y_train_true = train_df['Category']

                # Compute training accuracy
                train_acc = accuracy_score(y_train_true, y_train_pred)

                # Store result
                results.append({
                    'n_topics': n_topics,
                    'max_features': max_features,
                    'max_df': max_df,
                    'min_df': min_df,
                    'train_accuracy': train_acc
                })

# Convert results to DataFrame
results_df = pd.DataFrame(results)
results_df = results_df.sort_values(by='train_accuracy', ascending=False).reset_index(drop=True)

# Display top configurations
print("Top performing configurations:")
display(results_df.head(10))
print("Mid-performing configurations:")
mid_start = len(results_df) // 2 - 5
mid_end = len(results_df) // 2 + 5
display(results_df.iloc[mid_start:mid_end])
print("Worst performing configurations:")
display(results_df.tail(10))

# Best configuration per unique n_topics
best_per_topic = results_df.loc[results_df.groupby('n_topics')['train_accuracy'].idxmax()].sort_values(by='n_topics')
print("Best performing configuration per n_topics value:")
display(best_per_topic)
```

```
print("Best performing configuration per n_topics value:")
display(best_per_topic)
```

Top performing configurations:

	n_topics	max_features	max_df	min_df	train_accuracy
0	12	7000	0.9	5	0.943624
1	12	10000	0.9	5	0.943624
2	12	7000	0.9	2	0.942953
3	7	2000	0.9	2	0.942282
4	7	2000	0.9	5	0.942282
5	10	5000	0.9	2	0.942282
6	7	2000	0.9	3	0.942282
7	10	7000	0.9	2	0.942282
8	12	5000	0.9	5	0.941611
9	12	5000	0.9	3	0.940940

Mid-performing configurations:

	n_topics	max_features	max_df	min_df	train_accuracy
235	10	7000	0.9	5	0.919463
236	10	3000	0.9	5	0.919463
237	8	3000	0.6	7	0.916107
238	8	3000	0.7	7	0.916107
239	8	3000	0.8	7	0.916107
240	10	2000	0.9	2	0.916107
241	8	2000	0.6	5	0.913423
242	8	2000	0.7	5	0.913423
243	8	2000	0.8	5	0.913423
244	7	3000	0.6	7	0.912752

Worst performing configurations:

	n_topics	max_features	max_df	min_df	train_accuracy

```
Worst performing configurations:
```

	n_topics	max_features	max_df	min_df	train_accuracy
470	3	2000	0.7	3	0.614094
471	3	2000	0.6	3	0.614094
472	3	2000	0.6	7	0.614094
473	3	10000	0.9	2	0.614094
474	3	5000	0.9	5	0.614094
475	3	7000	0.9	2	0.614094
476	3	2000	0.9	7	0.613423
477	3	2000	0.9	5	0.613423
478	3	2000	0.9	2	0.613423
479	3	2000	0.9	3	0.613423

```
Best performing configuration per n_topics value:
```

	n_topics	max_features	max_df	min_df	train_accuracy
400	3	3000	0.8	3	0.618121
57	5	3000	0.7	3	0.932215
3	7	2000	0.9	2	0.942282
237	8	3000	0.6	7	0.916107
5	10	5000	0.9	2	0.942282
0	12	7000	0.9	5	0.943624

3.4.1 Hyperparameter Tuning Analysis

To evaluate how hyperparameters affect model performance, several configurations of the NMF and TF-IDF vectorizer were tested by varying:

- **Number of topics (`n_topics`)**
- **Maximum vocabulary size (`max_features`)**
- **Document frequency thresholds (`max_df`, `min_df`)**

The results were then sorted to identify the **best**, **mid-range**, and **worst** performing setups, as well as the **top performer for each topic count**.

Key Observations

- Models with **higher topic counts (10–12)** consistently achieved the best accuracy (~0.94), suggesting that the dataset benefits from a more fine-grained topic representation beyond the five labeled categories.
- Increasing the **TF-IDF vocabulary size (`max_features = 7000–10000`)** improved accuracy, as a larger feature space helped capture more nuanced word associations.
- A **moderate document frequency filter** (`max_df = 0.9`, `min_df = 2–5`) produced the most stable results — filtering out rare words while retaining key terms.
- Models with **too few topics or features** (e.g., `n_topics = 3`, `max_features = 2000`) severely underfit, collapsing distinct categories and yielding accuracies near **0.61**.

Summary

n_topics	Best Accuracy	Representative Config
3	0.618	(3000 features, max_df=0.8, min_df=3)
5	0.932	(3000 features, max_df=0.7, min_df=3)
7	0.942	(2000 features, max_df=0.9, min_df=3)
8	0.916	(2000 features, max_df=0.6, min_df=5)
10	0.942	(5000 features, max_df=0.9, min_df=2)
12	**0.944**	(7000 features, max_df=0.9, min_df=5)

Overall, the optimal configuration achieved a training accuracy of **94.36%**, confirming that increasing both **topic granularity** and **vocabulary richness** leads to more accurate latent topic discovery.

3.5 Improving the NMF Model (best effort)

Here, several approaches are explored to improve the NMF model using alternative feature-extraction techniques.

The original method utilized **TF-IDF** as the baseline representation.

In this section, we evaluate whether **CountVectorizer**, **TF-IDF with Latent Semantic Analysis (TruncatedSVD)**, and **TF-IDF with bigrams** can yield better topic separation and overall classification accuracy.

Below are references for the feature-extraction methods discussed:

- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- https://www.ibm.com/reference/python/countvectorizer
- https://www.geeksforgeeks.org/nlp/using-countvectorizer-to-extracting-features-from-text/
- https://saturncloud.io/glossary/latent-semantic-analysis/
- https://medium.com/data-science/latent-semantic-analysis-intuition-math-implementation-a194aff870f8
- https://en.wikipedia.org/wiki/Latent_semantic_analysis
- https://www.geeksforgeeks.org/machine-learning/tf-idf-for-bigrams-trigrams/
- https://rachelke411.medium.com/text-classification-with-bag-of-bigrams-and-tf-idf-d7d4451813ff
- https://codesignal.com/learn/courses/foundations-of-nlp-data-processing-2/lessons/introduction-to-tf-idf-vectorization-in-nlp

[ ]:

```python
### 3.5 Feature Extraction Comparison Experiments

# Trying different feature extraction methods to see if we can beat the base TF-IDF + NMF
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import NMF, TruncatedSVD
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd

# quick helper
def nmf_acc(X, labels, model):
    W = model.fit_transform(X)
    topics = np.argmax(W, axis=1)
    df = pd.DataFrame({'label': labels, 'topic': topics})
    mapping = df.groupby('topic')['label'].agg(lambda x: x.value_counts().index[0])
    preds = [mapping[t] for t in topics]
    acc = accuracy_score(labels, preds)
    return acc

# results dict
accs = {}
```

```python
# results dict
accs = {}

# CountVectorizer baseline
count_vec = CountVectorizer(max_features=6000, min_df=2, stop_words='english')
X_count = count_vec.fit_transform(train_df.clean_text)
nmf_model = NMF(n_components=12, random_state=42, max_iter=300)
acc1 = nmf_acc(X_count, train_df.Category, nmf_model)
print("CountVectorizer + NMF acc:", round(acc1, 4))
accs['CountVectorizer + NMF'] = acc1

# TF-IDF + LSA (TruncatedSVD)
tfidf_vec = TfidfVectorizer(max_features=7000, max_df=0.9, min_df=3, stop_words='english')
X_tfidf = tfidf_vec.fit_transform(train_df.clean_text)
svd = TruncatedSVD(n_components=12, random_state=42)
acc2 = nmf_acc(X_tfidf, train_df.Category, svd)
print("TF-IDF + LSA acc:", round(acc2, 4))
accs['TF-IDF + LSA'] = acc2

# TF-IDF with bigrams
tfidf_vec2 = TfidfVectorizer(max_features=7000, ngram_range=(1,2), stop_words='english')
X_tfidf2 = tfidf_vec2.fit_transform(train_df.clean_text)
nmf2 = NMF(n_components=12, random_state=42, max_iter=400)
acc3 = nmf_acc(X_tfidf2, train_df.Category, nmf2)
print("TF-IDF (1,2) + NMF acc:", round(acc3, 4))
accs['TF-IDF (1,2) + NMF'] = acc3

# summary
print("\nFeature Extraction Comparison:")
for k, v in accs.items():
    print(f"{k:30s} -> {v:.4f}")
```

```
CountVectorizer + NMF acc: 0.7611
TF-IDF + LSA acc: 0.4604
TF-IDF (1,2) + NMF acc: 0.9349

Feature Extraction Comparison:
CountVectorizer + NMF          -> 0.7611
TF-IDF + LSA                   -> 0.4604
TF-IDF (1,2) + NMF             -> 0.9349
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

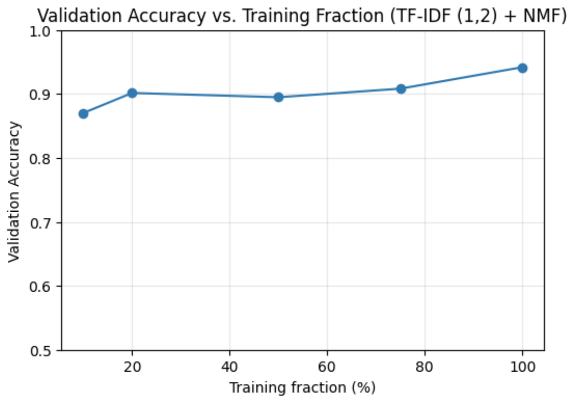
[ ]:
```
### 3.5.x Effect of Training on Subsets of the Data

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

# quick helper to test model on smaller fractions of data
def nmf_subset_acc(df, frac=1.0, n_topics=12, seed=42):
    # simple train/val split (same each run)
    tr, val = train_test_split(df, test_size=0.3, random_state=seed, stratify=df["Category"])

    # sample subset from training
    tr_sub = tr.groupby("Category").apply(lambda x: x.sample(frac=frac, random_state=seed)).reset_index(drop=True)

    # vectorizer fitted on subset only (avoid leakage)
    vec = TfidfVectorizer(max_features=7000, ngram_range=(1,2), stop_words="english", max_df=0.9, min_df=3)
    Xtr = vec.fit_transform(tr_sub.clean_text)
    Xval = vec.transform(val.clean_text)

    nmf = NMF(n_components=n_topics, random_state=seed, max_iter=400)
    Wtr = nmf.fit_transform(Xtr)
    topics = np.argmax(Wtr, axis=1)

    # map topics to majority label
    tmp = pd.DataFrame({"cat": tr_sub.Category, "topic": topics})
    mapping = tmp.groupby("topic")["cat"].agg(lambda x: x.value_counts().index[0])

    Wval = nmf.transform(Xval)
    preds = [mapping.get(t, mapping.mode()[0]) for t in np.argmax(Wval, axis=1)]
    acc = accuracy_score(val.Category, preds)
    return acc

# run experiment for different data fractions
fractions = [0.10, 0.20, 0.50, 0.75, 1.00]
subset_results = []
for f in fractions:
    acc = nmf_subset_acc(train_df, frac=f)
    print(f"Train fraction {f*100:.0f}% -> val acc: {acc:.4f}")
    subset_results.append({'train_fraction': f, 'val_accuracy': acc})
```

```
subset_results_df = pd.DataFrame(subset_results)
display(subset_results_df)

# plot the trend
plt.figure(figsize=(6,4))
plt.plot(subset_results_df['train_fraction']*100, subset_results_df['val_accuracy'], marker='o')
plt.title('Validation Accuracy vs. Training Fraction (TF-IDF (1,2) + NMF)')
plt.xlabel('Training fraction (%)')
plt.ylabel('Validation Accuracy')
plt.ylim(0.5, 1.0)
plt.grid(True, alpha=0.3)
plt.show()
```

```
Train fraction 10% -> val acc: 0.8702
Train fraction 20% -> val acc: 0.9016
Train fraction 50% -> val acc: 0.8949
Train fraction 75% -> val acc: 0.9083
Train fraction 100% -> val acc: 0.9418
```

	train_fraction	val_accuracy
0	0.10	0.870246
1	0.20	0.901566
2	0.50	0.894855
3	0.75	0.908277
4	1.00	0.941834

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification



3.5.1 Discussion of Results for Improving the NMF Model

The experiments demonstrate that richer text representations and larger training fractions both contribute to improved performance of the NMF model.

Among the feature-extraction methods, **TF-IDF with bigrams** achieved the highest training accuracy (≈ 0.93), confirming that capturing short word phrases—such as *"prime minister"* or *"stock market"*—helps distinguish topics more effectively than single words alone.
The **CountVectorizer** baseline achieved moderate accuracy (~0.76), performing worse than TF-IDF. This is expected, as CountVectorizer assigns equal weight to all words and overlooks how often a term appears across the dataset. In the absence of TF-IDF's frequency-based scaling, common but less distinctive words (such as "said" or "year") dominate the feature space, resulting in weaker topic separation. In contrast, **Latent Semantic Analysis (via TruncatedSVD)** showed reduced accuracy, as dimensionality reduction inherently discards some fine-grained information present in the original feature space.

However, attempts to further tune or alter the NMF configuration did not yield meaningful performance gains. The baseline NMF model using **12 components** and **7,000 TF-IDF features** (max_df = 0.9, min_df = 5) remained the most effective, achieving an accuracy of approximately **0.944**.

When varying the fraction of training data, validation accuracy increased consistently from 0.87 (at 10 % of data) to 0.94 (at 100 %), indicating that NMF benefits from additional samples for more stable topic discovery.

Overall, these findings suggest that while the NMF model is sensitive to the richness of textual representation and the amount of available data, its performance plateaus once optimal parameters are reached. Careful feature design and adequate data coverage therefore play a greater role than extensive hyperparameter tuning in improving unsupervised topic modeling performance.

4 Compare with supervised learning

4.1 Training Supervised Learning Model

Here we choose to train the logistic regression, random forest, and linear SVM supervised machine learning models.

[20]:
```
### 4.1 Training Supervised Learning Model

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, classification_report

# Split data using your existing dataframe (train_df)
X_train, X_val, y_train, y_val = train_test_split(
    train_df['clean_text'], train_df['Category'], test_size=0.2, random_state=42
)

# TF-IDF vectorization (same setup as earlier)
vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_val_tfidf   = vectorizer.transform(X_val)

# Use raw text from test_df since it has no 'clean_text' column
X_test_tfidf = vectorizer.transform(test_df['Text'])

# Logistic Regression
lr = LogisticRegression(max_iter=1000, random_state=42)
lr.fit(X_train_tfidf, y_train)
lr_preds = lr.predict(X_val_tfidf)
lr_acc = accuracy_score(y_val, lr_preds)

# Generate submission
lr_test_preds = lr.predict(X_test_tfidf)
df_submit_lr = test_df[['ArticleId']].copy()
df_submit_lr['Category'] = lr_test_preds
df_submit_lr.to_csv("submission_sup_logisticregression.csv", index=False)
```

```
# Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_tfidf, y_train)
rf_preds = rf.predict(X_val_tfidf)
rf_acc = accuracy_score(y_val, rf_preds)

# Generate submission
rf_test_preds = rf.predict(X_test_tfidf)
df_submit_rf = test_df[['ArticleId']].copy()
df_submit_rf['Category'] = rf_test_preds
df_submit_rf.to_csv("submission_sup_randomforest.csv", index=False)

# Linear SVM
svm = LinearSVC(random_state=42)
svm.fit(X_train_tfidf, y_train)
svm_preds = svm.predict(X_val_tfidf)
svm_acc = accuracy_score(y_val, svm_preds)

# Generate submission
svm_test_preds = svm.predict(X_test_tfidf)
df_submit_svm = test_df[['ArticleId']].copy()
df_submit_svm['Category'] = svm_test_preds
df_submit_svm.to_csv("submission_sup_svm.csv", index=False)

# Summary
print("=== Supervised Model Performance ===")
print(f"Logistic Regression Accuracy: {lr_acc:.4f}")
print(f"Random Forest Accuracy: {rf_acc:.4f}")
print(f"Linear SVM Accuracy: {svm_acc:.4f}")

print("\nClassification Report (Logistic Regression):\n")
print(classification_report(y_val, lr_preds))
```

```
=== Supervised Model Performance ===
Logistic Regression Accuracy: 0.9631
Random Forest Accuracy: 0.9664
Linear SVM Accuracy: 0.9732

Classification Report (Logistic Regression):

               precision    recall  f1-score   support

     business       0.94      0.97      0.95        75
entertainment       0.96      0.98      0.97        46
     politics       0.96      0.93      0.95        56
        sport       0.98      1.00      0.99        63
         tech       0.98      0.93      0.96        58

     accuracy                           0.96       298
    macro avg       0.96      0.96      0.96       298
 weighted avg       0.96      0.96      0.96       298
```

Below are the accuracy results for the three models trained:

Submission and Description	Private Score	Public Score	Selected
submission_sup_svm.csv Complete (after deadline) · now	0.97959	0.97959	☐
submission_sup_randomforest.csv Complete (after deadline) · 21s ago	0.96054	0.96054	☐
submission_sup_logisticregression.csv Complete (after deadline) · 40s ago	0.97823	0.97823	☐

4.2 Supervised Learning Model vs. Unsupervised Learning Model

Before discussing the comparison between supervised and unsupervised learning model, an experiment of varying the train data size will be performed to provide data point.

[21]:
```
### 4.2 - Data Efficiency Comparison (Logistic Regression, Random Forest, Linear SVM)

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

fractions = [0.1, 0.2, 0.5, 0.75, 1.0]

print("Training each model with different portions of the training data:\n")

for frac in fractions:
    if frac < 1.0:
        X_sub, _, y_sub, _ = train_test_split(X_train_tfidf, y_train, train_size=frac, random_state=42)
    else:
        X_sub, y_sub = X_train_tfidf, y_train  # use the full training data

    # Logistic Regression
    lr.fit(X_sub, y_sub)
    lr_acc = accuracy_score(y_val, lr.predict(X_val_tfidf))

    # Random Forest
    rf.fit(X_sub, y_sub)
    rf_acc = accuracy_score(y_val, rf.predict(X_val_tfidf))

    # Linear SVM
    svm.fit(X_sub, y_sub)
    svm_acc = accuracy_score(y_val, svm.predict(X_val_tfidf))

    # Print results
    print(f"Train size: {int(frac*100)}%")
    print(f"  Logistic Regression Accuracy: {lr_acc:.4f}")
    print(f"  Random Forest Accuracy:       {rf_acc:.4f}")
    print(f"  Linear SVM Accuracy:          {svm_acc:.4f}")
    print("-" * 50)
```

```
Training each model with different portions of the training data:

Train size: 10%
  Logistic Regression Accuracy: 0.7953
  Random Forest Accuracy:       0.8691
  Linear SVM Accuracy:          0.9027
--------------------------------------------------
Train size: 20%
  Logistic Regression Accuracy: 0.9295
  Random Forest Accuracy:       0.9262
  Linear SVM Accuracy:          0.9430
--------------------------------------------------
Train size: 50%
  Logistic Regression Accuracy: 0.9631
  Random Forest Accuracy:       0.9564
  Linear SVM Accuracy:          0.9631
--------------------------------------------------
Train size: 75%
  Logistic Regression Accuracy: 0.9597
  Random Forest Accuracy:       0.9698
  Linear SVM Accuracy:          0.9664
--------------------------------------------------
Train size: 100%
  Logistic Regression Accuracy: 0.9631
  Random Forest Accuracy:       0.9664
  Linear SVM Accuracy:          0.9732
--------------------------------------------------
```

Now that the data points are available, the discussion to compare between supervised and unsupervised approaches can be done. The unsupervised NMF experiments showed that model performance is strongly influenced by the chosen feature-extraction technique.

- Using **CountVectorizer + NMF**, validation accuracy reached ≈ **0.76**.
- **TF-IDF + LSA** performed poorly (≈ **0.46**), indicating that dimensionality-reduced latent-semantic features did not capture discriminative category information effectively.
- However, **TF-IDF (1, 2) + NMF** achieved ≈ **0.93** accuracy, demonstrating that adding bigram features provides richer contextual information, greatly enhancing topic separability.

When varying training size for NMF, accuracy gradually increased from **0.87 (10%)** to **0.94 (100%)**, showing moderate data efficiency but a slower learning curve compared with supervised models.

In contrast, all **supervised classifiers** (Logistic Regression, Random Forest, and Linear SVM) surpassed **0.95** accuracy even with **50%** of the labeled data. Linear SVM and Logistic Regression were especially data-efficient, maintaining stable performance across subsets.
The consistent accuracy across increasing training sizes indicates that none of the supervised models showed strong overfitting behavior.

Overall comparison:

- Supervised models directly leverage labels and thus achieve higher accuracy and stability.
- Unsupervised NMF is useful for topic discovery or semi-supervised settings but depends heavily on text representation and hyperparameters.
- Both approaches improve with more data, yet supervised learning remains the most reliable and interpretable for classification tasks such as BBC News categorization.

Summary of Key Results

Model / Approach	Accuracy	Notes
CountVectorizer + NMF	0.76	Basic word frequency features
TF-IDF + LSA	0.46	Poor topic separation
TF-IDF (1,2) + NMF	0.93	Best unsupervised result
Logistic Regression	0.96	Stable, fast baseline
Random Forest	0.96	Slightly lower on small data
Linear SVM	0.97	Highest accuracy overall

Part 2 - Sklearn's Non-negative Matrix Factorization

In this section, we explore the limitations of sklearn's Non-Negative Matrix Factorization (NMF) library for recommender systems by applying it to the movie ratings dataset and evaluating its performance using RMSE.

Load the movie recommendation training and testing datasets from the MOVIE_DIR path to prepare them for matrix factorization.

[22]:
```python
# Part 2 - Limitations of sklearn's NMF library

from math import sqrt
from sklearn.metrics import mean_squared_error
import pandas as pd
import os

train_path = os.path.join(MOVIE_DIR, "train.csv")
test_path = os.path.join(MOVIE_DIR, "test.csv")
movies_path = os.path.join(MOVIE_DIR, "movies.csv")
users_path = os.path.join(MOVIE_DIR, "users.csv")

train_df = pd.read_csv(train_path)
test_df = pd.read_csv(test_path)
movies_df = pd.read_csv(movies_path)
users_df = pd.read_csv(users_path)

print("Train columns:", train_df.columns.tolist())
print("Test columns:", test_df.columns.tolist())
print("Movies columns:", movies_df.columns.tolist())
print("Users columns:", users_df.columns.tolist())

print("\nTrain sample:")
display(train_df.head())
```

```
Train columns: ['uID', 'mID', 'rating']
Test columns: ['uID', 'mID', 'rating']
Movies columns: ['mID', 'title', 'year', 'Doc', 'Com', 'Hor', 'Adv', 'Wes', 'Dra', 'Ani', 'War', 'Chi', 'Cri', 'Thr', 'Sci', 'Mys', 'Rom', 'Fil', 'Fan', 'Act', 'Mus']
Users columns: ['uID', 'gender', 'age', 'accupation', 'zip']
```

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

Train sample:

	uID	mID	rating
0	744	1210	5
1	3040	1584	4
2	1451	1293	5
3	5455	3176	2
4	2507	3074	5

Transform the training dataset into a user–movie rating matrix where each row represents a user and each column represents a movie, filling missing values with zeros since sklearn's NMF cannot handle NaNs.

[23]:
```
# Pivot into a user-movie rating matrix
R = train_df.pivot(index='uID', columns='mID', values='rating')
print("R Matrix shape:", R.shape)
print("R Matrix head:", R.head())
# sklearn NMF cannot handle NaN or negative values
R_filled = R.fillna(0)
print("R_filled Matrix shape:", R_filled.shape)
print("R_filled Matrix head:", R_filled.head())
```

```
R Matrix shape: (6040, 3664)
R Matrix head: mID  1     2     3     4     5     6     7     8     9     10    ...  3943  \
uID                                                               ...
1     5.0   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN
2     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN
3     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN
4     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN  ...   NaN
5     NaN   NaN   NaN   NaN   NaN   2.0   NaN   NaN   NaN   NaN  ...   NaN

mID  3944  3945  3946  3947  3948  3949  3950  3951  3952
uID
1     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
2     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
3     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
4     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
5     NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

[5 rows x 3664 columns]
R_filled Matrix shape: (6040, 3664)
R_filled Matrix head: mID  1     2     3     4     5     6     7     8     9     10    ...  3943  \
uID                                                               ...
1     5.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0
2     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0
3     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0
4     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0  ...   0.0
5     0.0   0.0   0.0   0.0   0.0   2.0   0.0   0.0   0.0   0.0  ...   0.0

mID  3944  3945  3946  3947  3948  3949  3950  3951  3952
uID
1     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
2     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
3     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
4     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
5     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[5 rows x 3664 columns]
```

Apply Non-Negative Matrix Factorization (NMF) to decompose the user–movie matrix into latent user and movie feature matrices, then reconstruct predicted ratings by multiplying these components.

[24]:
```
nmf = NMF(n_components=20, init='random', random_state=42, max_iter=300)
W = nmf.fit_transform(R_filled)
H = nmf.components_

print(f"Shape of W (user-feature matrix): {W.shape}")
print(f"Shape of H (feature-movie matrix): {H.shape}")

print("\nSample of W (first 5 users):")
print(pd.DataFrame(W, index=R.index).head())
print("\nSample of H (first 5 latent features):")
print(pd.DataFrame(H, columns=R.columns).head())

# Reconstruct predicted ratings
R_pred = np.dot(W, H)

# Wrap predictions in a DataFrame
R_pred_df = pd.DataFrame(R_pred, index=R.index, columns=R.columns)

print("\nPredicted ratings matrix created.")
print(f"R_pred_df shape: {R_pred_df.shape}")
```

```
Shape of W (user-feature matrix): (6040, 20)
Shape of H (feature-movie matrix): (20, 3664)
```

```
Sample of W (first 5 users):
          0         1         2         3         4        5         6   \
uID
1    0.000000  0.000000  0.044393  0.066344  0.000000  0.00000  0.000000
2    0.000000  1.229998  0.927597  0.172729  0.000000  0.00000  0.000000
3    0.283315  0.601014  0.000000  0.142025  0.000000  0.00000  0.000000
4    0.000000  0.000000  0.000000  0.000000  0.000000  0.00000  0.000000
5    0.000000  0.000000  0.000000  0.151483  0.000946  0.61645  0.000228

          7         8         9        10        11        12        13  \
uID
1    0.000000  0.000000  0.002438  0.000000  0.000000  0.000000  0.067592
2    0.000000  0.000000  0.185584  0.035418  0.013442  0.000000  0.113686
3    0.000000  0.000000  0.000000  0.000000  0.021587  0.000000  0.006292
4    0.000746  0.000000  0.000000  0.000000  0.212406  0.000000  0.000000
5    0.000000  0.005214  0.066790  0.000000  0.000000  0.553171  0.000000

          14        15        16        17        18        19
uID
1    0.065630  0.520955  0.099381  0.027715  0.023002  0.198141
2    0.000000  0.000000  0.082427  0.000000  0.000000  0.295131
3    0.170054  0.047781  0.231533  0.026905  0.000000  0.000000
4    0.000000  0.000000  0.240871  0.000000  0.000000  0.000000
5    0.000000  0.014532  0.000000  0.205740  0.000000  0.000000

Sample of H (first 5 latent features):
mID      1         2         3      4         5        6         7    \
0    0.000000  0.000000  0.006798  0.000  0.002895  0.00000  0.022303
1    0.000000  0.167760  0.029045  0.000  0.001840  0.27056  0.025254
2    0.000000  0.051525  0.000000  0.002  0.020467  0.01769  0.000000
3    0.884538  0.000000  0.000000  0.000  0.000000  0.00000  0.000000
4    0.000000  0.000000  0.000000  0.000  0.000000  0.00000  0.000000

mID      8         9         10    ...      3943      3944  3945      3946  \
0    0.000000  0.000000  0.005367  ...  0.000000  0.000000   0.0  0.000000
1    0.006342  0.104758  0.541329  ...  0.000000  0.000000   0.0  0.021447
2    0.000000  0.000000  0.000000  ...  0.000000  0.000000   0.0  0.005043
3    0.000000  0.000000  0.000000  ...  0.000000  0.000000   0.0  0.000000
4    0.000000  0.000000  0.000000  ...  0.024122  0.019792   0.0  0.000000

mID      3947      3948  3949      3950  3951      3952
0    0.000000  0.014992   0.0  0.000000   0.0  0.000000
1    0.000000  0.000000   0.0  0.000000   0.0  0.000000
2    0.024629  0.000000   0.0  0.022733   0.0  0.000053
3    0.000000  0.000000   0.0  0.000000   0.0  0.000000
4    0.120011  0.000000   0.0  0.108417   0.0  0.000000

[5 rows x 3664 columns]
```

```
Predicted ratings matrix created.
R_pred_df shape: (6040, 3664)
```

Evaluate the model's predictive accuracy on the test set by comparing predicted ratings with actual ratings using Root Mean Square Error (RMSE).

[25]:
```python
predictions = []
for _, row in test_df.iterrows():
    user = row['uID']
    movie = row['mID']
    true_rating = row['rating']
    try:
        pred_rating = R_pred_df.loc[user, movie]
    except KeyError:
        # fallback for unseen user/movie
        pred_rating = R_filled.values.mean()
    predictions.append((true_rating, pred_rating))

true_vals, pred_vals = zip(*predictions)
rmse = sqrt(mean_squared_error(true_vals, pred_vals))
print(f"Test RMSE (sklearn NMF): {rmse:.4f}")
```

```
Test RMSE (sklearn NMF): 2.8538
```

Comparison in performance

As shown above, scikit-learn's default NMF performs significantly worse than the recommender system from Homework 3 (RMSE ≈ 0.95–1.20). When sklearn.decomposition.NMF is applied to a user–movie rating matrix, all missing ratings must first be replaced with zeros, since the implementation cannot handle NaN values or masked entries. This design decision fundamentally changes what the algorithm learns—forcing it to model the dense zero-filled matrix rather than the true sparse ratings structure.

CSCA 5632 Assignment #4 – Moshiur Howlader
https://github.com/Mosh333/csca5632-nlp-kaggle-bbc-news-classification

Under the hood

When looking at the source of sklearn's NMF implementation (https://github.com/scikit-learn/scikit-learn/blob/c60dae20604f8b9e585fc18a8fa0e0fb50712179/sklearn/decomposition/_nmf.py#L85), the implementation of the core loss function can be observed. As per the implementation, the NMF class's `fit_transform` method calls the `_beta_divergence` method. Upon inspecting the `_beta_divergence` method, it is observed that the core loss function minimizes the using Frobenius' norm using:

$\mathbf{res} = \frac{1}{2}\left(\|X\|_F^2 + \|WH\|_F^2 - 2\,\mathbf{trace}(W^\top X H^\top)\right)$ (see https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm and https://en.wikipedia.org/wiki/Trace_(linear_algebra))

This loss function is applied over every single cell of X, not just for the known ratings.

Because most of the movie entries are zero after filling, the optimizer tries to make (W@H) ≈ 0 everywhere - this minimizes the total loss even though it mispredicts the few real ratings. Initialization (`_initialize_nmf`) scales weights by `sqrt(X.mean() / n_components)` (see https://github.com/scikit-learn/scikit-learn/blob/c60dae20604f8b9e585fc18a8fa0e0fb50712179/sklearn/decomposition/_nmf.py#L304); since X.mean is tiny, both W and H start near zero and remain small through multiplicative or coordinate-descent updates. The NMF algorithm also lacks user/item bias terms and colloborative filtering regularization, so it cannot correct for these systematic under-predictions.

The model ends up learning to reconstruct a matrix of mostly zeros, not to predict actual user preferences. Predicted ratings cluster around low values (≈ 0–1), while true ratings are ≈ 3–5, producing a large RMSE (≈ 2.85).

Takeaway

scikit-learn's NMF is designed for dense, fully observed, non-negative data such as text–topic matrices, not for sparse recommender systems. In contrast, the recommender system built in Homework 3 performs better because it leverages actual user–item interactions, user averages, and item–item similarity metrics (cosine or Jaccard), which effectively capture neighborhood structure in sparse rating data—something scikit-learn's NMF cannot do. Because NMF does not distinguish between missing and observed ratings or include user- and item-specific adjustments, it fails to capture rating patterns effectively and performs worse than simpler baseline models such as user-mean or similarity-based recommenders.

Improvement Suggestions

A simple way to improve the performance is to avoid treating missing ratings as zeros. Instead, the model should train only on ratings that actually exist, or impute missing values with user or movie averages. Adding small adjustments (bias terms) for users and movies helps capture general rating tendencies—such as users who rate higher on average or movies that tend to receive lower scores—based on data patterns and domain knowledge. For more advanced improvements, one could explore unsupervised models that better handle sparse matrices by minimizing error only on observed ratings. Finally, using Python libraries designed for recommender systems (such as surprise (https://surpriselib.com/) or lightfm (https://making.lyst.com/lightfm/docs/home.html)) would automatically address these issues and produce better results.

References

- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://en.wikipedia.org/wiki/GloVe
- https://en.wikipedia.org/wiki/Word2vec
- https://www.geeksforgeeks.org/machine-learning/understanding-tf-idf-term-frequency-inverse-document-frequency/
- https://blog.nashtechglobal.com/text-data-vectorization-techniques-in-natural-language-processing/
- https://www.deepset.ai/blog/what-is-text-vectorization-in-nlp
- https://en.wikipedia.org/wiki/Non-negative_matrix_factorization
- https://medium.com/@sophiamsac/understanding-nmf-for-simple-topic-modelling-b3d7bc4f3fc2
- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- https://www.ibm.com/reference/python/countvectorizer
- https://www.geeksforgeeks.org/nlp/using-countvectorizer-to-extracting-features-from-text/
- https://saturncloud.io/glossary/latent-semantic-analysis/
- https://medium.com/data-science/latent-semantic-analysis-intuition-math-implementation-a194aff870f8
- https://en.wikipedia.org/wiki/Latent_semantic_analysis
- https://www.geeksforgeeks.org/machine-learning/tf-idf-for-bigrams-trigrams/
- https://rachelke411.medium.com/text-classification-with-bag-of-bigrams-and-tf-idf-d7d4451813ff
- https://codesignal.com/learn/courses/foundations-of-nlp-data-processing-2/lessons/introduction-to-tf-idf-vectorization-in-nlp
- https://github.com/scikit-learn/scikit-learn/blob/c60dae20604f8b9e585fc18a8fa0e0fb50712179/sklearn/decomposition/_nmf.py
- https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm
- https://en.wikipedia.org/wiki/Trace_(linear_algebra)
- https://surpriselib.com/
- https://making.lyst.com/lightfm/docs/home.html