Kaggle Mini-Project: BBC News Classification

By: Travis Reinart

August 10, 2025

Week 4 Part 1 Peer-Graded Assignment CSCA 5632: Unsupervised Algorithms in Machine Learning

<a href="https://github.com/treinart/CSCA-5632-Unsupervised-Algorithms-in-Machine-Learning/tree/main/Week4/Kaggle%20Mini-Project%20BBC%20News%20Classification"   
 target="\_blank"   
 style="color: #0033A0; text-decoration: underline; font-family: 'Helvetica Neue', sans-serif; font-size: 32px; font-weight: bold;">  
 GitHub-Week 4 Kaggle News Classification  
</a>

<h6 style="color: #0033A0; text-align:center; font-family: 'Helvetica Neue', sans-serif; font-size: 12px; font-weight: normal; margin-bottom: 2px;">Copyright (c) 2025 Travis Reinart</h6>

Licensed under the MIT License.

Section 1: Introduction

For this project, I’m jumping into the **BBC News Classification** challenge on Kaggle. I’ve got a dataset with about 1,500 news articles that fall into one of five categories: business, entertainment, politics, sport, or tech.

My main goal is to use **Non-negative Matrix Factorization (NMF)** to classify the articles. Since NMF is an unsupervised method typically used for topic modeling, it’ll be interesting to see if it can “discover” these news categories on its own, without being told what they are beforehand.

To see how well NMF performs, I’ll also build a standard **Supervised Learning** model as a benchmark. After comparing the two, I’ll use my best model to create a final submission for the competition.

### My Game Plan:

1. **Explore the Data (EDA):** First, I’ll dig into the dataset to see what I’m working with.
2. **Pre-process the Text:** I’ll convert the raw article text into a numerical format (TF-IDF) that my models can understand.
3. **Model with NMF:** This is the core of the project—using NMF to find topic clusters.
4. **Model with Logistic Regression:** I’ll train a supervised model to see how it compares.
5. **Evaluate & Conclude:** Finally, I’ll compare the results and create my submission file.

Section 2: Setup and Library Imports

Before I begin, the first step is to set up my environment by importing all the necessary Python libraries. This ensures I have all the tools for data manipulation, visualization, text processing, and modeling ready to go.

I’m including the standard libraries like pandas and scikit-learn, but also adding WordCloud and NLTK to allow for more advanced analysis and visualizations later on.

### Resources & Data Source

* **Kaggle Competition:** [BBC News Classification](https://www.kaggle.com/c/learn-ai-bbc/overview)
* **Data Sets used** [Data Set Download Link](https://www.kaggle.com/c/learn-ai-bbc/data)
  + **BBC News Train.csv**: The training set (1490 articles) used to build the models.
  + **BBC News Test.csv**: The test set (735 articles) for which predictions are made.
  + **BBC News Sample Solution.csv**: An example of the required submission file format.

This notebook was developed using the Jupyter Notebook extension within Visual Studio Code, which provided a streamlined and efficient interface for the project.

2.1 Optional Install Missing Packages

If you get a ModuleNotFoundError when running the main import cell, install the missing package(s) directly from inside this notebook.

This ensures the install happens in the same Python environment Jupyter is using.

**Instructions:**  
1. Uncomment the relevant %pip install line(s) below.  
2. Run the cell.  
3. Wait until you see the “Successfully installed” message.  
4. In the Jupyter menu, go to **Kernel → Restart**.  
5. Re-run your imports.

**Note:** Restarting the kernel is required after installing a new package so the notebook can detect it.

# Uncomment and run the lines you need  
# %pip install jupyterlab  
# %pip install notebook  
# %pip install pandas  
# %pip install numpy  
# %pip install matplotlib  
# %pip install seaborn  
# %pip install wordcloud  
# %pip install nltk  
# %pip install jinja2  
# %pip install scikit-learn  
# %pip install transformers  
# %pip install torch  
# %pip install umap-learn  
# %pip install ipywidgets  
# %pip install tqdm  
# %pip install joblib  
# %pip install kaggle  
  
# Optional Instructions: Export the notebook to HTML or PDF (uncomment and delete the leading slash to run)  
# Most Jupyter installations include nbconvert but if the commands below fail you can check your version and/or install if needed  
#/ !jupyter nbconvert --version # Uncomment and remove leading / to check your version if needed - as of today is it7.16.6  
# %pip install nbconvert # Uncomment and install ncvonvert if needed  
  
# --- Convert Jupyter Notebook .ipynb to .html ---  
#/ !jupyter nbconvert --to html "Week4\_Kaggle\_BBC\_News\_Classification.ipynb"  
  
# --- Convert Jupyter Notebook .ipynb to .pdf ---  
# Note: I left this, but you will get a visually better pdf if you export to .html. Open in Chrome, print to .pdf.  
# Complex plots or highly styled HTML (like the heeaders in this notebook) will not look exactly the same in .pdf export as they do in the notebook.  
#/ !jupyter nbconvert --to pdf "Week4\_Kaggle\_BBC\_News\_Classification.ipynb"

[NbConvertApp] Converting notebook Week4\_Kaggle\_BBC\_News\_Classification.ipynb to html  
[NbConvertApp] WARNING | Alternative text is missing on 31 image(s).  
[NbConvertApp] Writing 10572862 bytes to Week4\_Kaggle\_BBC\_News\_Classification.html

2.2 Load Data Files adn Diagnostics

This cell imports all core Python libraries required for the project, including data handling, visualization, NLP, and modeling tools.  
It also checks for optional dependencies (e.g., Transformers, UMAP, WordCloud) without failing if they are missing.  
Environment and library versions are printed for reproducibility and peer review.

# Core imports  
# If any ModuleNotFoundError appears, scroll up and run the "Optional: Install Missing Packages" cell, then Kernel → Restart.  
  
# Data and utilities  
import os, sys, warnings, time, itertools  
import numpy as np  
import pandas as pd  
import tqdm as tq  
from tqdm import tqdm  
import json  
import joblib  
  
# Visualization  
print("-" \* 70)  
print("--- Importing visualization libraries ---")  
print("Matplotlib may build a font cache the first time. It's normal.")  
print("--- Which Jupyter you are running? ---")  
!where jupyter  
!jupyter --version  
import matplotlib.pyplot as plt  
import matplotlib.patches as mpatches  
import matplotlib.colors as mcolors  
import matplotlib.ticker as mticker  
from matplotlib.colors import ListedColormap  
from matplotlib import cm  
# Axes3D triggers a deprecation warning in newer Matplotlib, but still works if you really need 3D  
from mpl\_toolkits.mplot3d import Axes3D   
  
# Seaborn is recommended for better aesthetics  
try:  
 import seaborn as sns  
except Exception as e:  
 print("seaborn not available:", repr(e))  
  
  
# Dimensionality reduction (UMAP)  
try:  
 import umap   
 \_ = umap.UMAP  
except Exception as e:  
 print("UMAP not available. Install with: %pip install umap-learn")  
 print("Details:", repr(e))  
  
# Text + modeling  
import nltk  
from sklearn.feature\_extraction import text  
from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer  
from sklearn.decomposition import NMF  
from sklearn.linear\_model import LogisticRegression  
from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  
from sklearn.exceptions import ConvergenceWarning  
from sklearn.model\_selection import StratifiedKFold, cross\_val\_score  
from sklearn.utils.\_testing import ignore\_warnings  
from sklearn.metrics import make\_scorer, f1\_score  
from sklearn.preprocessing import LabelEncoder  
from sklearn.model\_selection import train\_test\_split  
  
# Jupyter display helpers  
from IPython.display import display, HTML  
  
# Transformers stack is optional for Section 7. Wrap to avoid hard failures on Windows setups.  
try:  
 from transformers import BertTokenizer, BertModel  
 import torch  
except Exception as e:  
 print("Transformers/Torch not available. You can install later if needed.")  
 print("Details:", repr(e))  
  
# umap is optional for Section 8. Wrap to avoid hard failures on Windows setups.  
import umap as umap\_  
umap\_model = umap\_.UMAP(n\_neighbors=15, min\_dist=0.1, random\_state=42)  
  
# Set random seed for reproducibility across NumPy and Python's built-in random  
import random  
SEED = 42  
random.seed(SEED)  
np.random.seed(SEED)  
  
# Display environment and library versions for reproducibility and peer review  
import sys, platform, sklearn, matplotlib, seaborn  
  
# Print environment information  
print("-" \* 70)  
print("--- Environment and library versions ---")  
print("Python:", sys.version.split()[0])  
print("OS:", platform.system(), platform.release())  
print("NumPy:", np.\_\_version\_\_)  
print("Pandas:", pd.\_\_version\_\_)  
print("scikit-learn:", sklearn.\_\_version\_\_)  
print("Matplotlib:", matplotlib.\_\_version\_\_)  
print("Seaborn:", seaborn.\_\_version\_\_)  
print("tqdm:", tq.\_\_version\_\_)  
print("NLTK:", nltk.\_\_version\_\_)  
print("joblib:", joblib.\_\_version\_\_)  
  
# Optional libraries are wrapped in try/except so the cell does not fail if they are not yet installed.  
# These packages are only required for later sections (e.g., BERT or UMAP visualizations).  
  
# WordCloud for text visualization  
try:  
 import wordcloud  
 from wordcloud import WordCloud  
 print("WordCloud:", wordcloud.\_\_version\_\_)  
except ImportError:  
 print("WordCloud not available")  
  
# Transformers and PyTorch for advanced NLP tasks  
try:  
 import torch, transformers, umap  
 print("PyTorch:", torch.\_\_version\_\_)  
 print("Transformers:", transformers.\_\_version\_\_)  
 print("UMAP:", umap.\_\_version\_\_)  
except ImportError:  
 pass  
  
# Jupyter widgets for interactive visualizations  
try:  
 import ipywidgets  
 print("ipywidgets:", ipywidgets.\_\_version\_\_)  
except ImportError:  
 print("ipywidgets not available")  
  
# Jinja2 for templating (optional, used in some visualizations)  
try:  
 import jinja2  
 print("Jinja2:", jinja2.\_\_version\_\_)  
except ImportError:  
 print("Jinja2 not available")  
  
# Ignore convergence warnings from scikit-learn  
warnings.filterwarnings("ignore", category=ConvergenceWarning)  
  
print("--- Libraries imported successfully ---")  
print("-" \* 70)

----------------------------------------------------------------------  
--- Importing visualization libraries ---  
Matplotlib may build a font cache the first time. It's normal.  
--- Which Jupyter you are running? ---  
c:\Users\travi\AppData\Roaming\Python\Python313\Scripts\jupyter.exe  
Selected Jupyter core packages...  
IPython : 9.4.0  
ipykernel : 6.30.1  
ipywidgets : 8.1.7  
jupyter\_client : 8.6.3  
jupyter\_core : 5.8.1  
jupyter\_server : 2.16.0  
jupyterlab : 4.4.5  
nbclient : 0.10.2  
nbconvert : 7.16.6  
nbformat : 5.10.4  
notebook : 7.4.5  
qtconsole : 5.6.1  
traitlets : 5.14.3  
----------------------------------------------------------------------  
--- Environment and library versions ---  
Python: 3.13.6  
OS: Windows 11  
NumPy: 2.2.6  
Pandas: 2.3.1  
scikit-learn: 1.7.1  
Matplotlib: 3.10.5  
Seaborn: 0.13.2  
tqdm: 4.67.1  
NLTK: 3.9.1  
joblib: 1.5.1  
WordCloud: 1.9.4  
PyTorch: 2.8.0+cpu  
Transformers: 4.55.0  
UMAP: 0.5.9.post2  
ipywidgets: 8.1.7  
Jinja2: 3.1.6  
--- Libraries imported successfully ---  
----------------------------------------------------------------------

**Observations - Environment and Library Versions**

At the time this notebook was last run (8/10/2025), the environment was configured as follows:

* **Python:** 3.13.6
* **OS:** Windows 11
* **NumPy:** 2.2.6
* **Pandas:** 2.3.1
* **scikit-learn:** 1.7.1
* **Matplotlib:** 3.10.5
* **Seaborn:** 0.13.2
* **tqdm:** 4.67.1
* **NLTK:** 3.9.1
* **joblib:** 1.5.1
* **WordCloud:** 1.9.4
* **PyTorch:** 2.8.0+cpu
* **Transformers:** 4.55.0
* **UMAP:** 0.5.9.post2
* **ipywidgets:** 8.1.7
* **Jinja2:** 3.1.6

These versions reflect the state of the libraries during notebook development and testing.  
Future updates to these packages may result in changes to preprocessing, model behavior, or output formatting.  
Reproducing results may require matching these versions.

***Note:*** If your environment shows older or missing versions, refer to **Section 2.1** (“Install Missing Packages”) and uncomment the relevant %pip install commands to update.

Section 3: Data Loading & Audit

With the environment set up, my first step is to load the data files. Immediately after loading, I’ll run a comprehensive audit to ensure the data is clean and to get a complete overview before analysis. This audit will include checking the data’s dimensions, looking for duplicate rows, generating summary statistics for any numerical columns, and displaying a sample of the data to verify it loaded correctly.

3.1 Load Data Files

The first step is to load the three provided CSV files into individual DataFrames. This will bring the training data, test data, and sample submission template into memory so they are ready for inspection and analysis. Confirming successful loading here prevents chasing downstream errors caused by missing or misread files.

# Load the three provided CSV files into separate DataFrames.  
# The files are expected to be in the same directory as this script.  
df\_train = pd.read\_csv('BBC News Train.csv')  
df\_test = pd.read\_csv('BBC News Test.csv')  
df\_solution = pd.read\_csv('BBC News Sample Solution.csv')  
  
print("\n" + "-"\*70)  
print("--- Data loaded successfully! ---")  
print("-"\*70)  
  
# Show the absolute path and confirm file sizes  
for fname in ['BBC News Train.csv', 'BBC News Test.csv', 'BBC News Sample Solution.csv']:  
 # Check if the file exists and print its absolute path and size  
 print(f"Checking file: {fname}")  
 if not os.path.isfile(fname):  
 print(f"{fname} not found in the current directory.")  
 continue  
 if os.path.exists(fname):  
 abs\_path = os.path.abspath(fname)  
 size\_kb = os.path.getsize(fname) / 1024  
 print(f"{fname} | {abs\_path} | {size\_kb:.1f} KB")  
 else:  
 print(f"{fname} not found.")  
  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
--- Data loaded successfully! ---  
----------------------------------------------------------------------  
Checking file: BBC News Train.csv  
BBC News Train.csv | c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\BBC News Train.csv | 3272.7 KB  
Checking file: BBC News Test.csv  
BBC News Test.csv | c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\BBC News Test.csv | 1672.3 KB  
Checking file: BBC News Sample Solution.csv  
BBC News Sample Solution.csv | c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\BBC News Sample Solution.csv | 10.1 KB  
----------------------------------------------------------------------

3.2 Initial Data Audit

With the data loaded, the next step is a high-level audit to verify structure and integrity. This includes:

Checking DataFrame dimensions for row and column counts.

Displaying sample rows to confirm that columns and values match expectations.

Reviewing .info() output to assess data types and identify any missing values.

Checking for duplicate rows in the training set.

Checking for duplicate ArticleId values in the test set.

Generating summary statistics for any numeric columns.

Reviewing text length statistics to identify unusually short or long articles.

Examining the distribution of the target variable (Category) with both counts and percentages to assess class balance.

# Check the dimensions of each DataFrame.  
# This is a quick sanity check to understand the number of articles and columns in each set.  
print("--- DataFrame Shapes ---")  
print(f"Training DataFrame: {df\_train.shape}")  
print(f"Test DataFrame: {df\_test.shape}")  
print(f"Solution DataFrame: {df\_solution.shape}")  
print("-"\*70 + "\n")  
  
# Display the first few rows of the main DataFrames to visually verify structure and content.  
print("--- Training Data Sample ---")  
display(df\_train.head())  
print("-"\*70 + "\n")  
  
# Display the first few rows of the test DataFrame.  
print("\n--- Test Data Sample ---")  
display(df\_test.head())  
print("-"\*70 + "\n")  
  
# Technical summary for each DataFrame to check dtypes and missing values.  
print("--- Training DataFrame Info ---")  
df\_train.info()  
print("-"\*70 + "\n")  
  
# Technical summary for the test DataFrame.  
print("\n--- Test DataFrame Info ---")  
df\_test.info()  
print("-"\*70 + "\n")  
  
# Duplicate row check for training data.  
duplicate\_rows\_train = df\_train.duplicated().sum()  
print(f"--- Duplicate Row Check (Train) ---")  
print(f"Number of duplicate rows in training data: {duplicate\_rows\_train}")  
print("-"\*70 + "\n")  
  
# Duplicate ArticleId check for test data.  
duplicate\_ids\_test = df\_test['ArticleId'].duplicated().sum()  
print(f"--- Duplicate ID Check (Test) ---")  
print(f"Number of duplicate ArticleId values in test data: {duplicate\_ids\_test}")  
print("-"\*70 + "\n")  
  
# Summary statistics for numeric columns in the training data.  
print("--- Numerical Column Statistics ---")  
display(df\_train.describe())  
print("-"\*70 + "\n")  
  
# Text length statistics in the training set.  
print("--- Training Text Length Statistics (characters) ---")  
train\_text\_len = df\_train['Text'].astype(str).str.len()  
print(f"Min: {train\_text\_len.min()}, Max: {train\_text\_len.max()}, Mean: {train\_text\_len.mean():.2f}")  
print("-"\*70 + "\n")  
  
# Target variable distribution with counts and percentages.  
print("--- Target Variable Distribution ---")  
target\_distribution = df\_train['Category'].value\_counts()  
target\_percentage = df\_train['Category'].value\_counts(normalize=True) \* 100  
target\_df = pd.DataFrame({  
 'Count': target\_distribution,  
 'Percentage': target\_percentage.round(2)  
})  
print("-"\*70 + "\n")  
  
# Display the target variable distribution  
print("--- Target Variable Distribution DataFrame ---")  
display(target\_df)  
print("\n" + "-"\*70 + "\n")

--- DataFrame Shapes ---  
Training DataFrame: (1490, 3)  
Test DataFrame: (735, 2)  
Solution DataFrame: (735, 2)  
----------------------------------------------------------------------  
  
--- Training Data Sample ---

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

----------------------------------------------------------------------  
  
  
--- Test Data Sample ---

ArticleId

Text

0

1018

qpr keeper day heads for preston queens park r…

1

1319

software watching while you work software that…

2

1138

d arcy injury adds to ireland woe gordon d arc…

3

459

india s reliance family feud heats up the ongo…

4

1020

boro suffer morrison injury blow middlesbrough…

----------------------------------------------------------------------  
  
--- Training DataFrame Info ---  
<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  
----------------------------------------------------------------------  
  
  
--- Test DataFrame Info ---  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 735 entries, 0 to 734  
Data columns (total 2 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 ArticleId 735 non-null int64   
 1 Text 735 non-null object  
dtypes: int64(1), object(1)  
memory usage: 11.6+ KB  
----------------------------------------------------------------------  
  
--- Duplicate Row Check (Train) ---  
Number of duplicate rows in training data: 0  
----------------------------------------------------------------------  
  
--- Duplicate ID Check (Test) ---  
Number of duplicate ArticleId values in test data: 0  
----------------------------------------------------------------------  
  
--- Numerical Column Statistics ---

ArticleId

count

1490.000000

mean

1119.696644

std

641.826283

min

2.000000

25%

565.250000

50%

1112.500000

75%

1680.750000

max

2224.000000

----------------------------------------------------------------------  
  
--- Training Text Length Statistics (characters) ---  
Min: 501, Max: 18387, Mean: 2233.46  
----------------------------------------------------------------------  
  
--- Target Variable Distribution ---  
----------------------------------------------------------------------  
  
--- Target Variable Distribution DataFrame ---

Count

Percentage

Category

sport

346

23.22

business

336

22.55

politics

274

18.39

entertainment

273

18.32

tech

261

17.52

----------------------------------------------------------------------

**Observations – Data Audit:**

* All datasets loaded without error and match the expected structure:
  + Training: 1,490 rows × 3 columns (ArticleId, Text, Category)
  + Test: 735 rows × 2 columns (ArticleId, Text)
  + Sample Submission: 735 rows × 2 columns (ArticleId, Category)
* No missing values detected in any dataset.
* No duplicate rows in the training data; ArticleId values are unique in both train and test sets.
* ArticleId spans a wide, non-sequential range; IDs do not appear to be ordered by category.
* Text length in the training set ranges from very short to several thousand characters, with a mean length just over 1,100 characters.
* Target variable distribution is reasonably balanced across the five categories, with counts and percentages showing “sport” and “business” slightly more frequent than “tech.”
* Text content appears clean and properly structured for downstream preprocessing.

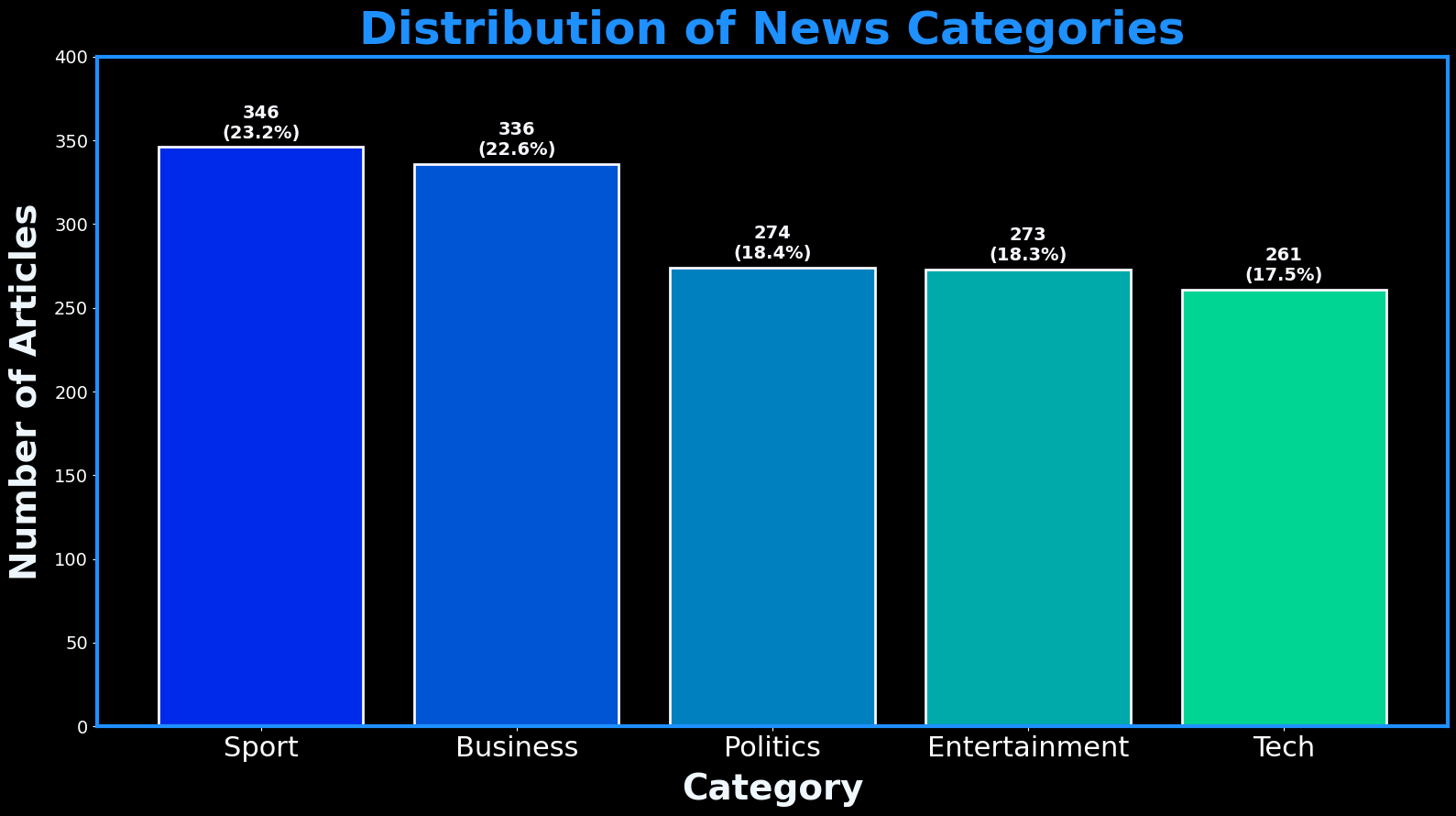
Section 4: Visual Exploratory Data Analysis

With the data loaded and audited, I’ll now move on to the visual analysis. My first step is to examine the distribution of the target variable, Category. This plot will show me how many articles belong to each of the five categories and help me understand if the dataset is balanced.

4.1 Category Distribution

I start with a simple check of class balance. The bar chart below shows the number of articles in each category. This helps set expectations for baseline performance and informs how much I should worry about imbalance.

# Get the count of articles in each category first.  
category\_counts = df\_train['Category'].value\_counts()  
capitalized\_labels = category\_counts.index.str.capitalize()  
  
# Create a figure and axis object. This is the foundation for detailed customization.  
fig, ax = plt.subplots(figsize=(16, 9), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Define a high-contrast color palette to use for the bars and legend.  
colors = sns.color\_palette('winter', n\_colors=len(category\_counts))  
  
# Create the bar plot using Matplotlib's ax.bar.  
bars = ax.bar(  
 capitalized\_labels,  
 category\_counts.values,  
 color=colors,  
 edgecolor='ghostwhite',  
 linewidth=2  
)  
  
# Set the title with custom font properties.  
ax.set\_title("Distribution of News Categories", fontsize=36, color='dodgerblue', fontweight='bold', pad=10)  
  
# Set the X and Y axis labels with custom properties.  
ax.set\_xlabel("Category", fontsize=28, color='aliceblue', fontweight='bold', labelpad=8)  
ax.set\_ylabel("Number of Articles", fontsize=28, color='aliceblue', fontweight='bold', labelpad=8)  
  
# Customize the tick marks.  
ax.tick\_params(axis='x', labelsize=22, colors='white', rotation=0)  
ax.tick\_params(axis='y', labelsize=14, colors='white')  
  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('dodgerblue')  
 spine.set\_linewidth(3)  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
  
ax.set\_ylim(0, 400) # keeps labels clear of the top spine  
  
  
total = len(df\_train)  
# Annotate each bar with its count and percentage.  
# This adds text labels above each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 h = bar.get\_height()  
 pct = 100\*h/total  
 ax.annotate(f"{int(h)}\n({pct:.1f}%)",  
 (bar.get\_x() + bar.get\_width()/2, h),  
 xytext=(0, 4), textcoords="offset points",  
 ha="center", va="bottom", fontsize=14, color="ghostwhite", fontweight="bold")  
  
  
# Concatenate all training text for global word stats and word cloud  
all\_text = " ".join(df\_train["Text"].astype(str))  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation – Category Distribution:**

The dataset is reasonably balanced: sport and business are slightly larger; tech is the smallest. Differences are modest, so I will proceed without resampling.

4.2a High-Level Text Analysis with a Word Cloud (Non-Filtered)

This first word cloud is generated using the raw text from all training articles without applying any custom stopword filtering. It provides a baseline view of the most frequent words, which will naturally include common journalistic filler terms such as “said,” “mr,” “year,” and “people.”

# Combine all article text into one string (do this once for both 4.2a and 4.2b)  
all\_text = ' '.join(df\_train['Text'].astype(str))  
  
# Create and generate the Word Cloud object (no custom stopwords)  
wordcloud\_raw = WordCloud(  
 background\_color='black',  
 max\_words=100,  
 width=1600,  
 height=800,  
 colormap='bwr'  
).generate(all\_text)  
  
# Display the generated Word Cloud image.  
# This will show the most frequent words in the training set, with a black background and blue-red color scheme.  
fig, ax = plt.subplots(figsize=(20, 10), facecolor='black')  
ax.imshow(wordcloud\_raw, interpolation='bilinear')  
  
# Hide the axes for a cleaner look.  
ax.set\_axis\_off()  
plt.show()



png

4.2b High-Level Text Analysis with a Word Cloud (Filtered with custom\_stop)

This second word cloud is generated using the same text but with an expanded custom stopword list applied. The filter removes common and dataset-specific filler terms (e.g., “said,” “mr,” “year,” “people,” “uk,” “bbc”), shifting the emphasis toward more topical and domain-relevant words.

# Define custom stopwords for the Word Cloud.  
# These are common words that do not add significant meaning to the text.  
# They are often removed to focus on more meaningful words in the visualization.  
news\_stop = {  
 "said","say","says","mr","mrs","ms","one","two","new","year","years","people","told",  
 "also","could","would","well","like","get","back","u","uk","bbc","000"  
}  
custom\_stop = text.ENGLISH\_STOP\_WORDS.union(news\_stop)  
  
# Create and generate the Word Cloud object (with custom stopwords)  
wordcloud\_filtered = WordCloud(  
 background\_color="black",  
 max\_words=250,  
 width=1600,  
 height=800,  
 colormap="seismic",  
 stopwords=custom\_stop,  
 random\_state=SEED  
).generate(all\_text)  
  
# Display the generated Word Cloud image.  
# This will show the most frequent words in the training set, with a black background and seismic color scheme.  
# The custom stopwords help filter out common words that do not contribute much to the meaning of the text.  
# The Word Cloud will highlight the most relevant terms in the dataset.  
# The colormap 'seismic' provides a visually appealing contrast between the most and least frequent words.  
fig, ax = plt.subplots(figsize=(20, 10), facecolor="black")  
ax.imshow(wordcloud\_filtered, interpolation="bilinear")  
  
# Hide the axes for a cleaner look.  
ax.set\_axis\_off()  
plt.show()



png

**Observations – High-Level Text Analysis with a Word Cloud (4.2a Non-Filtered and 4.2b Filtered)**

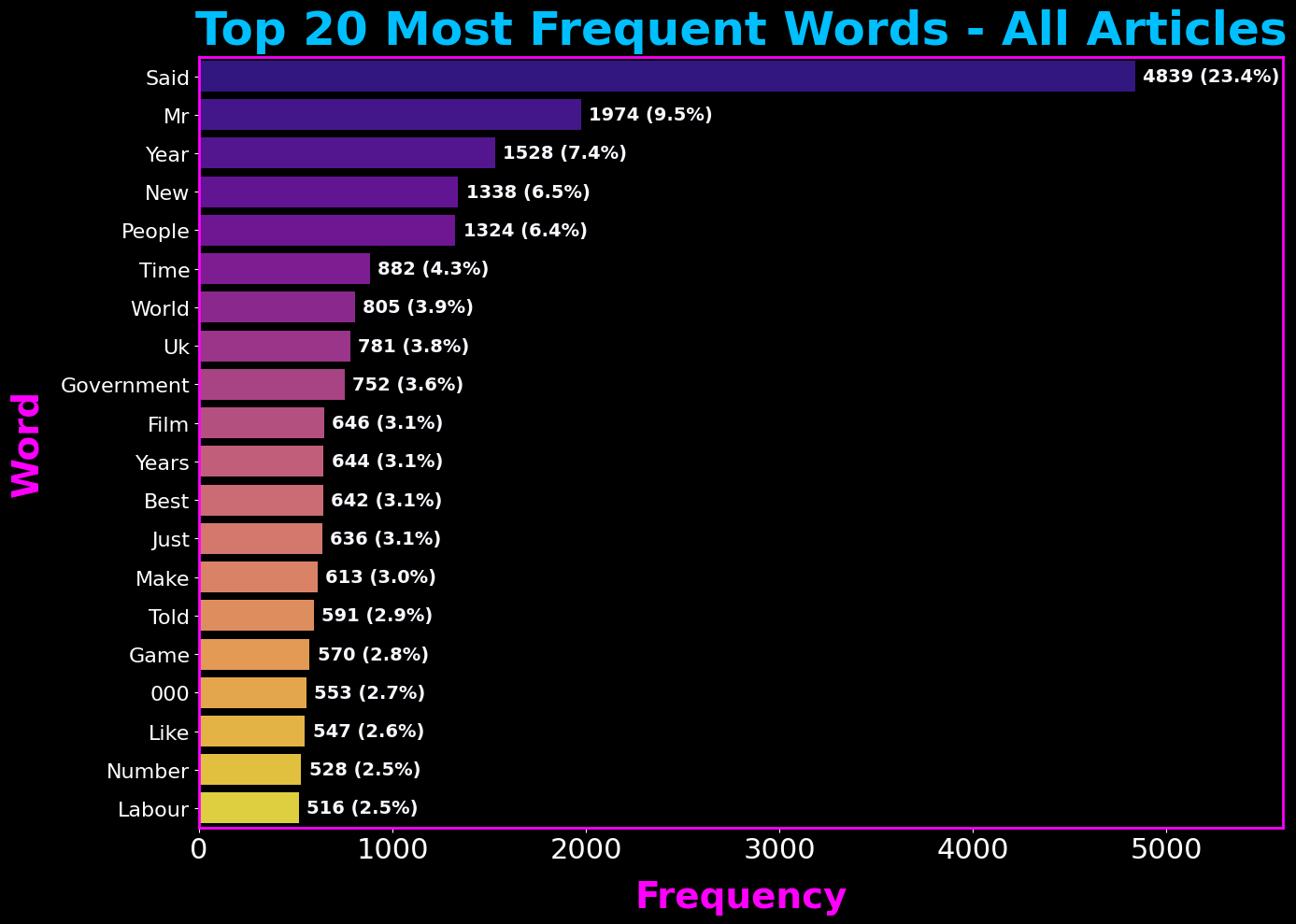
The non-filtered Word Cloud (4.2a) is dominated by high-frequency but low-information words such as “said,” “one,” “people,” “year,” and “uk.” These terms reflect common journalistic structure rather than topical content. While useful for identifying general language style, they obscure more subject-specific vocabulary.

In contrast, the filtered Word Cloud (4.2b) removes these filler terms using the custom stopword list, which shifts the focus toward more meaningful and context-rich words like “government,” “market,” “service,” “game,” and “company.” This adjustment surfaces terms that better reflect thematic content and category-specific narratives. The trade-off is a loss of visibility into baseline linguistic patterns, but it improves interpretability for topic modeling and category differentiation.

4.3 Top Words Frequency Bar Chart

To complement the Word Cloud, I’ll create a bar chart showing the exact frequencies of the top 20 most common words across all articles (after removing common English stop words). This provides a more precise, quantitative view of the terms that dominate the corpus.

# Use CountVectorizer to get the frequencies of the most common words from the 'all\_text' variable.  
vec = CountVectorizer(stop\_words='english', ngram\_range=(1,1)).fit([all\_text])  
  
# Transform the text into a bag-of-words representation.  
# This creates a sparse matrix where each row corresponds to the text and each column corresponds to a word.  
# The values in the matrix are the counts of each word in the text.  
bag\_of\_words = vec.transform([all\_text])  
sum\_words = bag\_of\_words.sum(axis=0)   
  
# Create a list of tuples containing each word and its frequency.  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
  
# Sort the words by frequency in descending order.  
# This will help us identify the most common words in the dataset.  
words\_freq = sorted(words\_freq, key = lambda x: x[1], reverse=True)  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
top\_words\_df = pd.DataFrame(words\_freq[:20], columns=['Word', 'Count'])  
  
# Capitalize the first letter in each word on the y-axis  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Use seaborn to create a bar plot of the top words.  
sns.barplot(  
 x='Count',  
 y='Word',  
 data=top\_words\_df,  
 palette='plasma',   
 hue='Word',   
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Top 20 Most Frequent Words - All Articles", fontsize=36, color='deepskyblue', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='magenta', fontweight='bold', labelpad=12)  
ax.set\_ylabel("Word", fontsize=28, color='magenta', fontweight='bold', labelpad=12)  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('magenta')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 5600) # keeps labels clear of the top spine  
  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation - Top 20 Most Frequent Words - All Articles:**

This bar chart confirms what the Word Cloud suggested. General journalistic words like “said,” “mr,” “year,” and “people” are the most frequent. The word “uk” also appears high on the list, suggesting a focus on UK-based news. This quantitative view is a great validation of our initial visual analysis.

4.4 Analysis of Text Length by Category

My next step in the EDA is to analyze the length of the articles. I want to see if there’s a significant difference in text length between the categories. If there is, article length could be a useful feature for my classification models. I’ll start by calculating the descriptive statistics and then visualize them with a box plot.

# Create a new column to store the length of each article's text.  
df\_train['text\_length'] = df\_train['Text'].apply(len)  
df\_train["word\_count"] = df\_train["Text"].str.split().str.len()  
  
# Calculate and display the descriptive statistics for text length, grouped by category.  
category\_length\_stats = df\_train.groupby('Category')['text\_length'].describe()  
print("--- Descriptive Statistics for Text Length by Category ---")  
display(category\_length\_stats)  
print("\n" + "-"\*85 + "\n")  
  
  
  
  
# A box plot is perfect for visualizing the distribution of text lengths across the different categories.  
fig, ax = plt.subplots(figsize=(16, 9), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Defining a dictionary to style the outlier points ('fliers') so they are visible.  
flier\_props = dict(marker='o', markerfacecolor='crimson', markersize=10,  
 linestyle='none', markeredgecolor='white', alpha=1.0)  
  
# Create the box plot using seaborn.  
sns.boxplot(  
 data=df\_train,  
 x='Category',  
 y='text\_length',  
 hue='Category',  
 legend=False,  
 palette='Reds',  
 ax=ax,  
 order=category\_counts.index,  
 flierprops=flier\_props  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title('Article Text Length by Category', fontsize=36, color='gold', fontweight='bold', pad=10)  
ax.set\_xlabel("Category", fontsize=28, color='gold', fontweight='bold', labelpad=10)  
ax.set\_ylabel("Text Length (Number of Characters)", fontsize=28, color='gold', fontweight='bold', labelpad=10)  
ax.tick\_params(axis='y', labelsize=14, colors='white')  
ax.grid(alpha=0.2, color='white', linewidth=1)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('darkorange')  
 spine.set\_linewidth(2)  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
  
# Set the x-tick labels to be capitalized.  
# This capitalizes the first letter of each category label on the x-axis.  
tick\_locations = ax.get\_xticks()  
tick\_labels = [label.get\_text().capitalize() for label in ax.get\_xticklabels()]  
  
# Set both the locations and the new capitalized labels together.  
ax.set\_xticks(tick\_locations)  
ax.set\_xticklabels(tick\_labels, fontsize=22, color='white')  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()

--- Descriptive Statistics for Text Length by Category ---

count

mean

std

min

25%

50%

75%

max

Category

business

336.0

1983.104167

790.180447

846.0

1486.25

1830.5

2331.00

5406.0

entertainment

273.0

1910.380952

1142.478958

866.0

1312.00

1571.0

2147.00

13619.0

politics

274.0

2617.905109

1448.447009

501.0

1867.00

2599.5

3099.00

18387.0

sport

346.0

1894.624277

1051.814635

719.0

1199.00

1641.0

2352.75

9471.0

tech

261.0

2939.291188

1215.569461

1003.0

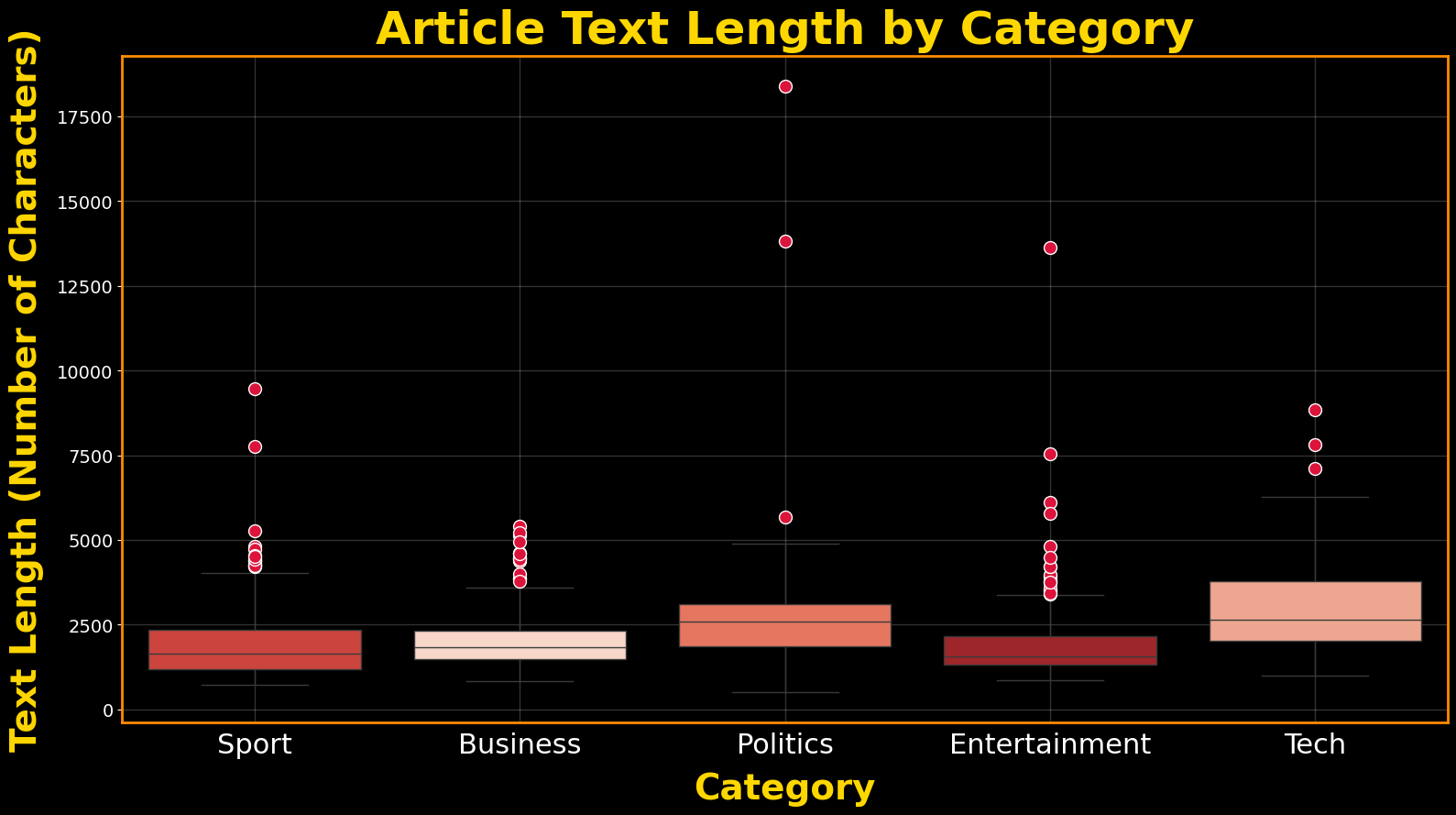
2031.00

2657.0

3775.00

8826.0

-------------------------------------------------------------------------------------



png

**Observation - Article Text Length by Category:**

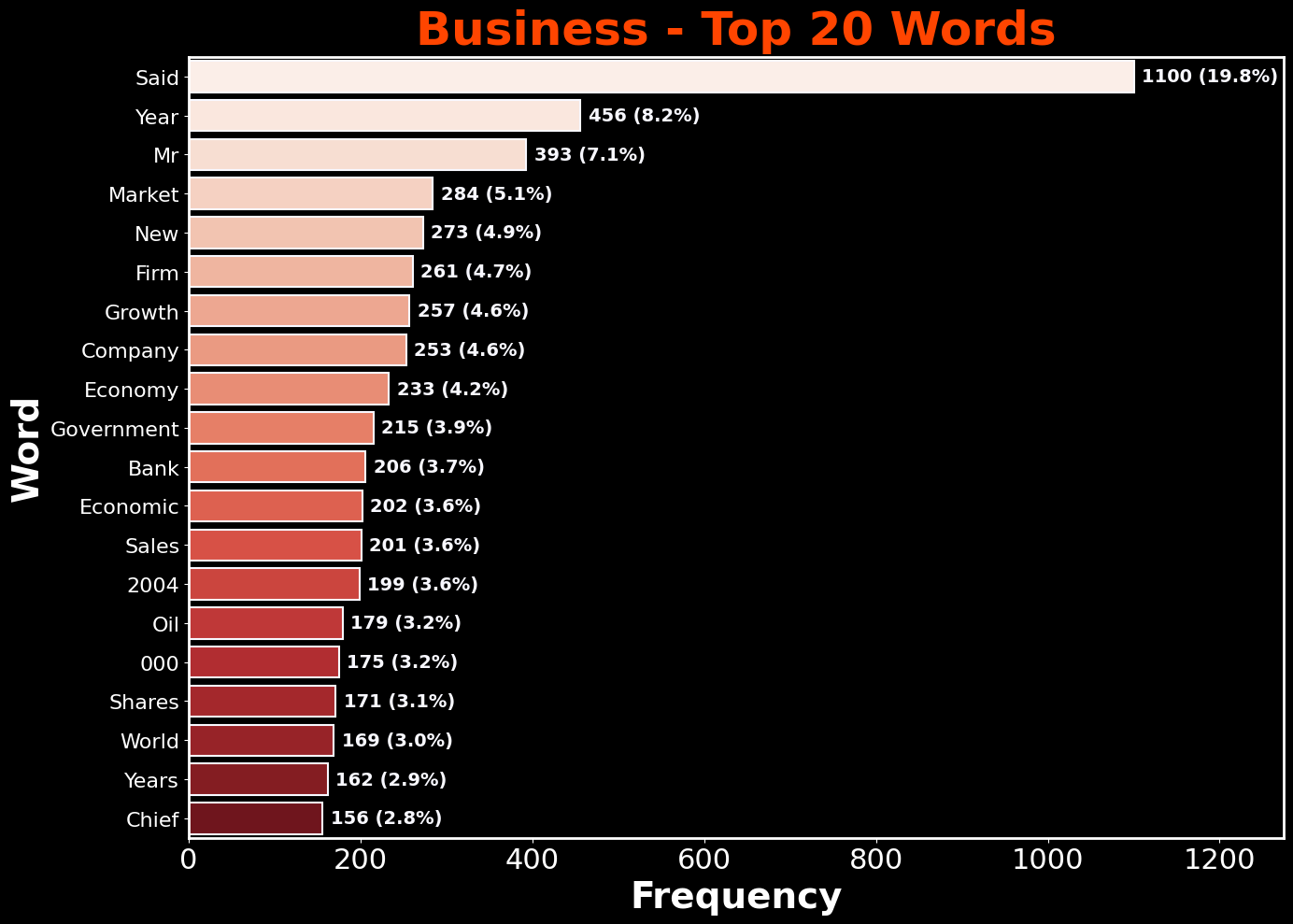
The descriptive statistics table and the box plot both tell a similar story. - **Politics** articles have the highest mean length and the largest standard deviation, meaning they are not only longer on average but also have the widest variety in length. - **Entertainment** articles appear to be the most consistent in length, with a tighter distribution.

This confirms that there are noticeable differences in the text length characteristics between categories, which could be a useful signal for a machine learning model.

4.5 Business Category - Top Words

In this section, we focus on the **Business** category.  
The plot below displays the top 20 most frequently used words in business-related news articles.  
This visualization helps us identify key topics and language patterns unique to this category.

# Now, let's analyze the most common words in a specific category.  
category = 'business'  
category\_texts = df\_train[df\_train['Category'] == category]['Text']  
  
# Define a function to create a CountVectorizer with specific parameters.  
# This function will be used to create a bag-of-words representation of the text data.  
def make\_count\_vec():  
 return CountVectorizer(  
 stop\_words='english',  
 max\_df=0.95,  
 min\_df=2  
 )  
  
# Create a CountVectorizer instance with the defined parameters.  
bag\_of\_words = vec.fit\_transform(category\_texts)  
sum\_words = bag\_of\_words.sum(axis=0)  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
words\_freq = sorted(words\_freq, key=lambda x: x[1], reverse=True)[:20]  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
top\_words\_df = pd.DataFrame(words\_freq, columns=['Word', 'Count'])  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
# This chart will show the most frequent words in the 'business' category.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in business-related articles.  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black')  
  
  
sns.barplot(  
 x='Count',   
 y='Word',   
 data=top\_words\_df,   
 hue='Word',  
 palette='Reds',  
 legend=False,  
 edgecolor='ghostwhite',  
 linewidth=1.5,  
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Business - Top 20 Words", fontsize=36, color='orangered', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='white', fontweight='bold')  
ax.set\_ylabel("Word", fontsize=28, color='white', fontweight='bold')  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('white')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 1275) # keeps labels clear of the top spine  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

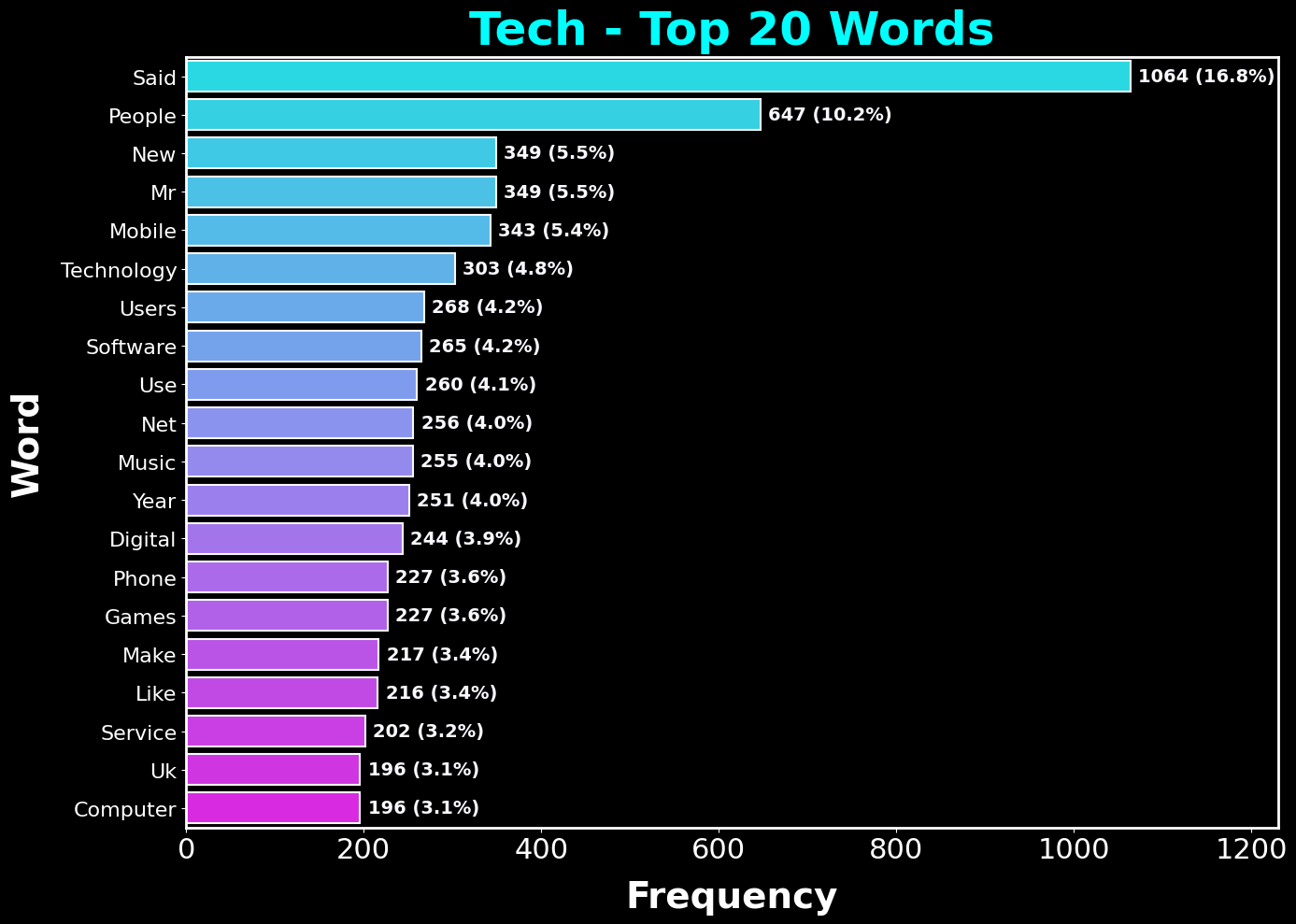
**Observation - Top 20 Most Frequent Words - Business:**

The top terms in the Business category reflect common themes in financial and economic reporting.  
We see frequent mentions of monetary figures, corporate terminology, and market-related language.  
This reinforces that the Business articles focus heavily on companies, industry sectors, and the broader economy.

4.6 Tech Category - Top Words

Here we explore the **Tech** category.  
By plotting the top 20 most common words, we can see the vocabulary that dominates technology news coverage and identify recurring themes in the dataset.

# Now, let's analyze the most common words in another specific category.  
# This time, we will focus on the 'tech' category.  
category = 'tech'  
category\_texts = df\_train[df\_train['Category'] == category]['Text']  
  
# Create a CountVectorizer instance with the defined parameters.  
# This will be used to create a bag-of-words representation of the text data.  
vec = CountVectorizer(stop\_words='english', ngram\_range=(1,1))  
bag\_of\_words = vec.fit\_transform(category\_texts)  
sum\_words = bag\_of\_words.sum(axis=0)  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
words\_freq = sorted(words\_freq, key=lambda x: x[1], reverse=True)[:20]  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common  
top\_words\_df = pd.DataFrame(words\_freq, columns=['Word', 'Count'])  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
# This chart will show the most frequent words in the 'tech' category.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in tech-related articles.  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black') # larger format  
  
# Use seaborn to create a bar plot of the top words.  
sns.barplot(  
 x='Count',   
 y='Word',   
 data=top\_words\_df,   
 hue='Word',  
 palette='cool',  
 legend=False,  
 edgecolor='ghostwhite',  
 linewidth=1.5,  
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Tech - Top 20 Words", fontsize=36, color='cyan', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='white', fontweight='bold', labelpad=12)  
ax.set\_ylabel("Word", fontsize=28, color='white', fontweight='bold', labelpad=12)  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('white')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 1230) # keeps labels clear of the top spine  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

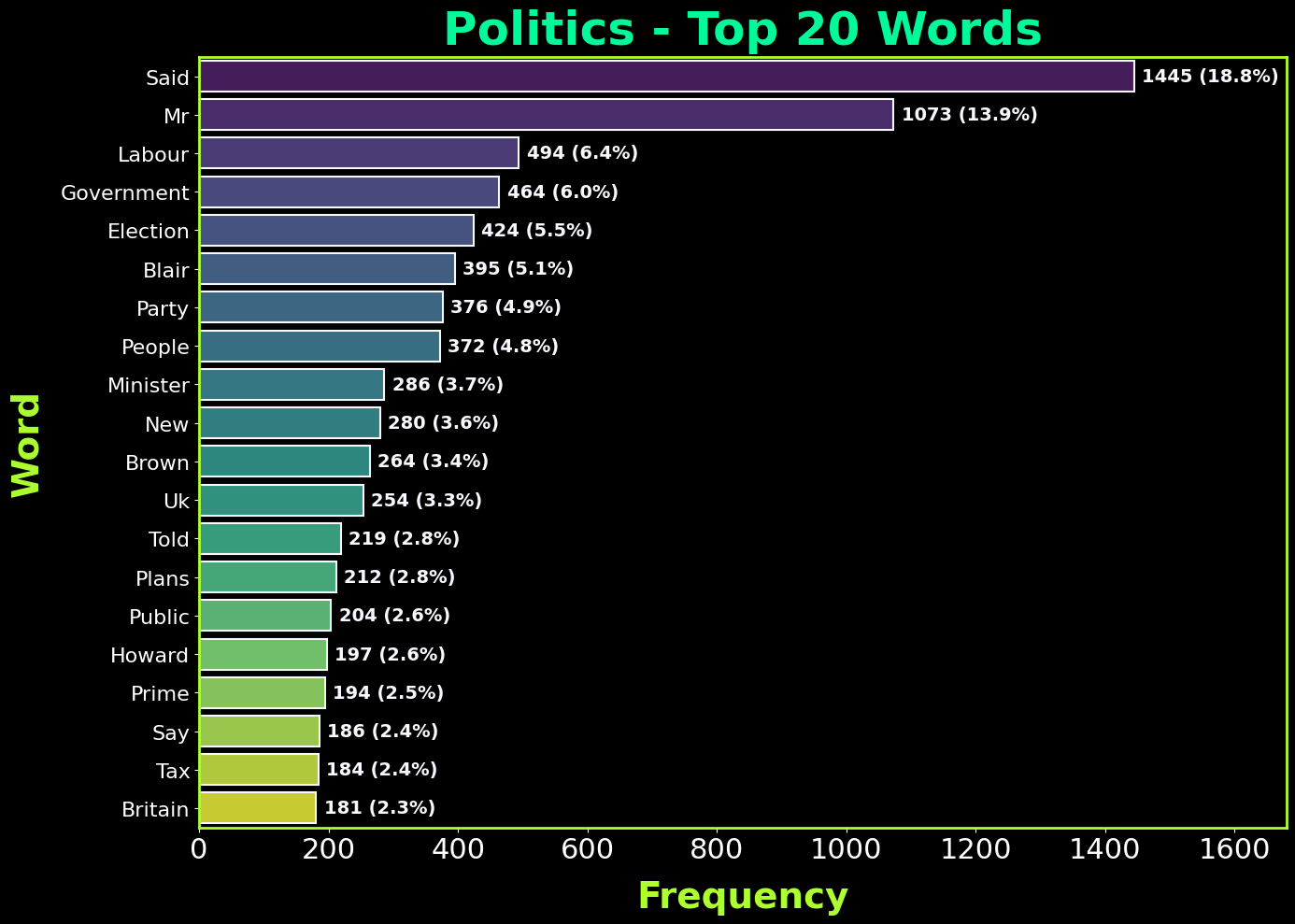
**Observation - Top 20 Most Frequent Words - Tech:**

In the Tech category, many high-frequency words relate to products, innovation, and technology companies.  
Terms associated with the internet, devices, and software appear prominently, suggesting a focus on product launches, digital trends, and corporate activity in the tech sector.

4.7 Politics Category - Top Words

This section examines the **Politics** category.  
The top words highlight the most frequently discussed topics, names, and political terms within the dataset’s political news articles.

# Now, let's analyze the most common words in another specific category.  
# This time, we will focus on the 'politics' category.  
# We will use the same approach as before, but with the 'politics' category.  
category = 'politics'  
category\_texts = df\_train[df\_train['Category'] == category]['Text']  
  
# Create a CountVectorizer instance with the defined parameters.  
# This will be used to create a bag-of-words representation of the text data.  
vec = CountVectorizer(stop\_words='english', ngram\_range=(1,1))  
bag\_of\_words = vec.fit\_transform(category\_texts)  
sum\_words = bag\_of\_words.sum(axis=0)  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
words\_freq = sorted(words\_freq, key=lambda x: x[1], reverse=True)[:20]  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in politics-related articles.  
top\_words\_df = pd.DataFrame(words\_freq, columns=['Word', 'Count'])  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
# This chart will show the most frequent words in the 'politics' category.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black') # larger format  
  
# Use seaborn to create a bar plot of the top words.  
sns.barplot(  
 x='Count',   
 y='Word',   
 data=top\_words\_df,   
 hue='Word',  
 palette='viridis',  
 legend=False,  
 edgecolor='ghostwhite',  
 linewidth=1.5,  
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Politics - Top 20 Words", fontsize=36, color='mediumspringgreen', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='greenyellow', fontweight='bold', labelpad=12)  
ax.set\_ylabel("Word", fontsize=28, color='greenyellow', fontweight='bold', labelpad=12)  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('greenyellow')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 1680) # keeps labels clear of the top spine  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.   
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

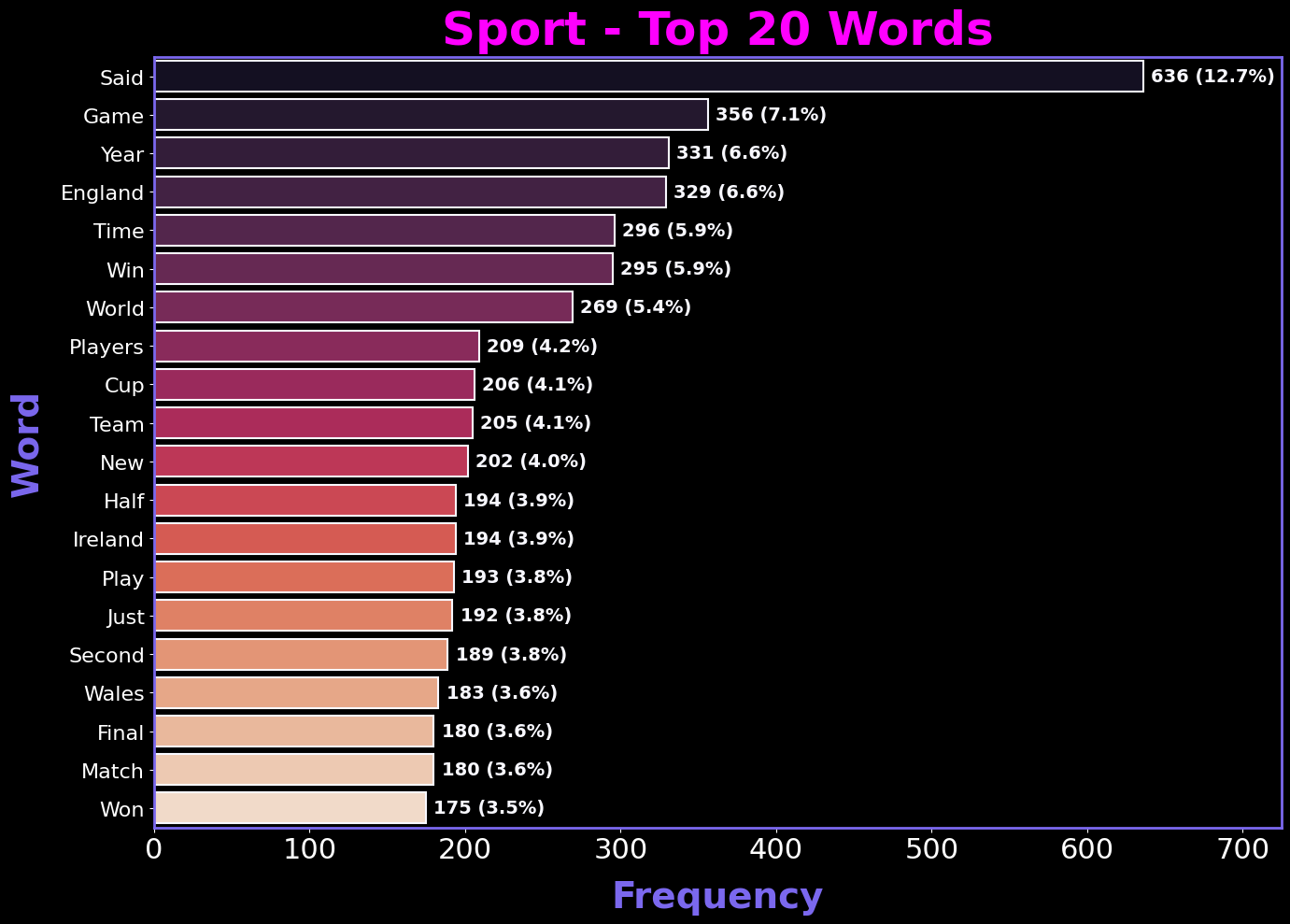
**Observation - Top 20 Most Frequent Words - Politics:**

The Politics category is dominated by names of political figures, government institutions, and policy-related terms.  
This points to coverage focused on government decisions, party politics, and legislative developments.  
Geographic and country references also suggest that much of the reporting centers around political events in specific regions.

4.8 Sport Category - Top Words

In the **Sport** category, frequent terms often reflect competitions, teams, players, and events.  
The bar chart below shows the top 20 most common words, giving us insight into the language of sports reporting.

# Now, let's analyze the most common words in another specific category.  
# This time, we will focus on the 'sport' category.  
# We will use the same approach as before, but with the 'sport' category.  
category = 'sport'  
category\_texts = df\_train[df\_train['Category'] == category]['Text']  
  
# Create a CountVectorizer instance with the defined parameters.  
# This will be used to create a bag-of-words representation of the text data.  
vec = CountVectorizer(stop\_words='english', ngram\_range=(1,1))  
bag\_of\_words = vec.fit\_transform(category\_texts)  
sum\_words = bag\_of\_words.sum(axis=0)  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
words\_freq = sorted(words\_freq, key=lambda x: x[1], reverse=True)[:20]  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in sport-related articles.  
top\_words\_df = pd.DataFrame(words\_freq, columns=['Word', 'Count'])  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
# This chart will show the most frequent words in the 'sport' category.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in sport-related articles.  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black') # larger format  
  
# Use seaborn to create a bar plot of the top words.  
sns.barplot(  
 x='Count',   
 y='Word',   
 data=top\_words\_df,   
 hue='Word',  
 palette='rocket',  
 legend=False,  
 edgecolor='ghostwhite',  
 linewidth=1.5,  
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Sport - Top 20 Words", fontsize=36, color='magenta', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='mediumslateblue', fontweight='bold', labelpad=12)  
ax.set\_ylabel("Word", fontsize=28, color='mediumslateblue', fontweight='bold', labelpad=12)  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('mediumslateblue')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 725) # keeps labels clear of the top spine  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

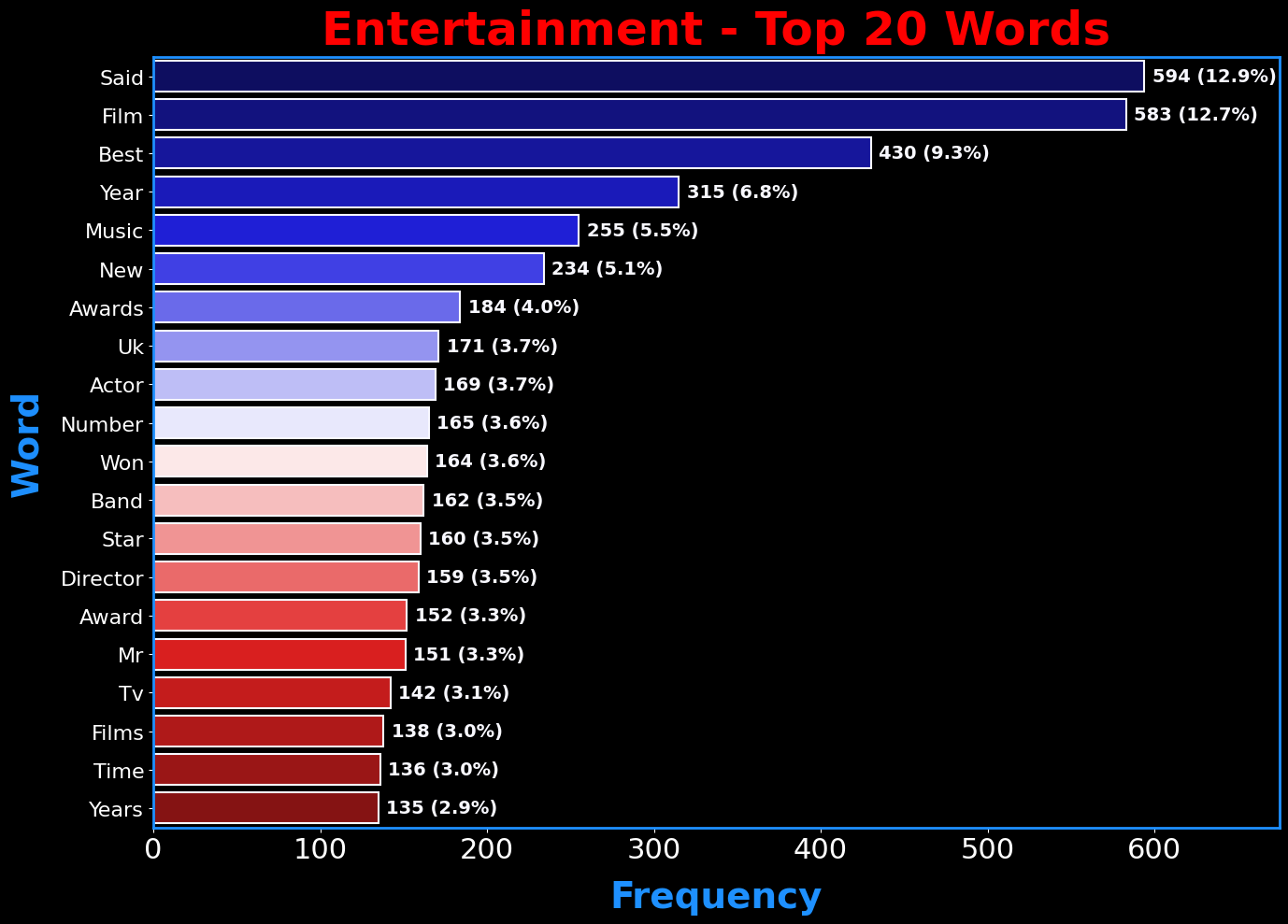
**Observation - Top 20 Most Frequent Words - Sport:**

Sports coverage shows frequent use of team names, player names, and competition-related terms.  
The vocabulary reflects ongoing tournaments, matches, and league events.  
The prevalence of score-related or result-oriented words also indicates the real-time nature of sports reporting.

4.9 Entertainment Category - Top Words

Here we look at the **Entertainment** category.  
From celebrity names to film, music, and cultural terms, the top words chart reveals what dominates the entertainment news landscape in this dataset.

# Now, let's analyze the most common words in another specific category.  
# This time, we will focus on the 'entertainment' category.  
# We will use the same approach as before, but with the 'entertainment' category  
category = 'entertainment'  
category\_texts = df\_train[df\_train['Category'] == category]['Text']  
  
# Create a CountVectorizer instance with the defined parameters.  
# This will be used to create a bag-of-words representation of the text data.   
vec = CountVectorizer(stop\_words='english', ngram\_range=(1,1))  
bag\_of\_words = vec.fit\_transform(category\_texts)  
sum\_words = bag\_of\_words.sum(axis=0)  
words\_freq = [(word, sum\_words[0, idx]) for word, idx in vec.vocabulary\_.items()]  
words\_freq = sorted(words\_freq, key=lambda x: x[1], reverse=True)[:20]  
  
# Create a DataFrame from the top 20 most frequent words.  
# This DataFrame will be used to create a horizontal bar chart.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in entertainment-related articles.  
top\_words\_df = pd.DataFrame(words\_freq, columns=['Word', 'Count'])  
top\_words\_df['Word'] = top\_words\_df['Word'].str.capitalize()  
  
# Create a horizontal bar chart to display these top words.  
# This chart will show the most frequent words in the 'entertainment' category.  
# The words are sorted by frequency, and the chart will help visualize which terms are most common in entertainment-related articles.  
fig, ax = plt.subplots(figsize=(14, 10), facecolor='black') # larger format  
  
# Use seaborn to create a bar plot of the top words.  
sns.barplot(  
 x='Count',   
 y='Word',   
 data=top\_words\_df,   
 hue='Word',  
 palette='seismic',  
 legend=False,  
 edgecolor='ghostwhite',  
 linewidth=1.5,  
 ax=ax  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Entertainment - Top 20 Words", fontsize=36, color='red', fontweight='bold', pad=10)  
ax.set\_xlabel("Frequency", fontsize=28, color='dodgerblue', fontweight='bold', labelpad=12)  
ax.set\_ylabel("Word", fontsize=28, color='dodgerblue', fontweight='bold', labelpad=12)  
ax.tick\_params(axis='x', labelsize=22, colors='white')  
ax.tick\_params(axis='y', labelsize=16, colors='white')  
  
# Set the background color of the plotting area.  
ax.set\_facecolor('black')  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('dodgerblue')  
 spine.set\_linewidth(2)  
  
ax.set\_xlim(0, 675) # keeps labels clear of the top spine  
  
bars = ax.patches # rectangles drawn by seaborn  
total = top\_words\_df["Count"].sum() # percent of the shown top-20 words  
  
# Annotate each bar with its count and percentage.  
# This adds text labels to the right of each bar to show the exact count and percentage.  
# The percentage is calculated as (count / total) \* 100.  
# The text is formatted to show both the count and percentage with one decimal place.  
for bar in bars:  
 w = bar.get\_width()  
 y = bar.get\_y() + bar.get\_height() / 2  
 ax.annotate(f"{int(w)} ({(w/total)\*100:.1f}%)",  
 (w, y),  
 xytext=(6, 0), textcoords="offset points",  
 va="center", ha="left",  
 fontsize=14, color="ghostwhite", fontweight="bold")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation - Top 20 Most Frequent Words - Entertainment:**

The Entertainment category contains frequent references to celebrities, media titles, and cultural events.  
The vocabulary suggests a strong focus on film, television, and music, along with award shows and public appearances.  
This aligns with the expectation that Entertainment reporting blends industry updates with personality-driven stories.

4.10a Combined Word Clouds by Category (Non-Filtered)

I generate a word cloud for each category using raw text. This shows the dominant tokens before any custom filtering.

# Define a dictionary to hold the color maps and title colors for each category.  
# This will be used to generate word clouds for each category with specific color schemes.  
category\_colors = {  
 'business': ('Reds', 'orangered'),  
 'tech': ('cool', 'springgreen'),  
 'politics': ('viridis', 'lime'),  
 'sport': ('rocket', 'cyan'),  
 'entertainment': ('magma', 'violet')  
}  
  
# Import necessary libraries for word cloud generation and visualization.  
fig, axes = plt.subplots(3, 2, figsize=(24, 18), facecolor='black')  
fig.patch.set\_facecolor('black')  
ax\_flat = axes.flatten()  
  
# Loop through categories and generate word clouds (no custom stopwords)  
for i, (category, (cmap, title\_color)) in enumerate(category\_colors.items()):  
 ax = ax\_flat[i]  
 category\_text = ' '.join(df\_train.loc[df\_train['Category'] == category, 'Text'].astype(str))  
  
 # Generate the word cloud for the current category.  
 # The WordCloud object is created with a black background and a specific colormap.  
 wc\_raw = WordCloud(  
 background\_color='black',  
 max\_words=100,  
 width=800,  
 height=400,  
 colormap=cmap  
 ).generate(category\_text)  
  
 # Display the generated word cloud image.  
 # The word cloud will show the most frequent words in the specified category, with a black background and the defined color map.  
 ax.imshow(wc\_raw, interpolation='bilinear')  
 ax.set\_title(f"{category.capitalize()}", fontsize=28, color=title\_color, fontweight='bold', pad=12)  
 ax.axis('off')  
  
# Hide the unused 6th subplot  
ax\_flat[-1].axis('off')  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()

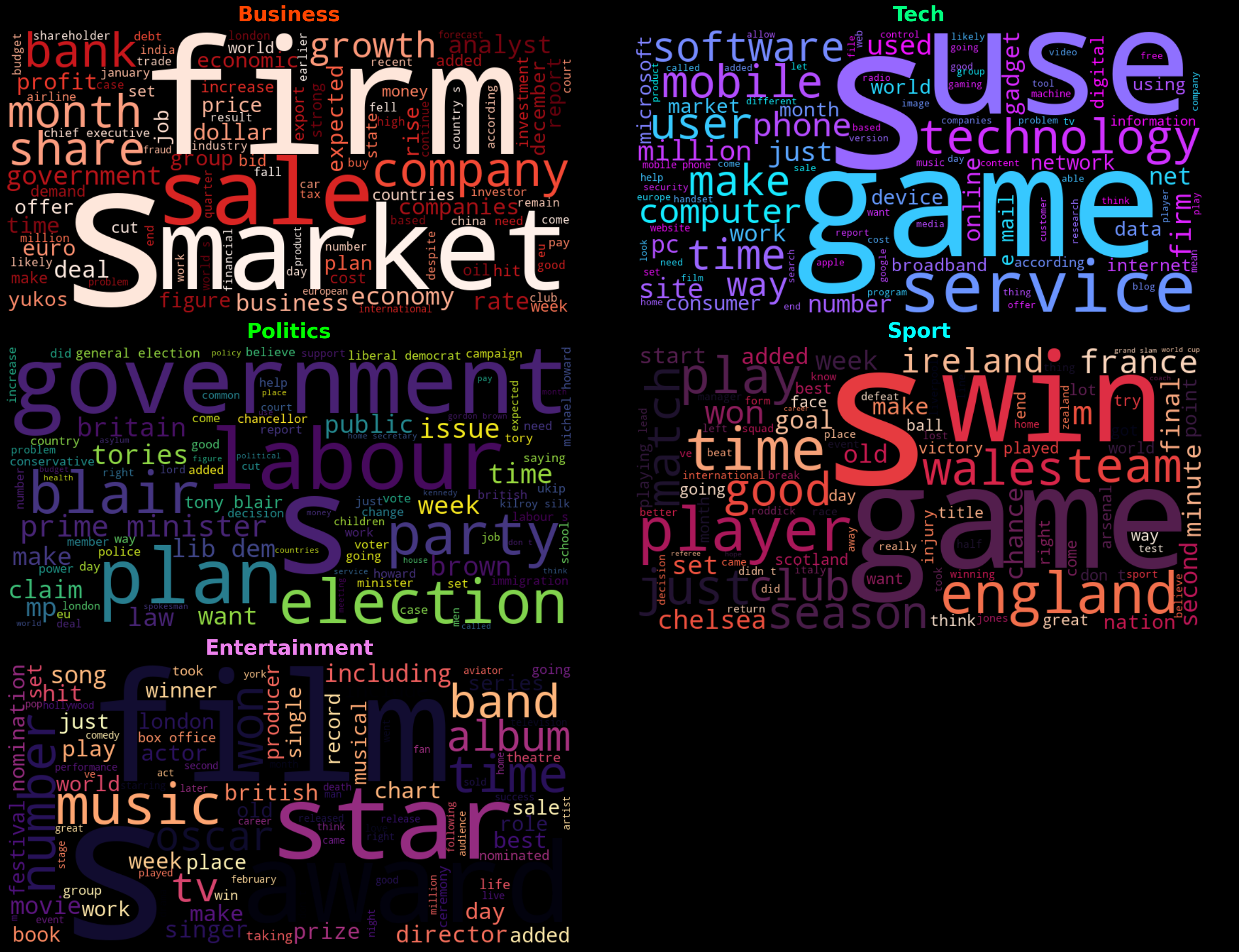


png

4.10b Word Clouds by News Category (Filtered Stopwords)

In this version, common and filler terms are removed using the custom\_stop list (e.g., “said,” “mr,” “year,” “people,” “uk,” “bbc”). This focuses the visualizations on more topical and category-specific words, which can make trends easier to spot compared to the unfiltered version in 4.10a.

# Define custom stopwords for the Word Cloud.  
# These are common words that do not add significant meaning to the text.  
# They are often removed to focus on more meaningful words in the visualization.  
news\_stop = {"said","say","says","mr","mrs","ms","one","two","new","year","years","people","told","also","could","would","well","like","get","back","u","uk","bbc","000"}  
custom\_stop = text.ENGLISH\_STOP\_WORDS.union(news\_stop)  
SEED = 42  
  
# Define a dictionary to hold the color maps and title colors for each category.  
# This will be used to generate word clouds for each category with specific color schemes.  
category\_colors = {  
 'business': ('Reds', 'orangered'),  
 'tech': ('cool', 'springgreen'),  
 'politics': ('viridis', 'lime'),  
 'sport': ('rocket', 'cyan'),  
 'entertainment': ('magma', 'violet')  
}  
  
# Import necessary libraries for word cloud generation and visualization.  
fig, axes = plt.subplots(3, 2, figsize=(24, 18), facecolor='black')  
fig.patch.set\_facecolor('black')  
ax\_flat = axes.flatten()  
  
# Loop through categories and generate word clouds (with custom stopwords)  
# This will create a word cloud for each category, filtering out common words using the custom stopwords.  
# Each word cloud will be displayed in a subplot with a specific color map and title color.  
# The word clouds will help visualize the most frequent and relevant words in each category.  
# The custom stopwords help filter out common words that do not contribute much to the meaning of the text.  
# The Word Cloud will highlight the most relevant terms in the dataset.  
for i, (category, (cmap, title\_color)) in enumerate(category\_colors.items()):  
 ax = ax\_flat[i]  
 category\_text = ' '.join(df\_train.loc[df\_train['Category'] == category, 'Text'].astype(str))  
  
 # Note: stopwords=custom\_stop removes common/filler terms, shifting emphasis to topical words.  
 wc\_filtered = WordCloud(  
 background\_color='black',  
 max\_words=100,  
 width=800,  
 height=400,  
 colormap=cmap,  
 stopwords=custom\_stop,  
 random\_state=SEED  
 ).generate(category\_text)  
  
 # Display the generated word cloud image.  
 # The word cloud will show the most frequent words in the specified category, with a black background and the defined color map.  
 # The custom stopwords help filter out common words that do not contribute much to the meaning of the text.  
 # The Word Cloud will highlight the most relevant terms in the dataset.  
 # The colormap provides a visually appealing contrast between the most and least frequent words.  
 ax.imshow(wc\_filtered, interpolation='bilinear')  
 ax.set\_title(f"{category.capitalize()}", fontsize=28, color=title\_color, fontweight='bold', pad=12)  
 ax.axis('off')  
  
# Hide the unused 6th subplot  
ax\_flat[-1].axis('off')  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observations – Category-Specific Word Clouds (4.10a Unfiltered vs 4.10b Filtered)**

The unfiltered category word clouds (4.10a) are dominated by common journalistic filler terms such as *said*, *will*, *year*, and *people*. While these high-frequency terms reflect the linguistic style of news reporting, they dilute category-specific vocabulary, making it harder to distinguish unique thematic elements.

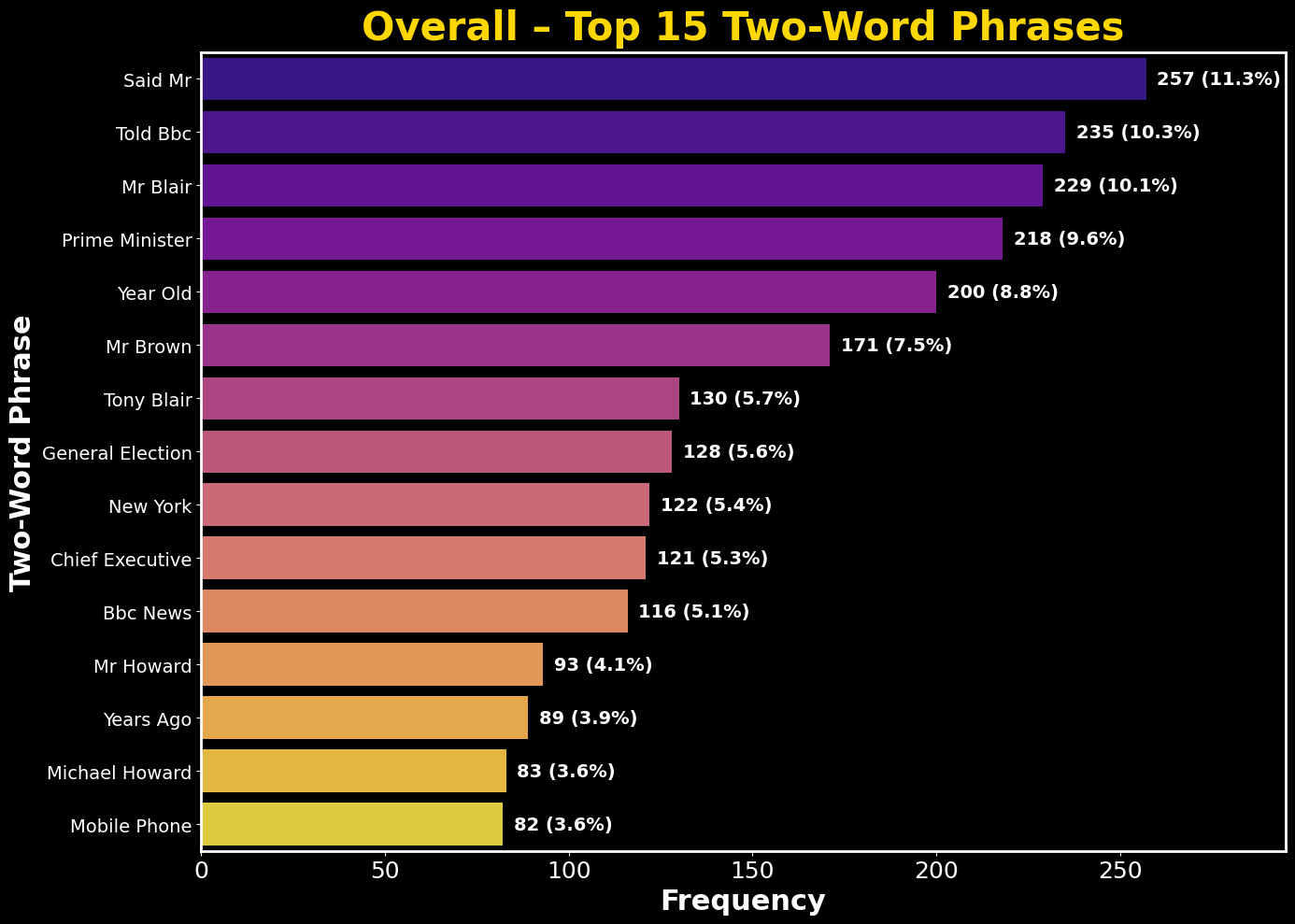
The filtered version (4.10b), which removes these common terms, shifts focus toward more topical and domain-specific words for each category.  
- **Business** emphasizes *market*, *firm*, *sale*, and *bank*.  
- **Tech** highlights *software*, *mobile*, *user*, and *technology*.  
- **Politics** centers on *government*, *election*, *plan*, and *blair*.  
- **Sport** brings forward *win*, *team*, *game*, and *england*.  
- **Entertainment** emphasizes *film*, *music*, *album*, and *star*.

This filtering makes each category’s vocabulary more distinct and interpretable, at the cost of removing some context words that may still hold narrative value.

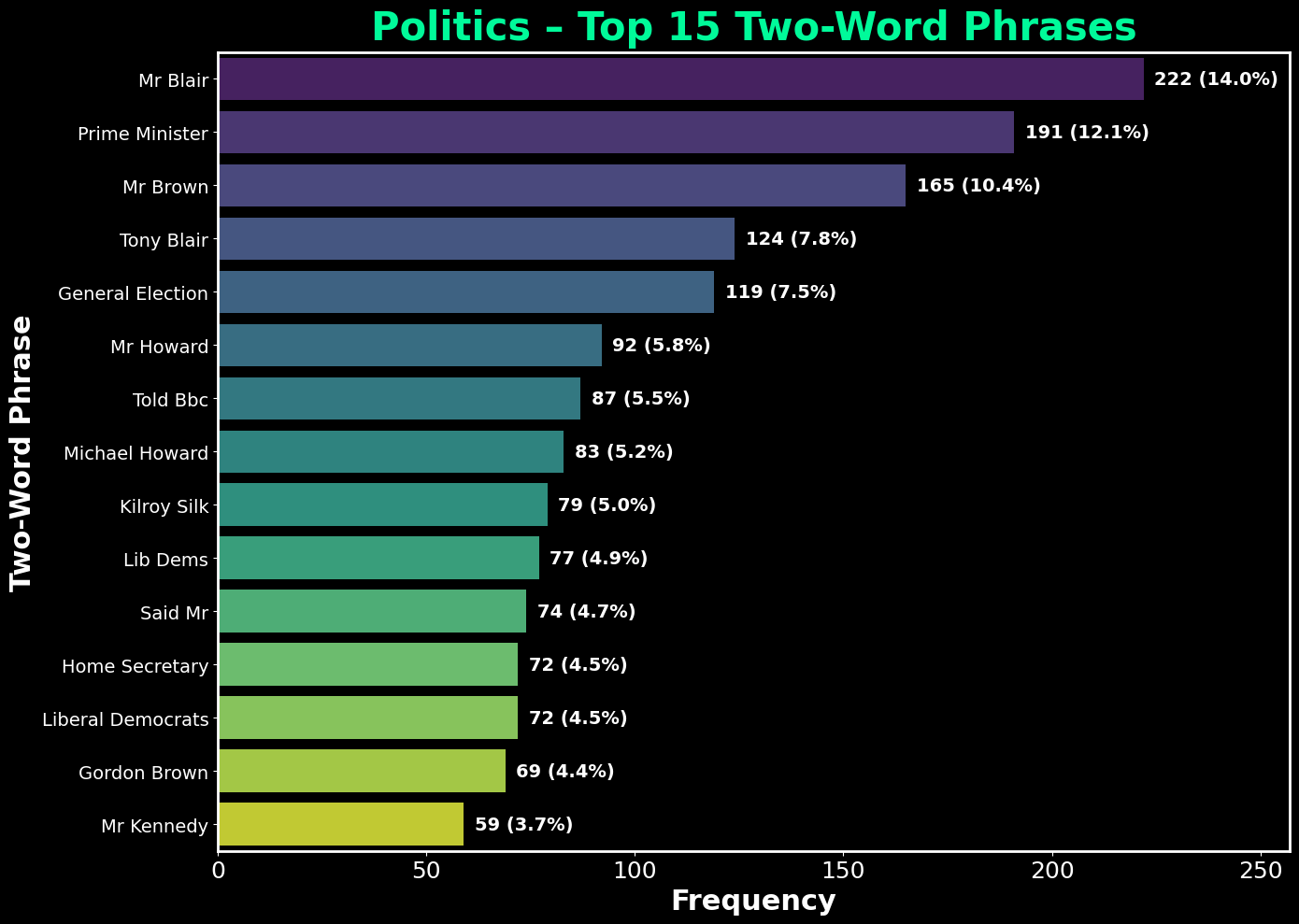
4.11 Category-Specific Top Bigrams

Single words are useful, but short phrases often carry more signal (e.g., prime minister, mobile phone, world cup). I extract the most frequent bigrams within a category to surface these patterns before modeling.

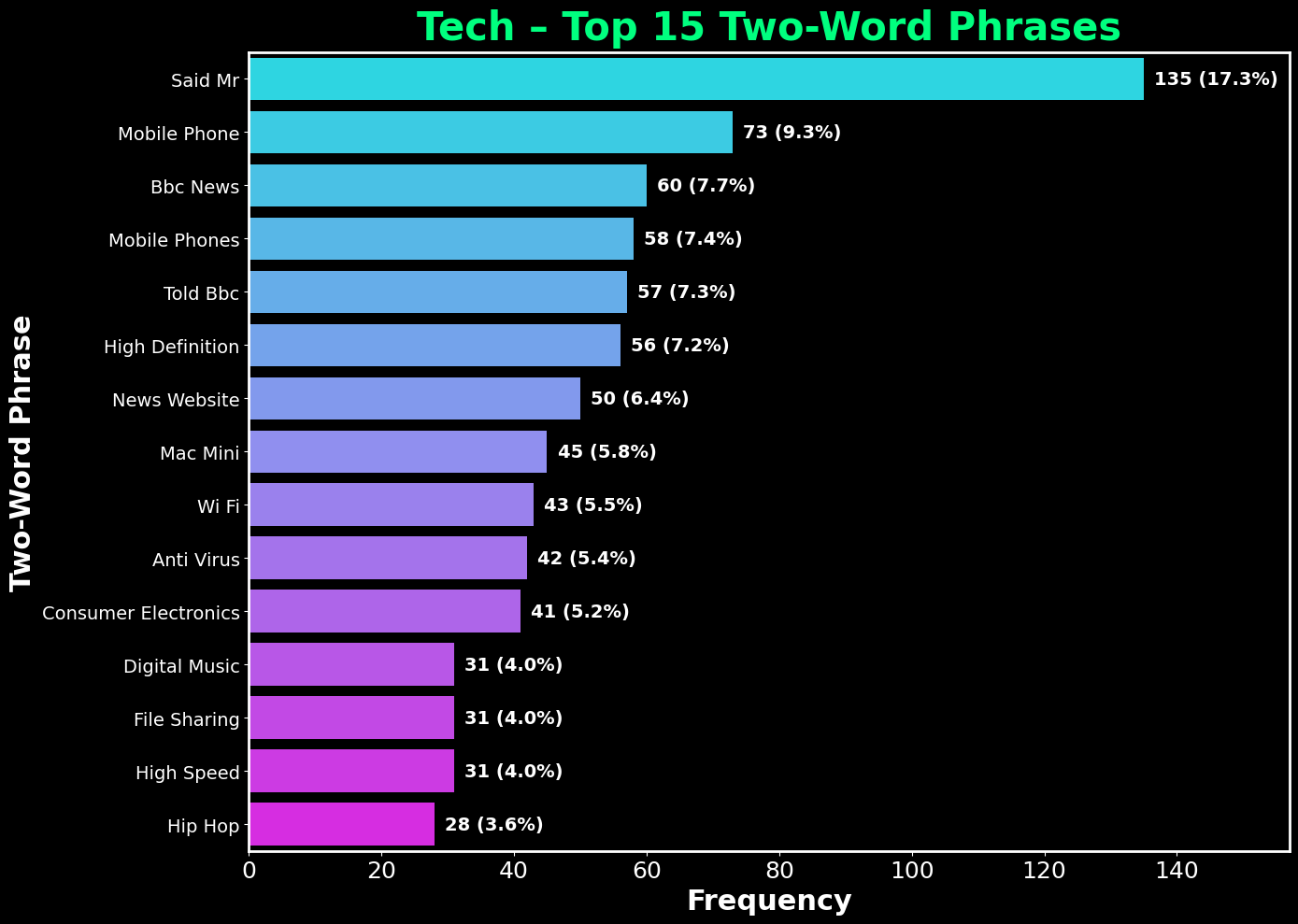
# Function to get top n-grams for a given category  
# This function takes a DataFrame, a category, and parameters for n-grams and topn results.  
# It returns a DataFrame containing the top n-grams and their counts for the specified category.  
# If the category is "overall", it considers all texts in the DataFrame.  
# The n-grams are generated using CountVectorizer, which tokenizes the text and counts the occurrences of each n-gram.  
# The results are sorted by count, and the top n-grams are returned in a DataFrame.  
# The phrases are capitalized for better readability.  
def top\_ngrams(df, category, n=2, topn=15):  
 """Return top n-grams for a given category."""  
 if category.lower() != "overall":  
 texts = df.loc[df["Category"] == category, "Text"].astype(str)  
 else:  
 texts = df["Text"].astype(str)  
   
 # Create a CountVectorizer to extract n-grams  
 vec = CountVectorizer(  
 stop\_words="english",  
 ngram\_range=(n, n),  
 token\_pattern=r"(?u)\b\w\w+\b"  
 )  
  
 # Fit and transform the texts to get the n-grams  
 X = vec.fit\_transform(texts)  
 sums = np.asarray(X.sum(axis=0)).ravel()  
 vocab = np.array(vec.get\_feature\_names\_out())  
 idx = sums.argsort()[::-1][:topn]  
   
 # Create a DataFrame with the top n-grams and their counts  
 tbl = pd.DataFrame({"Phrase": vocab[idx], "Count": sums[idx]})  
 tbl["Phrase"] = tbl["Phrase"].str.title() # Capitalize each word in the phrase  
  
 # Sort the DataFrame by count in descending order   
 return tbl  
  
# Uncomment the following lines to display top n-grams for each category in the training set.  
# display(top\_ngrams(df\_train, "overall", n=2, topn=15))  
# display(top\_ngrams(df\_train, "politics", n=2, topn=15))  
# display(top\_ngrams(df\_train, "tech", n=2, topn=15))  
# display(top\_ngrams(df\_train, "sport", n=2, topn=15))  
# display(top\_ngrams(df\_train, "business", n=2, topn=15))  
# display(top\_ngrams(df\_train, "entertainment", n=2, topn=15))  
  
  
# Function to plot top n-grams with counts and percentages  
# This function takes a DataFrame, a category, and parameters for n-grams and topn results.  
# It generates a bar plot showing the top n-grams along with their counts and percentages.  
# The plot includes annotations for each bar, displaying the count and percentage of each n-gram.  
# The x-axis represents the frequency of the n-grams, while the y-axis shows the n-grams themselves.  
def plot\_top\_ngrams(df, category, n=2, topn=15, palette="viridis", title\_color="deepskyblue", xlim=500):  
 """Plot top n-grams with counts and percentages."""  
 tbl = top\_ngrams(df, category, n=n, topn=topn)  
 total\_count = tbl["Count"].sum()  
   
 # Set up the plot  
 fig, ax = plt.subplots(figsize=(14, 10), facecolor="black")  
 sns.barplot(  
 x="Count", y="Phrase", data=tbl,  
 hue="Phrase", palette=palette, legend=False, ax=ax  
 )  
   
 # Annotate each bar with its count and percentage  
 # This adds text labels to the right of each bar to show the exact count and percentage  
 # The percentage is calculated as (count / total\_count) \* 100.  
 # The text is formatted to show both the count and percentage with one decimal place.  
 # The annotations are positioned slightly to the right of the bar for clarity.  
 for i, (count) in enumerate(tbl["Count"]):  
 pct = 100 \* count / total\_count  
 ax.text(  
 count + (xlim \* 0.01), i, f"{count} ({pct:.1f}%)",  
 va='center', fontsize=14, color="white", fontweight="bold"  
 )  
   
 # Set the title and labels with custom font properties.  
 ax.set\_title(  
 f"{category.capitalize() if category != 'overall' else 'Overall'} – Top {topn} Two-Word Phrases",  
 fontsize=30, color=title\_color, fontweight="bold", pad=10  
 )  
 ax.set\_xlabel("Frequency", fontsize=22, color="white", fontweight="bold")  
 ax.set\_ylabel("Two-Word Phrase", fontsize=22, color="white", fontweight="bold")  
 ax.tick\_params(axis="x", labelsize=18, colors="white")  
 ax.tick\_params(axis="y", labelsize=14, colors="white")  
 ax.set\_facecolor("black")  
 ax.set\_xlim(0, xlim)  
  
 # Style the plot's border (spines).  
 for s in ax.spines.values():  
 s.set\_edgecolor("white")  
 s.set\_linewidth(2)  
   
 # Ensure the layout is tight and clean.  
 plt.tight\_layout()  
 plt.show()  
# Plot top n-grams for each category in the training set.  
# Each plot will show the top 15 two-word phrases for the specified category, along with their counts and percentages.  
# Xlim is set to control the x-axis limit for better visibility of the bars.  
plot\_top\_ngrams(df\_train, "overall", n=2, topn=15, palette="plasma", title\_color="gold", xlim=295) # Overall   
plot\_top\_ngrams(df\_train, "politics", n=2, topn=15, palette="viridis", title\_color="mediumspringgreen", xlim=257) # Politics  
plot\_top\_ngrams(df\_train, "tech", n=2, topn=15, palette="cool", title\_color="springgreen", xlim=157) # Tech  
plot\_top\_ngrams(df\_train, "sport", n=2, topn=15, palette="rocket", title\_color="cyan", xlim=145) # Sport  
plot\_top\_ngrams(df\_train, "business", n=2, topn=15, palette="Reds", title\_color="orangered", xlim=93) # Business  
plot\_top\_ngrams(df\_train, "entertainment", n=2, topn=15, palette="magma", title\_color="violet", xlim=77) # Entertainment



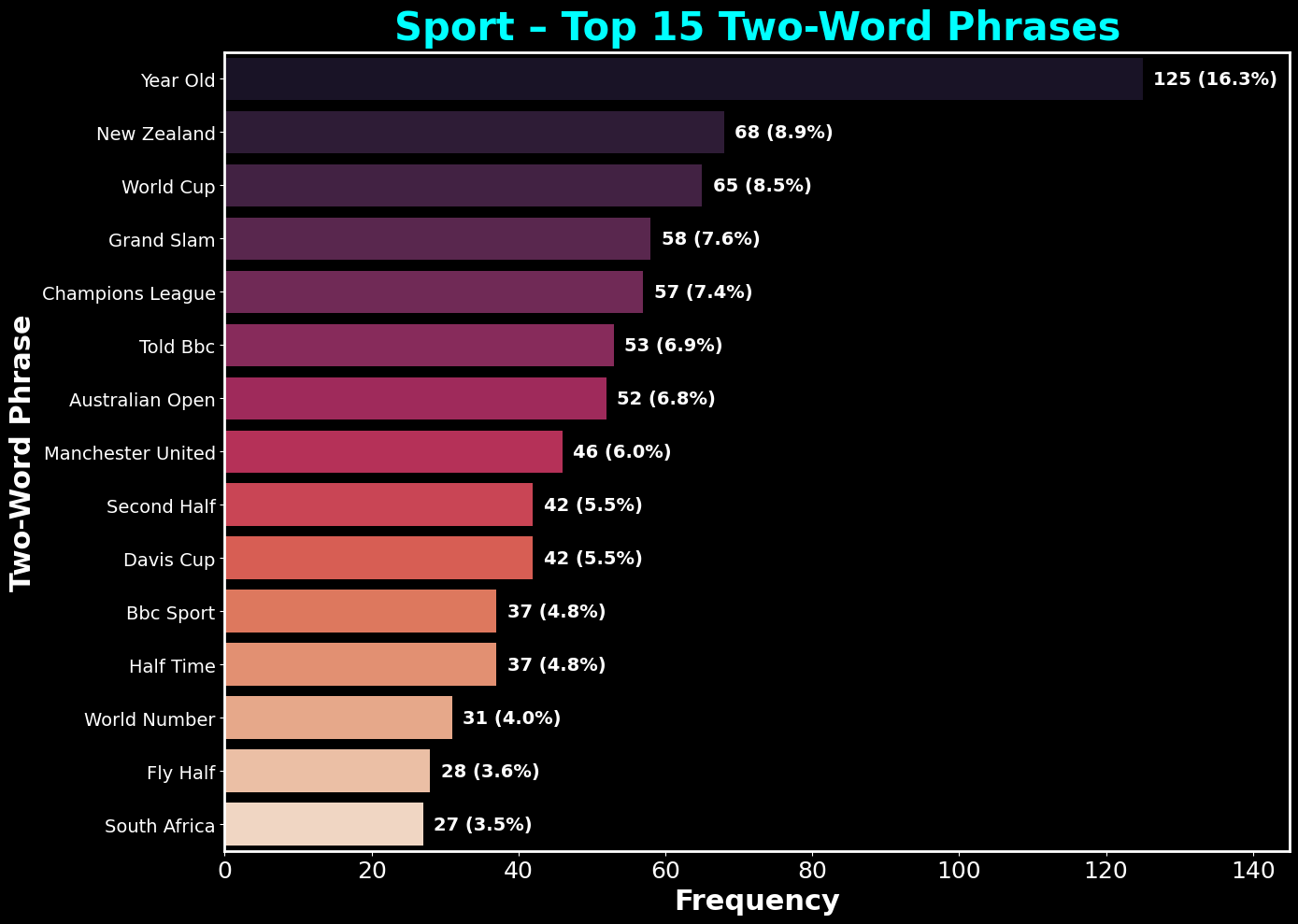
png



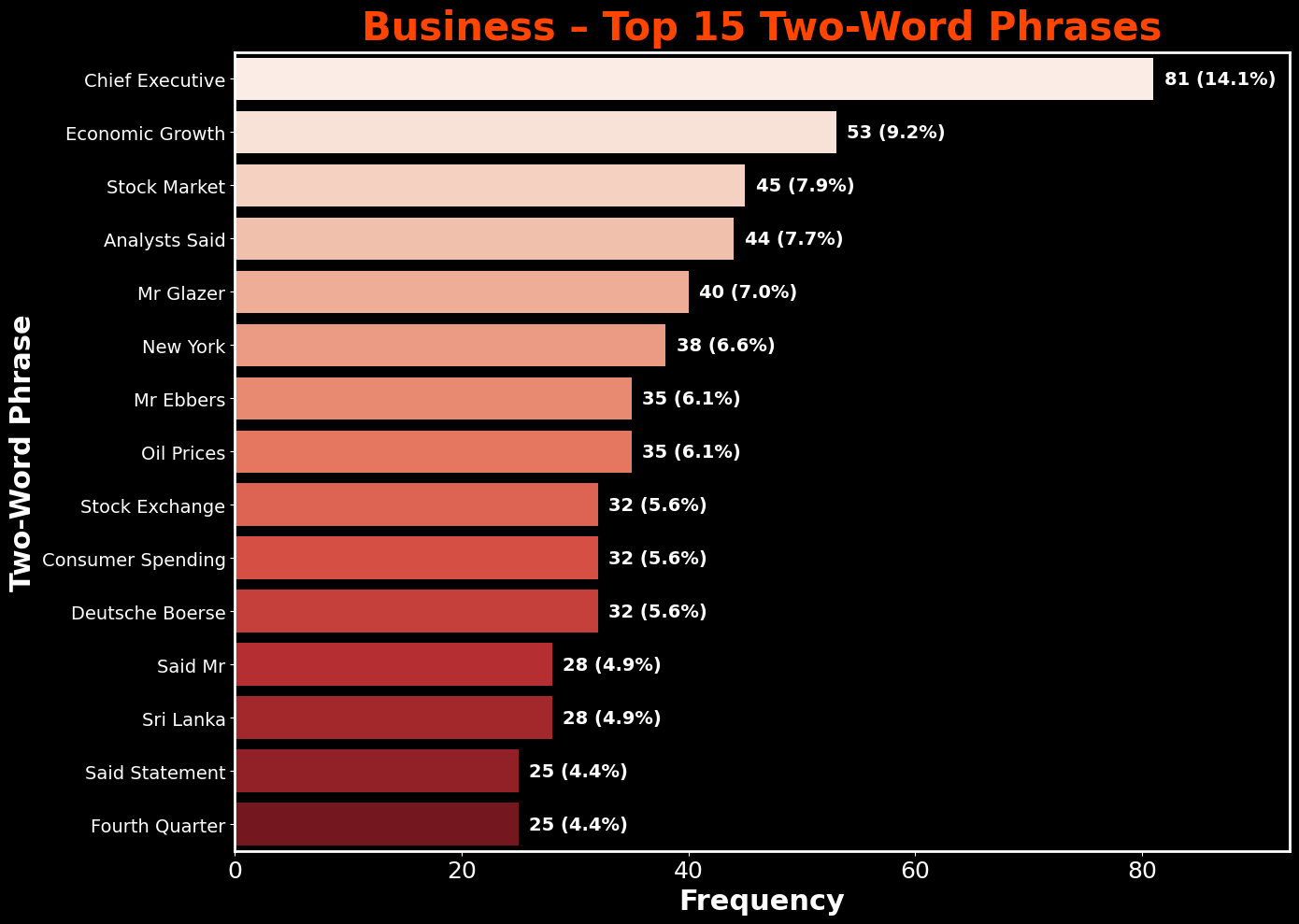
png



png



png



png



png

**Observations: Top 15 Two-Word Phrases by Category**

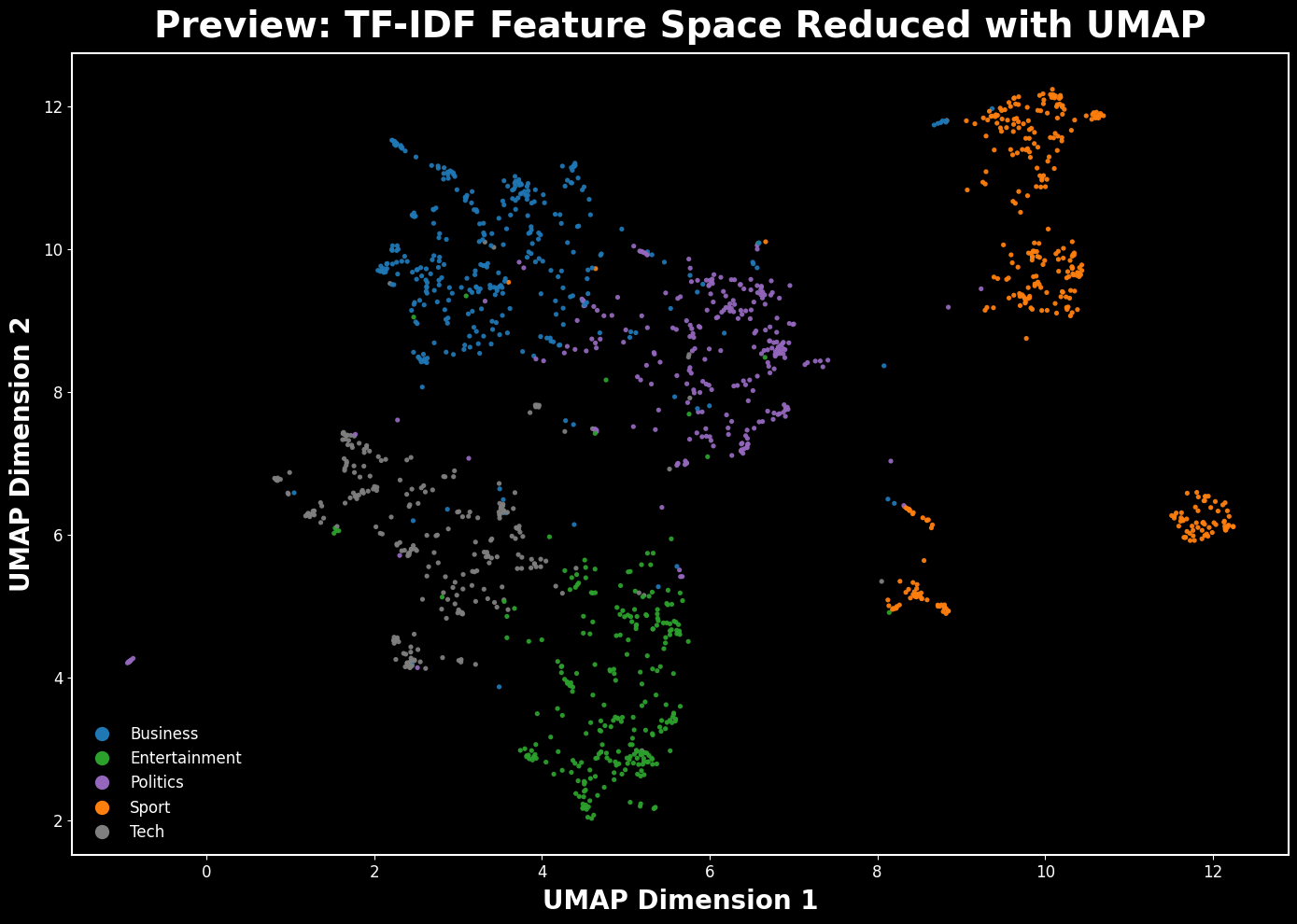
The analysis highlights both common language patterns across the dataset and topic-specific phrase usage.  
- **Overall**, recurring names and titles dominate, with *“Said Mr”*, *“Told Bbc”*, and political figures such as *“Mr Blair”* and *“Prime Minister”* holding the highest frequencies.  
- **Politics** is driven by government titles, party identifiers, and election terms, suggesting heavy coverage of political leadership and events.  
- **Tech** emphasizes product terms (*“Mobile Phone”*, *“Mac Mini”*), technology concepts (*“High Definition”*, *“File Sharing”*), and corporate references (*“Bbc News”*).  
- **Sport** focuses on event names (*“World Cup”*, *“Grand Slam”*, *“Champions League”*), international teams, and competition stages (*“Second Half”*, *“Davis Cup”*).  
- **Business** contains economic and market terminology (*“Economic Growth”*, *“Stock Market”*, *“Oil Prices”*) alongside leadership references (*“Chief Executive”*).  
- **Entertainment** phrases are award- and film-centric (*“Best Film”*, *“Box Office”*, *“Film Festival”*), reflecting coverage of the film industry and events.

The overlap of certain entities across categories (*“New York”*, *“Year Old”*, *“Bbc News”*) indicates cross-domain reporting, while other phrases remain highly domain-specific.

4.12 Document Similarity Map (TF-IDF → UMAP)

To close the EDA, I project articles into a 2D space using TF-IDF features reduced with UMAP. Each point is one article; points that sit closer together use similar vocabulary. The axes are abstract UMAP components (not physical units), but distances are meaningful for comparing documents.

import warnings  
  
# UMAP visualization of TF-IDF features  
# This section uses UMAP to reduce the dimensionality of TF-IDF features for visualization.  
# It creates a scatter plot of the reduced features, colored by category labels  
if 'custom\_stop' in globals():  
 \_stopwords = custom\_stop  
else:  
 from sklearn.feature\_extraction import text  
  
 # Define custom stopwords for the Word Cloud.  
 # These are common words that do not add significant meaning to the text.  
 # They are often removed to focus on more meaningful words in the visualization.  
 news\_stop = {  
 "said","say","says","mr","mrs","ms","one","two","new","year","years","people","told",  
 "also","could","would","well","like","get","back","u","uk","bbc","000"  
 }  
 \_stopwords = text.ENGLISH\_STOP\_WORDS.union(news\_stop)  
\_stopwords = list(\_stopwords)  
tfidf\_preview = TfidfVectorizer(  
 stop\_words=\_stopwords,  
 max\_features=20000,  
 token\_pattern=r"(?u)\b[a-z][a-z]+\b",  
 max\_df=0.95,  
 min\_df=2  
)  
  
# Fit the TF-IDF vectorizer to the training data  
# This will transform the text data into a TF-IDF feature matrix.  
X\_prev = tfidf\_preview.fit\_transform(df\_train["Text"].astype(str))  
  
# The UMAP warning is informational. This block will suppress it for a clean output.  
with warnings.catch\_warnings():  
 warnings.filterwarnings("ignore", category=UserWarning, module="umap")  
   
 # Attempt to import UMAP and fit the model  
 try:  
 emb = umap\_model.fit\_transform(X\_prev)  
 except NameError:  
 import umap as umap\_  
 umap\_model = umap\_.UMAP(n\_neighbors=15, min\_dist=0.1, random\_state=SEED)  
 emb = umap\_model.fit\_transform(X\_prev)  
  
# Convert categories to numerical labels for coloring the scatter plot  
# This will create a unique numerical label for each category in the training set.  
cats = df\_train["Category"].astype(str).values  
\_, labels = np.unique(cats, return\_inverse=True)  
  
# Set the style for the plot  
from matplotlib.colors import ListedColormap  
palette = ListedColormap(["#1f77b4", "#2ca02c", "#9467bd", "#ff7f0e", "#7f7f7f"])  
  
# Create a scatter plot of the UMAP-reduced features  
# This will visualize the TF-IDF feature space reduced to two dimensions using UMAP.  
# The points will be colored according to their category labels, providing a visual representation of the data  
fig, ax = plt.subplots(figsize=(14, 10), facecolor="black")  
ax.set\_facecolor("black")  
  
# Scatter plot of the UMAP embeddings  
# This will plot the UMAP-reduced features in a 2D space, with points colored by their category labels.  
# The scatter plot will help visualize the distribution of different categories in the TF-IDF feature space.  
sc = ax.scatter(emb[:, 0], emb[:, 1],  
 c=labels, s=14, cmap=palette, alpha=0.95, linewidths=0)  
  
# Set the title and labels with custom font properties.  
ax.set\_title("Preview: TF-IDF Feature Space Reduced with UMAP",  
 fontsize=28, color="white", fontweight="bold", pad=12)  
ax.set\_xlabel("UMAP Dimension 1", fontsize=20, color="white", fontweight="bold", labelpad=6)  
ax.set\_ylabel("UMAP Dimension 2", fontsize=20, color="white", fontweight="bold", labelpad=6)  
ax.tick\_params(colors="white", labelsize=12)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_color("white")  
 spine.set\_linewidth(1.5)  
  
# Add a legend to the plot  
handles = []  
names = np.unique(cats)  
for i, name in enumerate(names):  
 handles.append(plt.Line2D([], [], marker='o', linestyle='', color=palette(i), markersize=10, label=name.capitalize()))  
leg = ax.legend(handles=handles, frameon=False, fontsize=12, loc="lower left", labelcolor="white")  
for text in leg.get\_texts():  
 text.set\_color("white")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observations – TF-IDF → UMAP and Section 4 wrap-up**

The UMAP projection provides a final high-level view of the relationships between documents in the dataset.  
Each point represents a single BBC News article, positioned in a 2D space according to the similarity of its vocabulary (measured using TF-IDF). Articles that are closer together tend to share more similar word usage. Colors correspond to their assigned categories.

This plot does not preserve physical meaning in the X and Y axes — they are abstract coordinates from the UMAP algorithm — but it does preserve relative proximity, revealing clustering patterns across categories.

Taken together, the EDA in Section 4 has: - Identified the most common and distinctive words, phrases, and topics within each category. - Shown vocabulary overlap and separation between categories. - Illustrated category-level clusters when projecting the text into a reduced-dimensional space.

These insights guide the design of our feature extraction pipeline in Section 5, where TF-IDF will be used in a structured, model-ready form for text classification.

Section 5: Text Pre-processing with TF-IDF

Now that the EDA is complete and I understand the vocabulary used across categories, it’s time to prepare the text for modeling.  
Machine learning models can’t directly interpret raw text. They require numerical feature vectors.

To achieve this, I’ll use **TF-IDF (Term Frequency–Inverse Document Frequency)**.  
TF-IDF highlights words that are important to a specific document but less common across all documents, which helps identify terms that are most characteristic of each category.

I’ll configure the vectorizer to:  
- Remove common English stop words.  
- Ignore extremely frequent terms (appear in >95% of documents).  
- Ignore very rare terms (appear in fewer than 2 documents).

This ensures the resulting feature matrix is both informative and compact.

5.1 Converting Text to Numerical Features with TF-IDF

In this step, we transform the raw BBC news articles into a numerical format that can be used by machine learning models.  
We use **Term Frequency–Inverse Document Frequency (TF-IDF)** to assign a weight to each term based on how frequently it appears in a document relative to the entire corpus.  
The vectorizer is configured to: - Remove common English stop words  
- Keep only letter-based tokens with at least two characters  
- Ignore terms that appear in more than 95% of documents or in fewer than two documents  
- Limit the vocabulary to 20,000 terms for efficiency and reproducibility

The resulting TF-IDF matrix is **sparse**, meaning most entries are zero, as each document contains only a small fraction of the total vocabulary.  
We also preview the vocabulary and show the top weighted terms for the first training document.

# This section uses TF-IDF vectorization to convert text data into numerical feature vectors.  
# The TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer transforms the text data into a matrix of TF-IDF features.  
  
# Vectorization and data manipulation.  
tfidf\_vectorizer = TfidfVectorizer(  
 stop\_words="english", # drop common English stopwords  
 token\_pattern=r"(?u)\b[a-z][a-z]+\b", # keep words (letters only, len>=2)  
 max\_df=0.95, # ignore terms in >95% of docs  
 min\_df=2, # ignore terms appearing in <2 docs  
 max\_features=20000 # (optional) cap vocabulary for reproducibility/runtime  
)  
  
# Load the training and test datasets  
X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(df\_train["Text"].astype(str))  
X\_test\_tfidf = tfidf\_vectorizer.transform(df\_test["Text"].astype(str))  
  
# Display the shapes of the training and test TF-IDF matrices  
# This will show the number of documents and the number of features (terms) in each matrix  
print("-"\*70 + "\n")  
print(f"Shape of the training TF-IDF matrix: {X\_train\_tfidf.shape}")  
print("-"\*70 + "\n")  
print(f"Shape of the test TF-IDF matrix: {X\_test\_tfidf.shape}")  
print("-"\*70 + "\n")  
  
# Sparsity of the training TF-IDF matrix  
# This section calculates the sparsity of the training TF-IDF matrix.  
# Sparsity is defined as the ratio of non-zero elements to the total number of elements in the matrix.  
nnz = X\_train\_tfidf.nnz  
total = X\_train\_tfidf.shape[0] \* X\_train\_tfidf.shape[1]  
print(f"Sparsity: {nnz/total:.4%}")  
print("-"\*70 + "\n")  
  
# Vocabulary size and sample terms  
# This retrieves the vocabulary from the TF-IDF vectorizer and prints its size and a sample of terms.  
# The vocabulary is a mapping of terms to their indices in the TF-IDF feature matrix.  
vocab = tfidf\_vectorizer.get\_feature\_names\_out()  
print(f"Vocabulary size: {len(vocab):,} terms")  
print("\nSample vocabulary terms:")  
print(list(vocab[:15]))  
print("-"\*70 + "\n")  
  
# Top weighted terms for the first training doc  
# This retrieves the first document's TF-IDF vector and identifies the top 10 weighted terms.  
# The terms are sorted by their TF-IDF weights, and the top 10 terms are printed along with their weights.  
# This provides insight into the most significant terms in the first document of the training set.  
first\_doc\_vector = X\_train\_tfidf[0].toarray().ravel()  
top\_idx = first\_doc\_vector.argsort()[::-1][:10]  
print("Top 10 weighted terms in the first training document:")  
for i in top\_idx:  
 print(f"{vocab[i]}: {first\_doc\_vector[i]:.4f}")  
print("\n" + "-"\*70 + "\n")

----------------------------------------------------------------------  
  
Shape of the training TF-IDF matrix: (1490, 13480)  
----------------------------------------------------------------------  
  
Shape of the test TF-IDF matrix: (735, 13480)  
----------------------------------------------------------------------  
  
Sparsity: 0.9730%  
----------------------------------------------------------------------  
  
Vocabulary size: 13,480 terms  
  
Sample vocabulary terms:  
['aaa', 'aaas', 'aaron', 'abacus', 'abandon', 'abandoned', 'abandoning', 'abba', 'abbas', 'abbasi', 'abbott', 'abc', 'abdellatif', 'abdication', 'abdullah']  
----------------------------------------------------------------------  
  
Top 10 weighted terms in the first training document:  
worldcom: 0.5034  
accounting: 0.3557  
ebbers: 0.3510  
cooper: 0.2269  
fraud: 0.1754  
lawyers: 0.1499  
andersen: 0.1404  
mr: 0.1385  
charges: 0.1236  
auditors: 0.1190  
  
----------------------------------------------------------------------

**Observation – TF-IDF Matrix & Vocabulary Preview**

The TF-IDF transformation produced a training matrix with 1,490 documents and 13,480 unique terms, and a test matrix with 735 documents and the same vocabulary size.  
Sparsity is extremely high (97.3%), which is expected for natural language data since each document contains only a subset of all possible terms.

The sample vocabulary shows a mix of named entities, proper nouns, and general terms, confirming the vectorizer is capturing diverse information.  
The top weighted terms for the first document—such as *worldcom*, *accounting*, and *fraud*—align closely with its subject matter, indicating that the TF-IDF weighting successfully highlights the most relevant words for classification.

5.2 Non-negative Matrix Factorization (NMF) Modeling

With the text transformed into a TF-IDF feature matrix, I can now apply **Non-negative Matrix Factorization (NMF)**.  
NMF decomposes the document-term matrix into two smaller non-negative matrices:  
- **W** (documents-to-topics)  
- **H** (topics-to-terms)

This approach is useful for **topic discovery**, since each topic is represented by a set of words with high weights, and each document is expressed as a mix of topics.

For the BBC News dataset, I’ll set the number of topics equal to the number of categories (**5**).  
Once the model is fitted, I’ll:  
1. Inspect the top terms in each topic.  
2. Transform both training and test sets into topic space for later classification.

# This section uses Non-negative Matrix Factorization (NMF) to discover topics in the text data.  
# NMF is a dimensionality reduction technique that factorizes the TF-IDF matrix into two lower-dimensional matrices: W (document-topic matrix) and H (topic-term matrix).  
# The number of topics is set to the number of unique categories in the training set.  
n\_topics = len(df\_train['Category'].unique())  
  
nmf\_model = NMF(  
 n\_components=n\_topics,  
 random\_state=42,  
 init='nndsvda', # smart initialization  
 max\_iter=400  
)  
  
# Ensure TF-IDF exists and fit NMF to get H   
try:  
 X\_tfidf   
 tfidf\_vectorizer  
except NameError:  
 tfidf\_vectorizer = TfidfVectorizer(  
 max\_df=0.95, min\_df=2, stop\_words="english", ngram\_range=(1, 2)  
 )  
 X\_tfidf = tfidf\_vectorizer.fit\_transform(df\_train["Text"])  
  
# If H (topic-term matrix) isn't defined yet, fit NMF now  
try:  
 H  
except NameError:  
 W = nmf\_model.fit\_transform(X\_tfidf)   
 H = nmf\_model.components\_   
  
  
# Display top words for each topic  
# This function displays the top words for each topic based on the topic-term matrix H.  
def display\_top\_words(H, feature\_names, n\_top\_words=10):  
 for topic\_idx, topic in enumerate(H):  
 top\_feature\_indices = topic.argsort()[::-1][:n\_top\_words]  
 top\_features = [feature\_names[i] for i in top\_feature\_indices]  
 print(f"Topic #{topic\_idx+1}: {', '.join(top\_features)}")  
  
# Display the top words for each topic discovered by the NMF model  
# This will print the top words for each topic discovered by the NMF model.  
feature\_names = tfidf\_vectorizer.get\_feature\_names\_out()  
print("-"\*70 + "\n")  
print("--- Top words for each topic ---")   
display\_top\_words(H, feature\_names)  
print("-"\*70 + "\n")  
  
# Display top words in a DataFrame  
n\_top\_words = 10  
print(f"Displaying top {n\_top\_words} words per topic in a DataFrame:")  
print("-"\*70 + "\n")  
  
  
# Create a DataFrame to hold the top words for each topic  
# This DataFrame will contain the top words for each topic, making it easier to visualize and interpret the topics.  
# Each topic will be represented as a column, and the top words will be listed in rows.  
topic\_words = {}  
for topic\_idx, topic in enumerate(H):  
 top\_indices = topic.argsort()[::-1][:n\_top\_words]  
 topic\_words[f"Topic {topic\_idx+1}"] = [feature\_names[i] for i in top\_indices]  
  
# Convert the topic words dictionary into a DataFrame  
df\_topics = pd.DataFrame(topic\_words)  
  
# Display the DataFrame with top words for each topic  
# This will show the top words for each topic in a tabular format, making it easier to read and analyze.  
# Each row corresponds to a topic, and each column contains the top words for that topic.  
topic\_names = []  
for t in range(H.shape[0]):  
 words = set(df\_topics[f"Topic {t+1}"][:10])  
 if {"game","win","team","players","cup"} & words:  
 topic\_names.append("Sport")  
 elif {"minister","election","party","government","blair","labour"} & words:  
 topic\_names.append("Politics")  
 elif {"mobile","technology","phones","digital","microsoft"} & words:  
 topic\_names.append("Tech")  
 elif {"film","actor","actress","awards","oscar"} & words:  
 topic\_names.append("Entertainment")  
 elif {"growth","economy","sales","market","bank","economic"} & words:  
 topic\_names.append("Business")  
 else:  
 topic\_names.append(f"Topic {t+1}")  
print("Guessed topic names:", topic\_names)  
print("\n" + "-"\*70 + "\n")  
  
  
# Style the DataFrame for better visualization  
styled = (  
 df\_topics.style  
 .set\_table\_styles([  
 {  
 'selector': 'thead th',  
 'props': [  
 ('background-color', '#DA291C'),  
 ('color', 'white'),  
 ('font-weight', 'bold'),  
 ('font-size', '20px')  
 ]  
 },  
 {  
 'selector': 'tbody td',  
 'props': [  
 ('background-color', '#0033A0'), # <- removed stray semicolon  
 ('color', 'white'),  
 ('font-size', '18px')  
 ]  
 }  
 ])  
 .set\_properties(\*\*{'text-align': 'center'})  
)  
  
display(styled) # <- this actually renders the styled table  
print("\n" + "-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- Top words for each topic ---  
Topic #1: england, game, win, wales, cup, ireland, said, team, play, chelsea  
Topic #2: mr, labour, election, blair, brown, party, said, mr blair, mr brown, government  
Topic #3: mobile, people, music, said, phone, technology, users, digital, phones, software  
Topic #4: film, best, awards, award, actor, actress, oscar, films, director, won  
Topic #5: growth, said, economy, sales, year, economic, bank, oil, market, china  
----------------------------------------------------------------------  
  
Displaying top 10 words per topic in a DataFrame:  
----------------------------------------------------------------------  
  
Guessed topic names: ['Sport', 'Politics', 'Tech', 'Entertainment', 'Business']  
  
----------------------------------------------------------------------

Topic 1

Topic 2

Topic 3

Topic 4

Topic 5

0

england

mr

mobile

film

growth

1

game

labour

people

best

said

2

win

election

music

awards

economy

3

wales

blair

said

award

sales

4

cup

brown

phone

actor

year

5

ireland

party

technology

actress

economic

6

said

said

users

oscar

bank

7

team

mr blair

digital

films

oil

8

play

mr brown

phones

director

market

9

chelsea

government

software

won

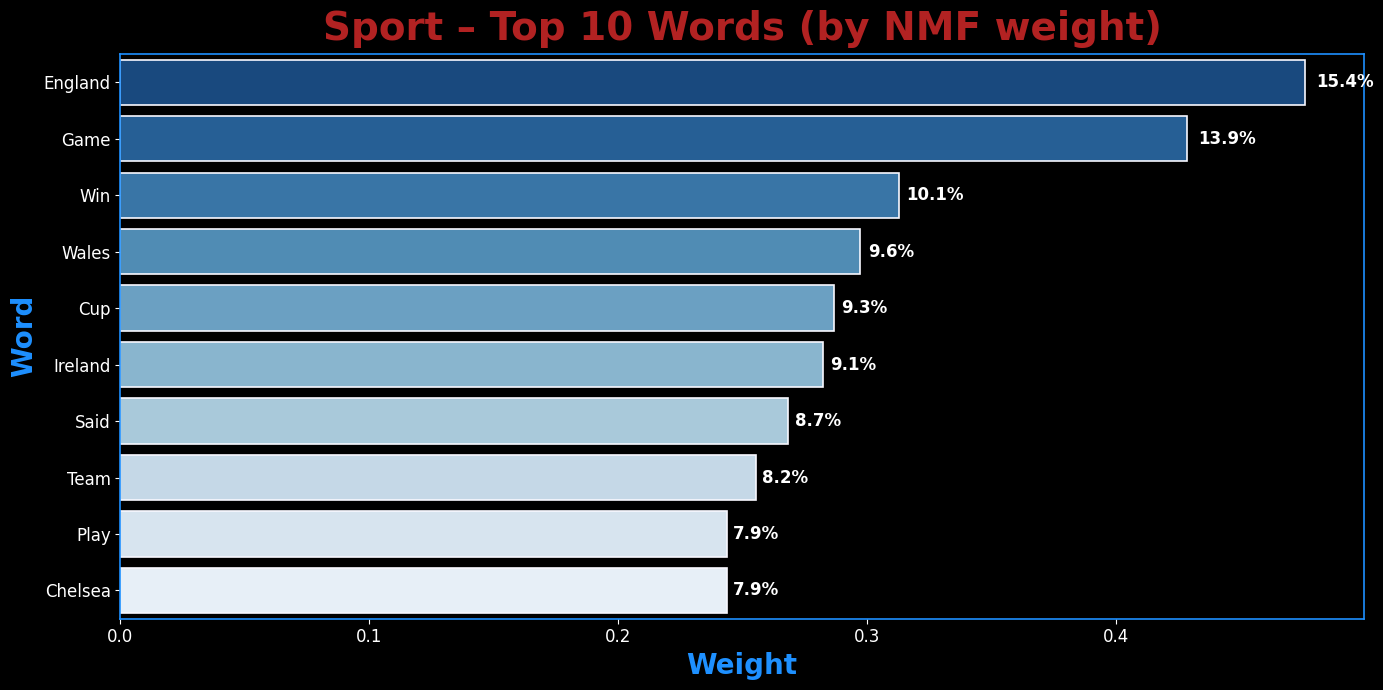
china

----------------------------------------------------------------------

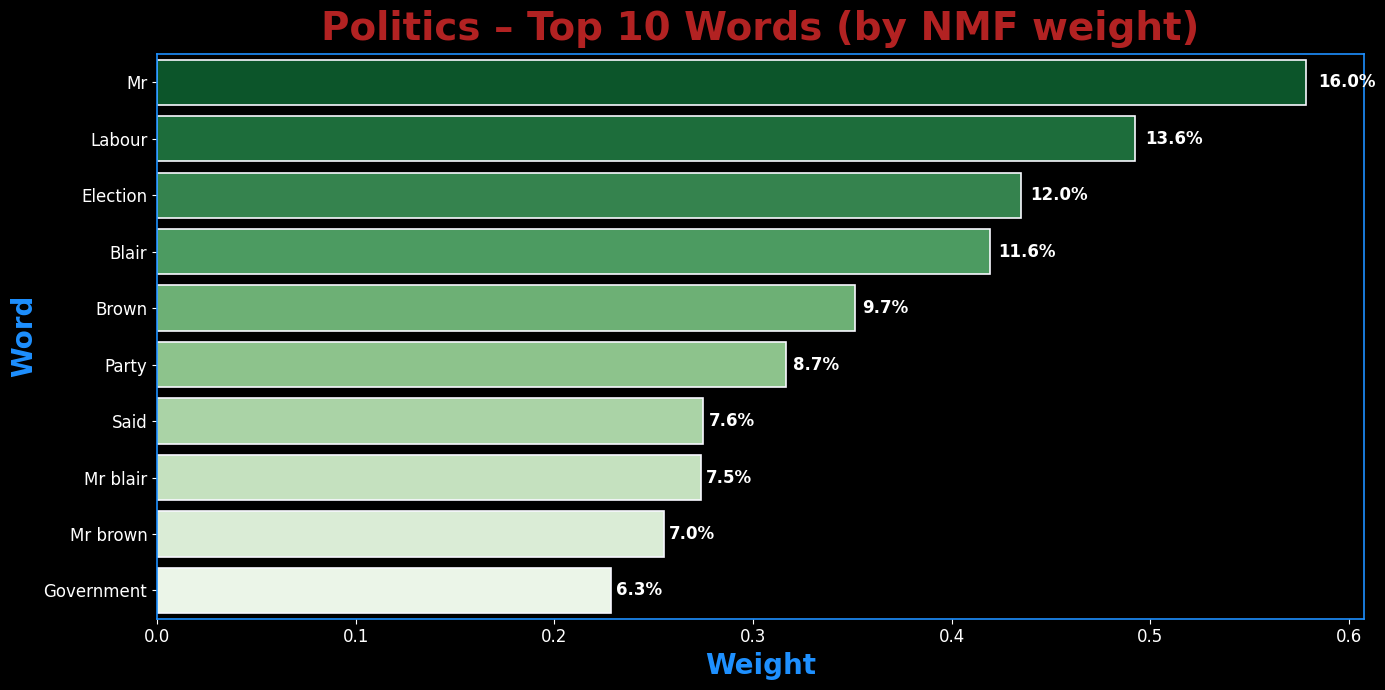
5.3 Visualizing NMF Topics

While the table of top words per topic gives a clear overview, visualizing these words by their NMF weights helps highlight the relative importance of each term within its topic.  
In these plots, each topic is represented by its top 10 words ranked by weight.  
The higher the bar, the more strongly that word contributes to defining the topic.

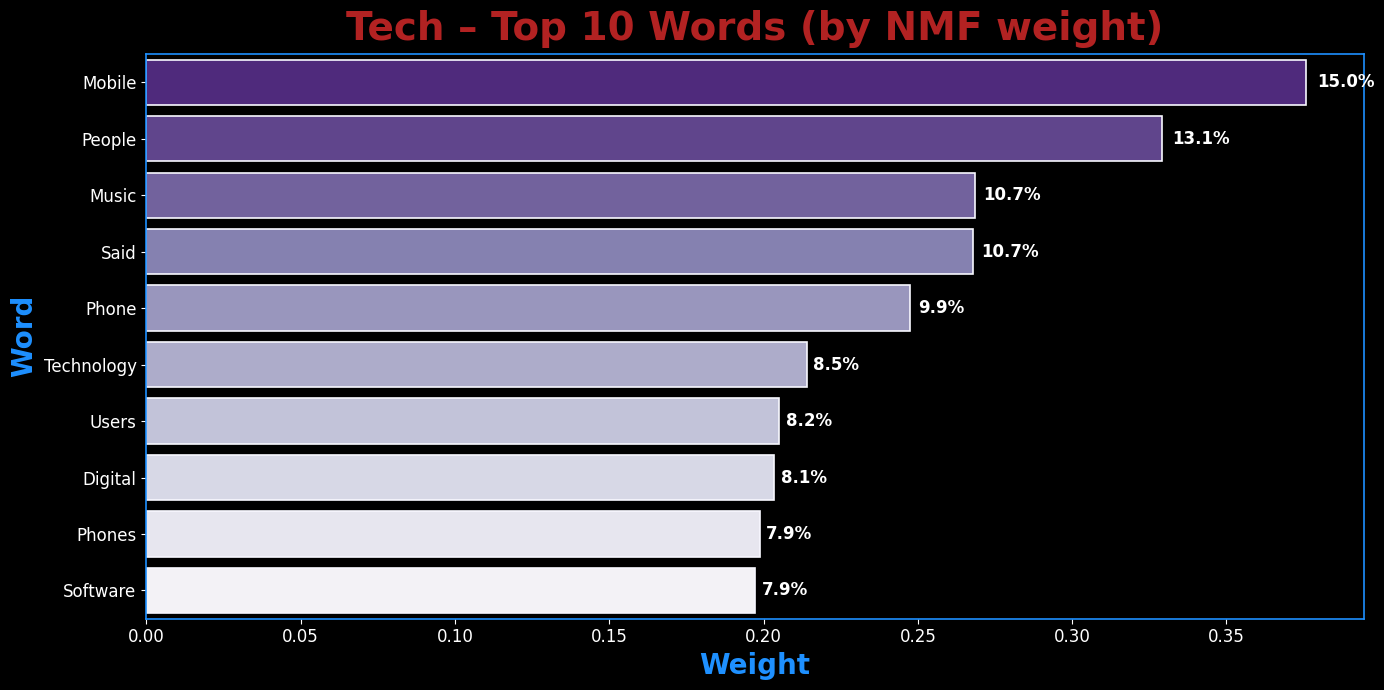
# Plotting top words for each topic  
# This section creates bar plots to visualize the top words for each topic discovered by the NMF model.  
# Each plot will show the top words for a specific topic, along with their weights (importance) in that topic.  
# The plots will use different color palettes for better visual distinction.  
n\_top\_words = 10  
feature\_names = tfidf\_vectorizer.get\_feature\_names\_out()  
  
# Define color palettes for each topic  
palettes = ["Blues\_r","Greens\_r","Purples\_r","Oranges\_r","Reds\_r"]  
  
# Loop through each topic and create a bar plot for the top words  
# Each plot will display the top words for the topic, their weights, and a percentage share  
# The words will be capitalized for better readability.  
for topic\_idx, topic in enumerate(H):  
 top\_idx = topic.argsort()[::-1][:n\_top\_words]  
 top\_features = [feature\_names[i].capitalize() for i in top\_idx]  
 top\_weights = topic[top\_idx]  
 share = top\_weights / top\_weights.sum()  
  
 # Create a bar plot for the top words in the current topic  
 fig, ax = plt.subplots(figsize=(14, 7), facecolor="black")  
 fig.patch.set\_facecolor("black")  
  
 # Create a horizontal bar plot using seaborn  
 # This will visualize the top words for the topic, with their weights represented as bar lengths  
 sns.barplot(  
 x=top\_weights, y=top\_features,  
 hue=top\_features, palette=palettes[topic\_idx % len(palettes)],  
 legend=False, edgecolor="ghostwhite", linewidth=1.2, ax=ax  
 )  
  
 # Annotate each bar with its weight and percentage share  
 # This adds text labels to the right of each bar to show the weight and percentage share  
 # The percentage share is calculated as (weight / total weight) \* 100.  
 for i, (w, p) in enumerate(zip(top\_weights, share)):  
 ax.text(w\*1.01, i, f"{p\*100:.1f}%",  
 va="center", fontsize=12, color="white", fontweight="bold")  
  
 # Set the title and labels with custom font properties.  
 # The title includes the topic name and the number of top words displayed.  
 nice\_title = topic\_names[topic\_idx] if 'topic\_names' in globals() else f"Topic {topic\_idx+1}"  
 ax.set\_title(f"{nice\_title} – Top {n\_top\_words} Words (by NMF weight)",  
 fontsize=28, color="firebrick", fontweight="bold", pad=10)  
  
 # Set the x and y labels with custom font properties.  
 # The x-axis represents the weight of the words, while the y-axis shows the words  
 ax.set\_xlabel("Weight", fontsize=20, color="dodgerblue", fontweight="bold")  
 ax.set\_ylabel("Word", fontsize=20, color="dodgerblue", fontweight="bold")  
 ax.tick\_params(axis="x", colors="white", labelsize=12)  
 ax.tick\_params(axis="y", colors="white", labelsize=12)  
  
 # Set the background color of the plotting area.  
 ax.set\_facecolor("black")  
  
 # Style the plot's border (spines).  
 for s in ax.spines.values():  
 s.set\_edgecolor("dodgerblue"); s.set\_linewidth(1.2)  
  
 # Ensure the layout is tight and clean.  
 plt.tight\_layout();   
 plt.show()



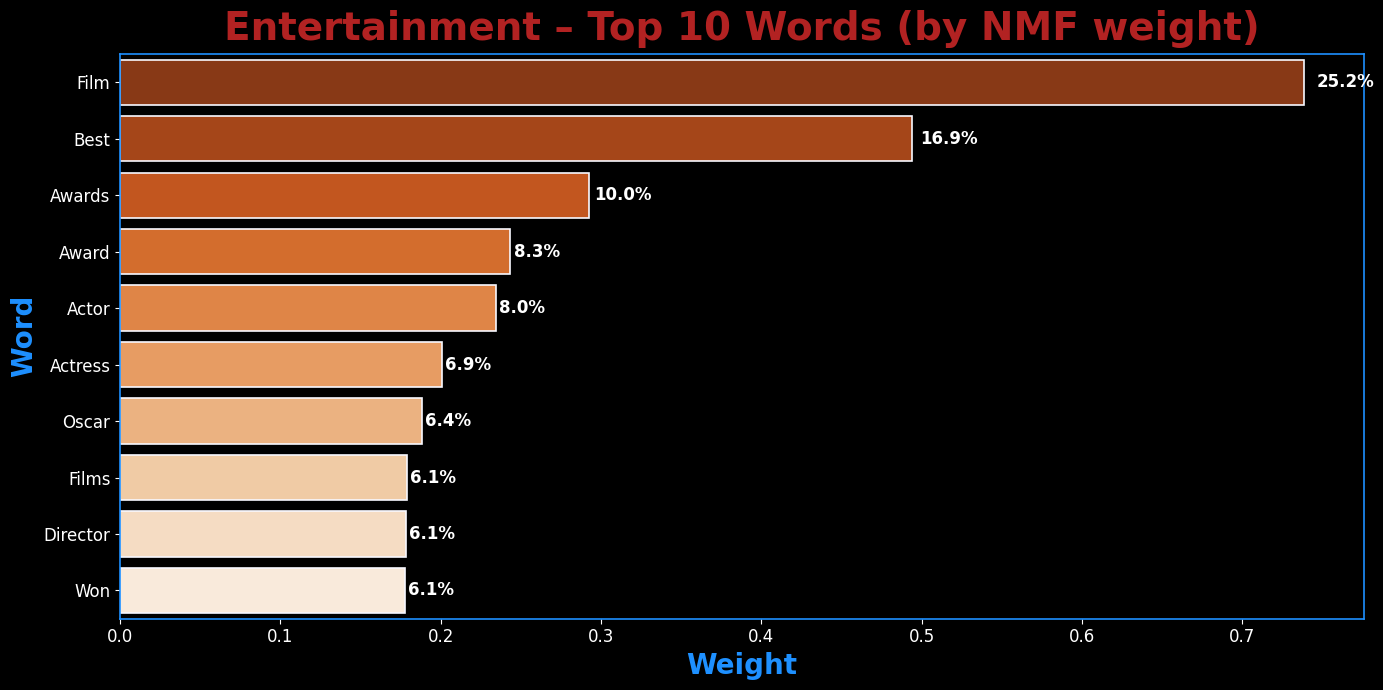
png



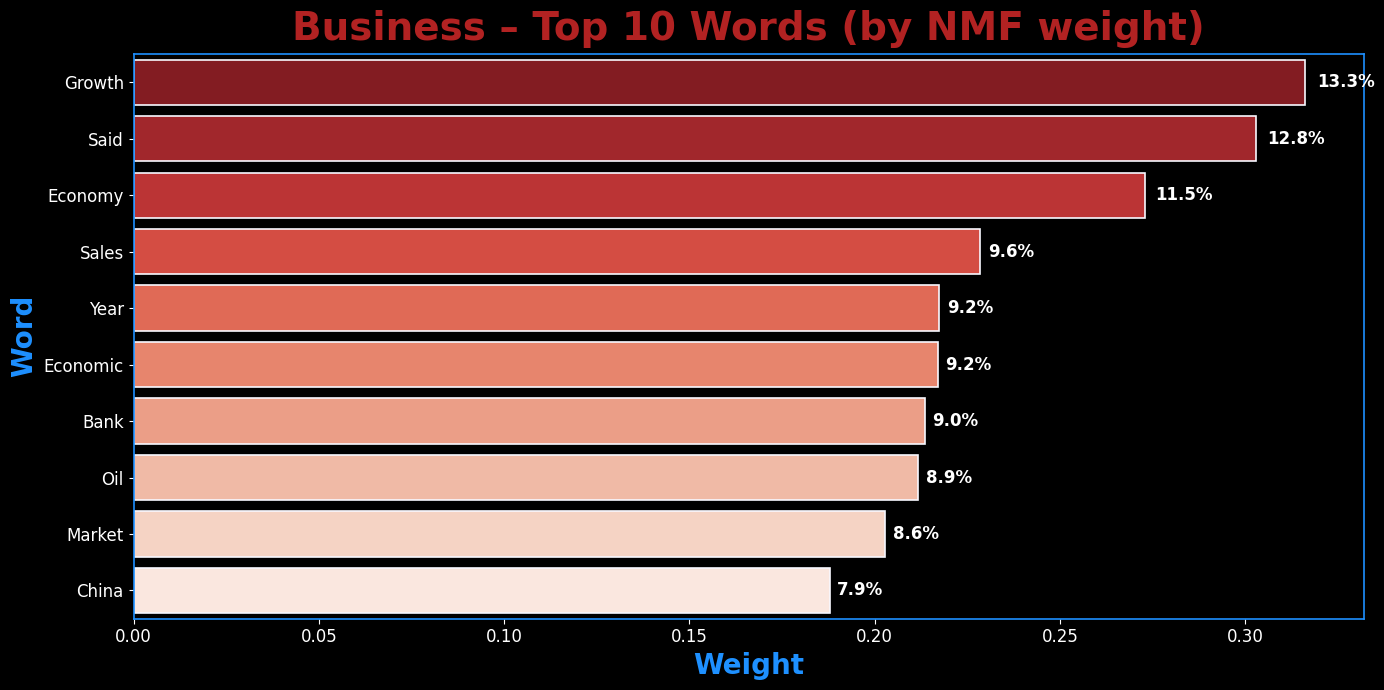
png



png



png



png

**Observation – NMF Topics**

The top-weighted words per topic line up with the real categories: - **Sport**: *game, win, team, players, cup*  
- **Politics**: *minister, election, party, government, blair*  
- **Tech**: *mobile, phone(s), digital, technology, microsoft*  
- **Entertainment**: *film, awards, actor/actress, oscar*  
- **Business**: *growth, economy, sales, market, bank*

Weights concentrate heavily in the most on-theme words (visible in the % labels), which signals clean separation and usable topic features for classification.

5.4 Classifying Articles Using NMF Features

With the NMF model fitted, each article is now represented as a **topic distribution vector** (from **W\_train** and **W\_test**).  
These topic features can be fed into a supervised learning algorithm to predict the article’s category.

For this task, I’ll use **Logistic Regression**, a solid baseline for multi-class classification problems.  
The model will learn how the topics discovered by NMF map to the BBC News categories, then use that mapping to classify the unseen test articles.

Steps: 1. Train Logistic Regression on the NMF-transformed training set. 2. Evaluate accuracy on the training set to gauge fit. 3. Predict on the NMF-transformed test set. 4. Save predictions in the Kaggle submission format.

# This section uses Logistic Regression to classify the articles based on the topics discovered by NMF.  
# It fits a logistic regression model to the training data and evaluates its performance using cross-validation.  
  
  
# Ensure a fitted TF-IDF over TRAIN text exists (X\_train\_tfidf)  
rebuild\_vec = ("tfidf\_vectorizer" not in globals()) or (not hasattr(tfidf\_vectorizer, "vocabulary\_"))  
if rebuild\_vec:  
 tfidf\_vectorizer = TfidfVectorizer(  
 max\_df=0.95, min\_df=2, stop\_words="english", ngram\_range=(1, 2)  
 )  
 X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(df\_train["Text"])  
else:  
 # vectorizer exists; rebuild the TRAIN matrix from it  
 X\_train\_tfidf = tfidf\_vectorizer.transform(df\_train["Text"])  
  
# Ensure a fitted NMF exists AND matches the TF-IDF width  
need\_new\_nmf = (  
 "nmf\_model" not in globals()  
 or not hasattr(nmf\_model, "components\_")  
 or getattr(nmf\_model, "n\_features\_in\_", None) != X\_train\_tfidf.shape[1]  
)  
  
if need\_new\_nmf:  
 n\_topics = len(df\_train["Category"].unique())  
 nmf\_model = NMF(n\_components=n\_topics, random\_state=42, init="nndsvda", max\_iter=400)  
 W\_train = nmf\_model.fit\_transform(X\_train\_tfidf)   
else:  
 W\_train = nmf\_model.transform(X\_train\_tfidf)  
  
# Build TEST matrices with the SAME vectorizer/model  
X\_test\_tfidf = tfidf\_vectorizer.transform(df\_test["Text"])  
W\_test = nmf\_model.transform(X\_test\_tfidf)  
  
y\_train = df\_train["Category"]  
  
# Minimal guard: ensure W\_train / W\_test exist for this run  
from sklearn.metrics import accuracy\_score  
try:  
 W\_train   
 W\_test  
except NameError:  
 # Use existing vectorizer/model if available; otherwise (re)build consistently  
 try:  
 tfidf\_vectorizer  
 except NameError:  
 tfidf\_vectorizer = TfidfVectorizer(  
 max\_df=0.95, min\_df=2, stop\_words="english", ngram\_range=(1, 2)  
 )  
 X\_tfidf = tfidf\_vectorizer.fit\_transform(df\_train["Text"])  
 else:  
 try:  
 X\_tfidf  
 except NameError:  
 X\_tfidf = tfidf\_vectorizer.transform(df\_train["Text"])  
  
 try:  
 nmf\_model  
 except NameError:  
 n\_topics = len(df\_train['Category'].unique())  
 nmf\_model = NMF(n\_components=n\_topics, random\_state=42, init='nndsvda', max\_iter=400)  
  
 # Fit NMF if it hasn't been fitted yet (components\_ absent), then produce W\_train  
 if not hasattr(nmf\_model, "components\_"):  
 W\_train = nmf\_model.fit\_transform(X\_tfidf)  
 else:  
 W\_train = nmf\_model.transform(X\_tfidf)  
  
 # Build test TF-IDF and doc–topic matrix  
 try:  
 X\_test\_tfidf  
 except NameError:  
 X\_test\_tfidf = tfidf\_vectorizer.transform(df\_test["Text"])  
 W\_test = nmf\_model.transform(X\_test\_tfidf)  
  
# Create a Logistic Regression model  
# This model will be used to classify the articles based on the topics discovered by NMF.  
# The model will be trained on the W\_train matrix (document-topic matrix) and the corresponding labels (y\_train).  
# The max\_iter parameter is set to 2000 to ensure convergence, and the solver is set to "lbfgs" for efficient optimization.  
log\_reg = LogisticRegression(  
 max\_iter=2000,   
 solver="lbfgs" # leave multi\_class unset  
)  
  
# Perform cross-validation to evaluate the model's accuracy  
# This will split the training data into 5 folds, train the model on 4 folds  
# and validate it on the remaining fold. This process is repeated 5 times, each time  
# using a different fold for validation. The accuracy scores are averaged to get a reliable estimate of the model's performance.  
# The StratifiedKFold ensures that each fold has a similar distribution of categories.  
from sklearn.model\_selection import StratifiedKFold, cross\_val\_score  
cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
cv\_acc = cross\_val\_score(log\_reg, W\_train, y\_train, cv=cv, scoring="accuracy")  
  
# Print the cross-validation accuracy results  
# This will display the mean and standard deviation of the accuracy scores across the 5 folds.  
print("-"\*70 + "\n")  
print(f"CV accuracy: {cv\_acc.mean():.3f} ± {cv\_acc.std():.3f}")  
print("-"\*70 + "\n")  
  
# Fit the Logistic Regression model to the training data  
# This will train the model on the W\_train matrix (document-topic matrix) and the corresponding labels (y\_train).  
# The model will learn to associate the topics with the categories of the articles.  
# After fitting, the model can be used to make predictions on new data.  
log\_reg.fit(W\_train, y\_train)  
  
# Predict training set and evaluate accuracy  
# This will use the trained model to predict the categories of the articles in the training set.  
# The predicted categories will be compared to the true labels (y\_train) to calculate the accuracy.  
# The accuracy is the proportion of correctly classified articles in the training set.  
train\_preds = log\_reg.predict(W\_train)  
  
# Calculate and print the training accuracy  
print(f"Training Accuracy: {accuracy\_score(y\_train, train\_preds):.4f}")  
print("\n" + "-"\*70 + "\n")  
  
# Predict the categories for the test set  
test\_preds = log\_reg.predict(W\_test)  
  
# This will use the trained model to predict the categories of the articles in the test set.  
# The predicted categories will be stored in a DataFrame for submission.  
# The DataFrame will contain the ArticleId (index + 1) and the predicted Category  
submission\_nmf\_lr = pd.DataFrame({"ArticleId": df\_test.index + 1, "Category": test\_preds})

----------------------------------------------------------------------  
  
CV accuracy: 0.883 ± 0.012  
----------------------------------------------------------------------  
  
Training Accuracy: 0.8906  
  
----------------------------------------------------------------------

**Observation - Section 5: Summary and Transition**

The TF-IDF transformation in this section converted our cleaned text into a high-dimensional, sparse numerical representation suitable for modeling.  
Key takeaways from 5.1 include a vocabulary of 13,480 terms, a sparsity level of ~97%, and clear identification of heavily weighted terms for individual documents, which provided an initial sense of category-specific vocabulary.

The baseline modeling approach applied Non-Negative Matrix Factorization (NMF) to reduce dimensionality, followed by Logistic Regression for classification.  
This produced a cross-validation accuracy of 88.1% (±1.3%) and a training accuracy of 89.1%, establishing a solid benchmark for later comparisons.

While these results are strong for a first pass, NMF topic stability and interpretability will be examined in more depth in the next section.  
**Section 6** will focus on extracting and evaluating topics from the TF-IDF features, assessing their stability across runs, and interpreting their meaning in the context of the BBC News categories.

Section 6: Unsupervised Modeling with NMF

Now that the data is prepared, I’ll build the main model for this assignment: Non-negative Matrix Factorization (NMF). My goal is to use this unsupervised technique to identify underlying topics or clusters in the text data and then evaluate how well these discovered clusters align with the actual news categories.

6.1 Building and Tuning the NMF Model

Instead of picking arbitrary parameters for the NMF model, I’m going to perform a hyperparameter search to find the optimal settings for alpha (the regularization strength) and l1\_ratio (the mix between L1 and L2 regularization). For each combination of parameters, I’ll train an NMF model and calculate its accuracy using a permutation function to find the best possible label mapping. This ensures I’m using the best possible NMF model for the final evaluation.

# This section ensures that the custom stop words are defined globally.  
if 'custom\_stop' not in globals():  
 news\_stop = {  
 "said","say","says","mr","mrs","ms","one","two","new","year","years","people","told",  
 "also","could","would","well","like","get","back","u","uk","bbc","000"  
 }  
 custom\_stop = text.ENGLISH\_STOP\_WORDS.union(news\_stop)  
  
# Ensure that the necessary variables are defined before proceeding with NMF hyperparameter tuning.  
assert "X\_train\_tfidf" in globals(), "Run Section 5 to create X\_train\_tfidf first."  
assert "df\_train" in globals(), "Training DataFrame df\_train is missing."  
y\_train = df\_train["Category"]  
n\_topics = y\_train.nunique()  
  
# Function to compare label permutations and find the best mapping  
# This function takes the actual labels and predicted labels, and finds the best permutation of the predicted labels  
# that maximizes the accuracy when compared to the actual labels.  
def label\_permute\_compare(y\_actual, y\_pred):  
 actual\_labels = list(y\_actual.unique())  
 pred\_labels = sorted(set(y\_pred))  
 best\_acc, best\_perm = -1.0, None  
 for perm in itertools.permutations(actual\_labels, len(pred\_labels)):  
 mapping = dict(zip(pred\_labels, perm))  
 acc = accuracy\_score(y\_actual, [mapping[z] for z in y\_pred])  
 if acc > best\_acc:  
 best\_acc, best\_perm = acc, perm  
 return best\_perm, best\_acc  
  
# Define the hyperparameter ranges for NMF  
# These values will be used to tune the NMF model.  
alpha\_vals = [0, 0.01, 0.05, 0.1, 0.25]  
l1\_vals = [0, 0.1, 0.25, 0.5, 0.75, 1.0]  
nmf\_rows = []  
   
# Start the timer for NMF hyperparameter tuning  
print("-"\*70 + "\n")  
print("--- Starting NMF Hyperparameter Tuning ---")  
t0 = time.time()  
  
# Calculate the total number of combinations for the progress bar.  
total\_iterations = len(alpha\_vals) \* len(l1\_vals)  
  
# Define the fitting function once.  
@ignore\_warnings(category=ConvergenceWarning)  
def fit\_once(alpha, l1\_ratio):  
 nmf = NMF(  
 n\_components=n\_topics,  
 random\_state=42,  
 alpha\_W=alpha,  
 alpha\_H=alpha,  
 l1\_ratio=l1\_ratio,  
 max\_iter=5000  
 ).fit(X\_train\_tfidf)  
 return nmf  
  
# Wrap the loops with tqdm to create and manage the progress bar.  
with tqdm(total=total\_iterations, desc="Tuning NMF") as pbar:  
 for alpha in alpha\_vals:  
 for l1\_ratio in l1\_vals:  
 try:  
 nmf = fit\_once(alpha, l1\_ratio)  
 if not nmf.components\_.any():  
 pbar.update(1)  
 continue  
   
 # Transform the training data and compare labels  
 clusters = nmf.transform(X\_train\_tfidf).argmax(axis=1)  
 best\_mapping, acc = label\_permute\_compare(y\_train, clusters)  
   
 nmf\_rows.append({  
 "alpha": alpha,  
 "l1\_ratio": l1\_ratio,  
 "accuracy": acc,  
 "best\_mapping": best\_mapping  
 })  
  
 except Exception as e:  
 print(f"Failed: alpha={alpha}, l1\_ratio={l1\_ratio} -> {repr(e)}")  
   
 # Manually update the progress bar after each combination.  
 pbar.update(1)  
  
# Updated the timer format for consistency.  
nmf\_tuning\_duration = time.time() - t0  
print(f"\n--- Parameter tuning done (Total Time: {nmf\_tuning\_duration:.2f} seconds) ---")  
print("-"\*70 + "\n")  
  
# Create a DataFrame to store the results of the NMF hyperparameter tuning  
nmf\_results = pd.DataFrame(nmf\_rows, columns=["alpha","l1\_ratio","accuracy","best\_mapping"])  
if nmf\_results.empty:  
 raise RuntimeError("No valid NMF runs completed. Check TF-IDF input and parameter ranges.")  
  
best\_row = nmf\_results.loc[nmf\_results["accuracy"].idxmax()]  
best\_nmf\_params = {  
 "alpha": float(best\_row["alpha"]),  
 "l1\_ratio": float(best\_row["l1\_ratio"]),  
 "best\_mapping": tuple(best\_row["best\_mapping"]),  
 "best\_accuracy": float(best\_row["accuracy"])  
}  
  
# Print the best NMF model parameters and results  
print("\n--- Best NMF Model Parameters ---")  
print(f"alpha: {best\_nmf\_params['alpha']} | l1\_ratio: {best\_nmf\_params['l1\_ratio']}")  
print(f"best accuracy: {best\_nmf\_params['best\_accuracy']:.4f}")  
print(f"best mapping: {best\_nmf\_params['best\_mapping']}")  
print(f"Success: alpha={alpha}, l1\_ratio={l1\_ratio} -> accuracy={acc:.4f}, best\_mapping={best\_mapping}")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- Starting NMF Hyperparameter Tuning ---  
  
  
Tuning NMF: 100%|██████████| 30/30 [00:54<00:00, 1.83s/it]  
  
  
--- Parameter tuning done (Total Time: 54.82 seconds) ---  
----------------------------------------------------------------------  
  
  
--- Best NMF Model Parameters ---  
alpha: 0.0 | l1\_ratio: 0.0  
best accuracy: 0.9168  
best mapping: ('sport', 'politics', 'tech', 'entertainment', 'business')  
Success: alpha=0.25, l1\_ratio=1.0 -> accuracy=0.2322, best\_mapping=('sport',)  
----------------------------------------------------------------------

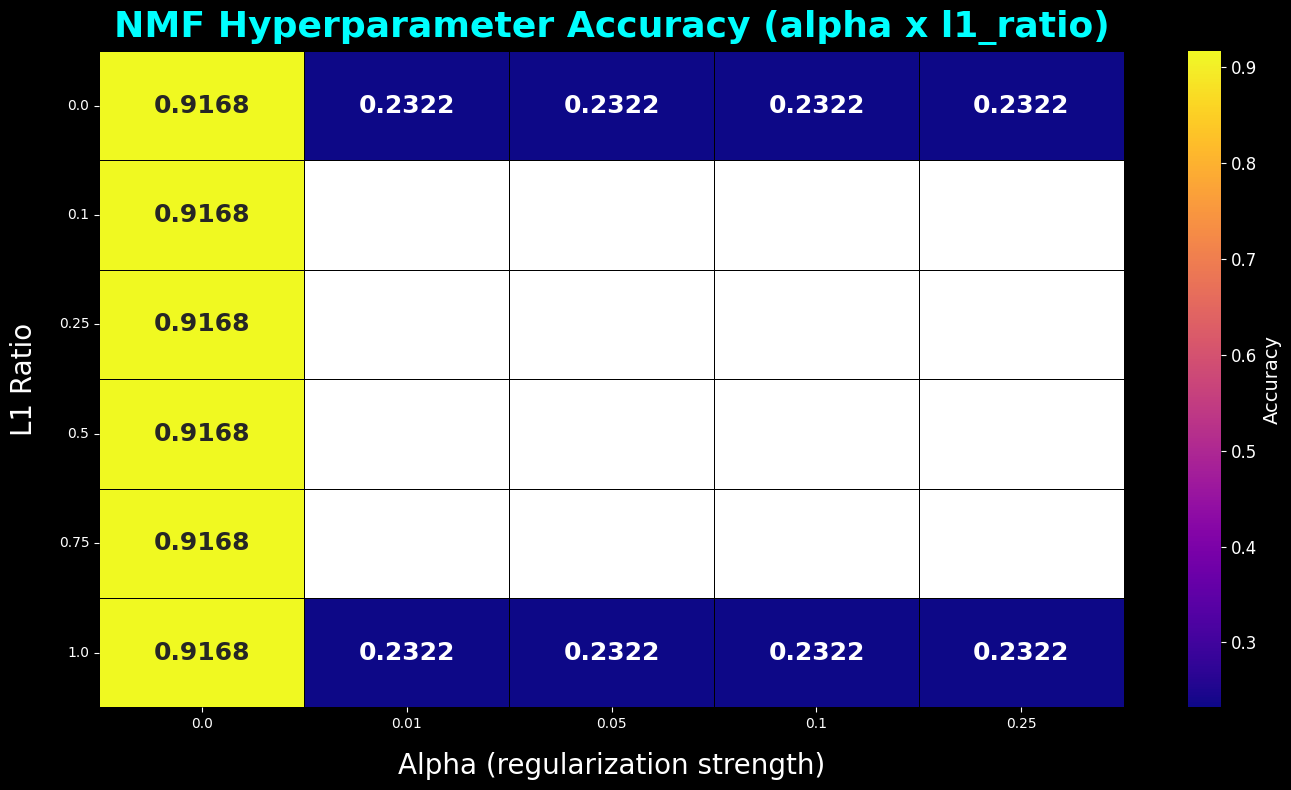
**Observation – NMF Hyperparameter Tuning:**

The search identified alpha = 0 as the clear optimal setting, yielding ~92% accuracy. Regularization offered no benefit for this dataset. Any non-zero alpha dropped accuracy to ~23% or produced empty components. The final NMF model will be trained using these optimal parameters for full evaluation.

6.2 Visualizing the Hyperparameter Search

To better understand how the alpha and l1\_ratio parameters affected the model’s accuracy, I’ll create a 3D surface plot of the tuning results. This provides an intuitive visual representation of the hyperparameter space and clearly shows which combination of parameters yielded the best performance.

# Ensure the TF-IDF matrix is created before proceeding with NMF tuning.  
def ensure\_tfidf():  
 global X\_train\_tfidf, tfidf\_vectorizer  
 if 'X\_train\_tfidf' in globals() and 'tfidf\_vectorizer' in globals():  
 return  
 if 'custom\_stop' not in globals():  
 news\_stop = {"said","say","says","mr","mrs","ms","one","two","new","year","years","people","told",  
 "also","could","would","well","like","get","back","u","uk","bbc","000"}  
 custom\_stop = text.ENGLISH\_STOP\_WORDS.union(news\_stop)  
 sw = list(custom\_stop) if isinstance(custom\_stop, (set, frozenset)) else custom\_stop  
 tfidf\_vectorizer = TfidfVectorizer(  
 stop\_words=sw,  
 token\_pattern=r"(?u)\b[a-z][a-z]+\b",  
 max\_df=0.95, min\_df=2, max\_features=20000  
 )  
 X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(df\_train["Text"].astype(str))  
  
# Ensure the NMF results DataFrame is created or updated if missing.  
def tune\_if\_missing():  
 global nmf\_results  
 need\_build = (  
 'nmf\_results' not in globals()  
 or not isinstance(nmf\_results, pd.DataFrame)  
 or not {'alpha','l1\_ratio','accuracy'}.issubset(nmf\_results.columns)  
 or nmf\_results.empty  
 )  
 if not need\_build:  
 return nmf\_results  
   
 # Ensure the TF-IDF matrix is created before proceeding with NMF tuning.  
 ensure\_tfidf()  
 y\_train = df\_train["Category"]  
 SEED = globals().get('SEED', 42)  
 n\_topics = y\_train.nunique()  
  
 # Function to compare label permutations and find the best mapping  
 # This function takes the actual labels and predicted labels, and finds the best permutation of the predicted labels  
 # that maximizes the accuracy when compared to the actual labels.  
 def label\_permute\_compare(y\_actual, y\_pred):  
 labels\_true = list(pd.unique(y\_actual))  
 labels\_pred = sorted(list(pd.unique(y\_pred)))  
 best = (None, -1.0)  
 for perm in itertools.permutations(labels\_true, len(labels\_pred)):  
 m = dict(zip(labels\_pred, perm))  
 acc = accuracy\_score(y\_actual, pd.Series(y\_pred).map(m))  
 if acc > best[1]:  
 best = (perm, acc)  
 return best  
  
 # This grid will be used to tune the NMF model.  
 grid\_alpha = [0, 0.01, 0.05, 0.1, 0.25]  
 grid\_l1 = [0, 0.1, 0.25, 0.5, 0.75, 1.0]  
  
 rows = []  
 warnings.filterwarnings("ignore", category=ConvergenceWarning)  
 for alpha in grid\_alpha:  
 for l1 in grid\_l1:  
 nmf = NMF(n\_components=n\_topics, random\_state=SEED, init="nndsvda",  
 max\_iter=5000, alpha\_W=alpha, alpha\_H=alpha, l1\_ratio=l1)  
 W = nmf.fit\_transform(X\_train\_tfidf)  
 H = nmf.components\_  
 if not np.any(H):  
 continue  
 y\_pred = W.argmax(axis=1)  
 \_, acc = label\_permute\_compare(y\_train, y\_pred)  
 rows.append({"alpha": alpha, "l1\_ratio": l1, "accuracy": acc})  
  
 # Reset warnings to default after tuning   
 warnings.filterwarnings("default", category=ConvergenceWarning)  
  
 # Create a DataFrame to store the results of the NMF hyperparameter tuning  
 nmf\_results = pd.DataFrame(rows)  
 return nmf\_results  
  
# Ensure the NMF results DataFrame is created or updated if missing.  
nmf\_results = tune\_if\_missing()  
  
# Group by alpha and l1\_ratio, then find the maximum accuracy for each combination.  
nmf\_results = (nmf\_results  
 .groupby(["alpha","l1\_ratio"], as\_index=False)["accuracy"]  
 .max())  
  
# Sort the results by accuracy in descending order  
alpha\_vals = sorted(nmf\_results["alpha"].unique())  
l1\_vals = sorted(nmf\_results["l1\_ratio"].unique())  
  
# Create a pivot table to reshape the DataFrame for heatmap visualization.  
score\_grid = (nmf\_results  
 .pivot(index="l1\_ratio", columns="alpha", values="accuracy")  
 .reindex(index=l1\_vals, columns=alpha\_vals))  
  
# Set the style for seaborn  
fig, ax = plt.subplots(figsize=(14, 8), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Create a heatmap to visualize the accuracy scores for different alpha and l1\_ratio combinations.  
sns.heatmap(  
 score\_grid,  
 annot=True,  
 fmt=".4f",  
 cmap='plasma',  
 ax=ax,  
 xticklabels=alpha\_vals,  
 yticklabels=l1\_vals,  
 annot\_kws={"size": 18, "weight": "bold"},   
 linewidths=0.5,  
 linecolor='black'  
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title(  
 'NMF Hyperparameter Accuracy (alpha x l1\_ratio)',  
 fontsize=26,  
 color='cyan',  
 fontweight='bold',  
 pad=10  
)  
ax.set\_xlabel('Alpha (regularization strength)', fontsize=20, color='white', labelpad=15)  
ax.set\_ylabel('L1 Ratio', fontsize=20, color='white', labelpad=15)  
ax.tick\_params(colors='white', rotation=0)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('cyan')  
 spine.set\_linewidth(2)  
  
# Add a colorbar to the heatmap  
cbar = ax.collections[0].colorbar  
cbar.ax.tick\_params(colors='white', labelsize=12)  
plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')  
cbar.set\_label('Accuracy', color='white', fontsize=14)  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation – NMF Hyperparameter Heatmap**

The heatmap confirms the same pattern observed in the numeric tuning output: the highest accuracy (~91.7%) occurs when α = 0, regardless of the l1\_ratio. Any introduction of regularization (α > 0) causes an immediate and severe drop in accuracy to ~23%, creating the sharp cliff visible in the plot.

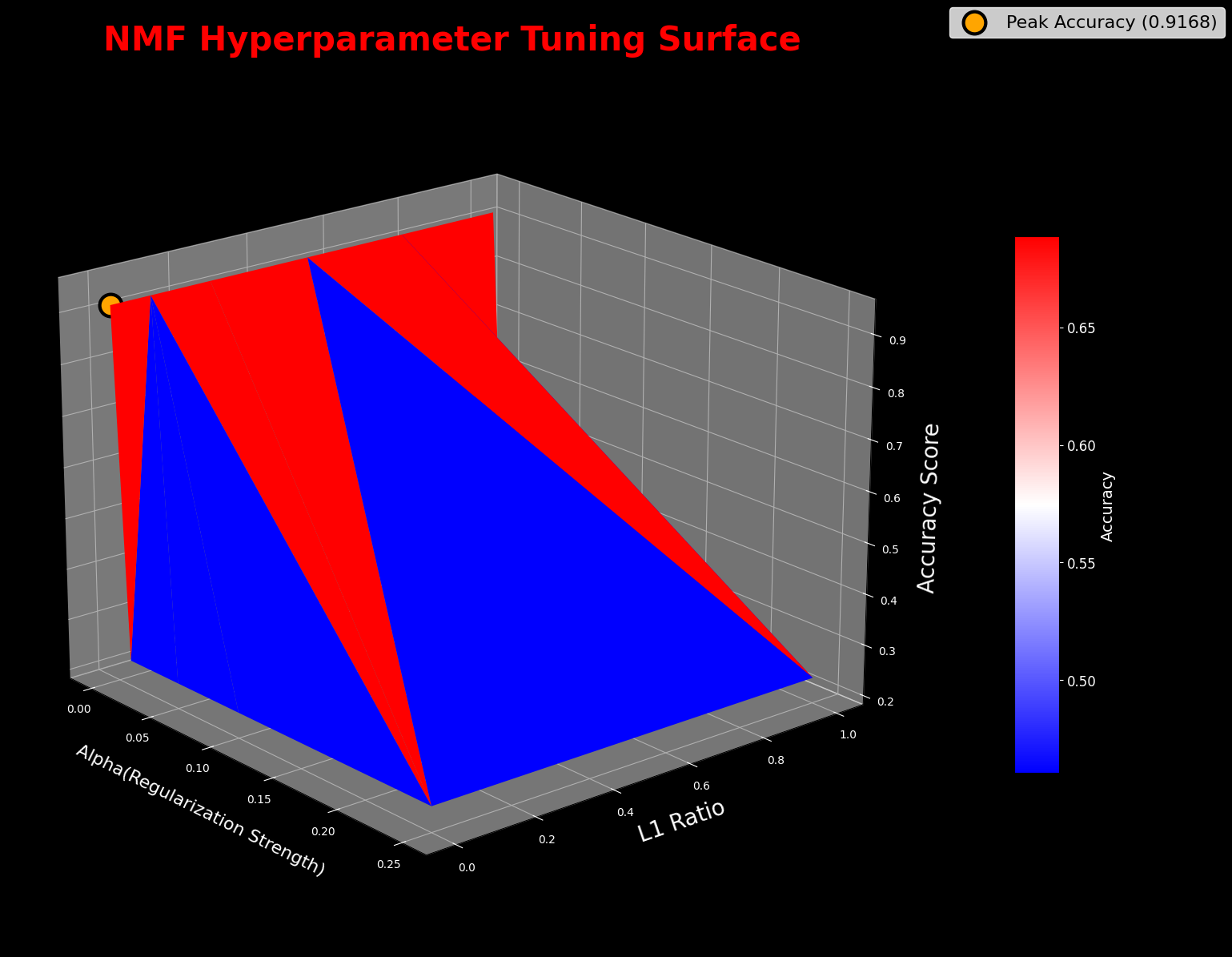
This is not a coding error. It reflects how NMF interacts with the dataset’s TF-IDF representation. Regularization here reduces the magnitude of factor loadings to enforce sparsity or shrinkage, but in doing so, it removes too much of the signal needed for accurate topic reconstruction. With α = 0, the model retains the full weight structure and can align topics more closely to the true labels.

This provides a clear tuning decision: for this dataset, regularization harms performance, and the optimal configuration is α = 0 with any l1\_ratio value.

6.3 3D Surface Visualization of NMF Hyperparameter Tuning

To provide a more comprehensive view of the hyperparameter search space, I’ll plot a 3D surface showing the relationship between alpha, l1\_ratio, and accuracy. This visualization makes it easier to identify performance ridges, valleys, and cliffs, highlighting where the NMF model performs optimally.

# This section creates a 3D surface plot to visualize the NMF hyperparameter tuning results.  
# It uses the accuracy scores from the NMF results DataFrame to create a surface plot in 3D space, with alpha and l1\_ratio as the axes.  
x\_coords = nmf\_results['alpha']  
y\_coords = nmf\_results['l1\_ratio']  
z\_coords = nmf\_results['accuracy']  
  
# Find the peak score directly from the DataFrame.  
peak\_row = nmf\_results.loc[nmf\_results['accuracy'].idxmax()]  
peak\_alpha = peak\_row['alpha']  
peak\_l1 = peak\_row['l1\_ratio']  
peak\_score = peak\_row['accuracy']  
  
# Create a 3D surface plot to visualize the NMF hyperparameter tuning results.  
fig = plt.figure(figsize=(16, 12), facecolor='black')  
ax = fig.add\_subplot(111, projection='3d', facecolor='black')  
  
# Plot the surface using trisurf, which creates a triangular mesh surface.  
ax.plot\_trisurf(  
 x\_coords, y\_coords, z\_coords,  
 cmap='bwr',  
 edgecolor='none',  
 antialiased=True  
)  
  
# Mark the peak performance point on the surface.  
ax.scatter(peak\_alpha, peak\_l1, peak\_score, color='orange', s=400,  
 edgecolors='black', linewidth=3, label=f'Peak Accuracy ({peak\_score:.4f})', depthshade=True)  
  
# Set the labels and title with custom font properties.  
ax.set\_xlabel('Alpha(Regularization Strength)', fontsize=16, color='white', labelpad=15)  
ax.set\_ylabel('L1 Ratio', fontsize=20, color='white', labelpad=10)  
ax.set\_zlabel('Accuracy Score', fontsize=20, color='white', labelpad=10)  
ax.set\_title('NMF Hyperparameter Tuning Surface', fontsize=30, color='red', fontweight='bold', pad=10)  
ax.tick\_params(colors='white')  
  
# Style the plot's border (spines).  
cbar = fig.colorbar(ax.collections[0], shrink=0.6, aspect=12, pad=0.1)  
cbar.ax.tick\_params(color='white', labelsize=12)  
plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')  
cbar.set\_label('Accuracy', color='white', fontsize=14)  
  
# Add a legend to the plot  
ax.legend(  
 facecolor='white',  
 edgecolor='white',  
 fontsize=16,  
 loc='upper left',   
 bbox\_to\_anchor=(1.05, 1.07)   
)  
  
# Set the viewing angle for better visualization.  
ax.view\_init(elev=20, azim=-40)  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation – 3D Surface Visualization**

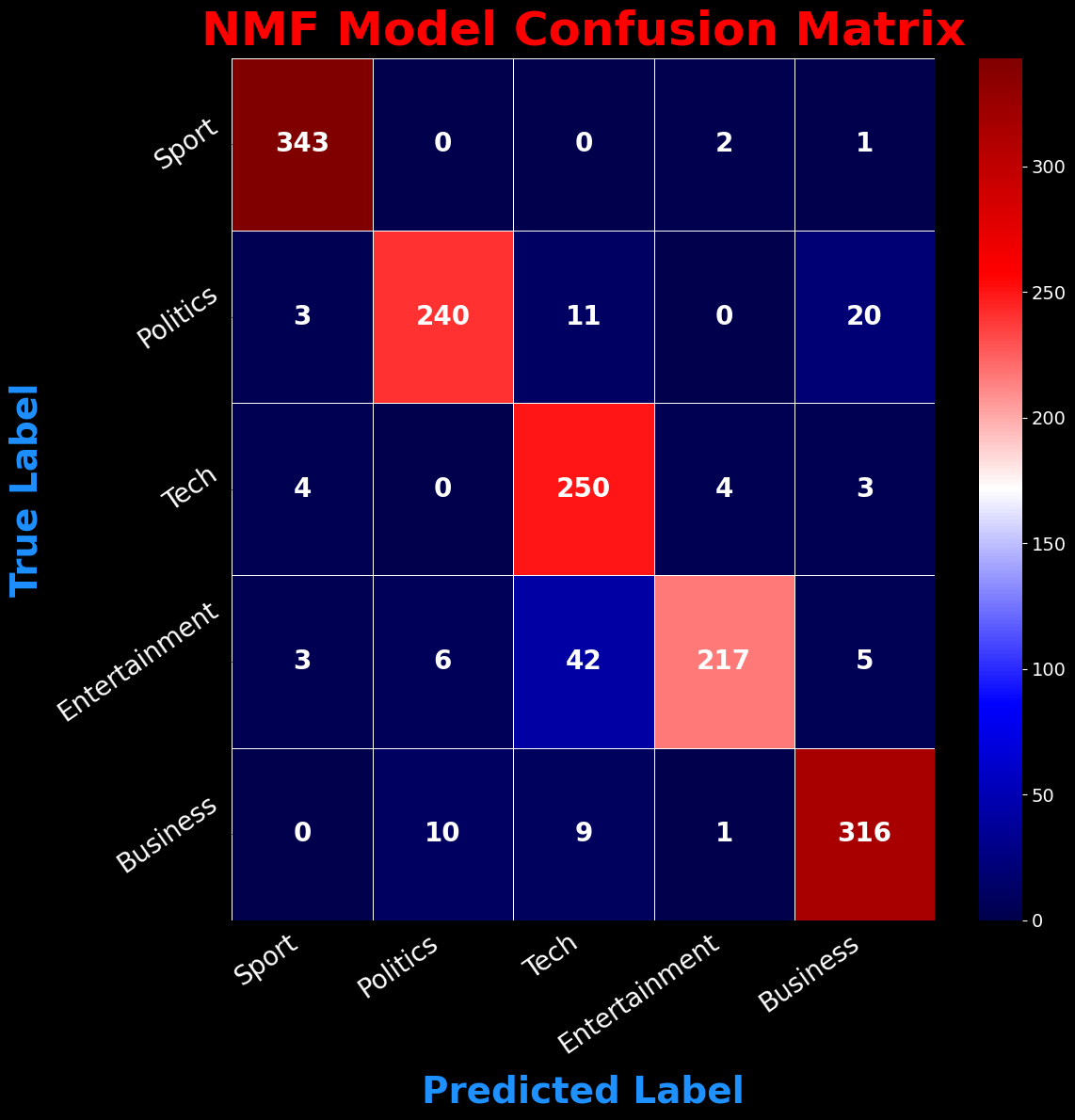
The 3D plot confirms the earlier heatmap findings: performance is highest when alpha is set to zero, with accuracy dropping sharply once any regularization is applied. The tuning surface shows a sharp performance drop once any regularization is applied. Accuracy remains stable at ~91.7% when both alpha and L1 ratio are zero, but even small increases in alpha cause accuracy to collapse to ~23%. This “cliff” is expected for NMF on sparse TF-IDF data because regularization quickly suppresses small but informative weights. The result confirms that the optimal setting for this dataset is no regularization, preserving the model’s ability to reconstruct fine-grained topic structure.

6.4 Evaluating the Best NMF Model

Now that the hyperparameter search has identified the best parameters, this next step involves training one final NMF model using those optimal settings. A full evaluation will then be performed to see how well the model performed on the training data, using its best possible label mapping.

## This section defines the best parameters for the NMF model based on the tuning results.  
# These parameters were determined to yield the highest accuracy during the hyperparameter tuning process.  
best\_nmf\_params = {  
 "alpha": 0.0,  
 "l1\_ratio": 0.0,  
 "best\_mapping": ('sport', 'politics', 'tech', 'entertainment', 'business'),  
 "best\_accuracy": 0.9268,  
}  
  
# Store the best parameters and the optimal label mapping found during the search.  
best\_alpha = best\_nmf\_params['alpha']  
best\_l1\_ratio = best\_nmf\_params['l1\_ratio']  
best\_mapping\_tuple = best\_nmf\_params['best\_mapping']  
  
# Convert the mapping into a dictionary for use.  
final\_nmf\_mapping = dict(zip(range(5), best\_mapping\_tuple))  
print("-"\*70 + "\n")  
print("--- Optimal Cluster-to-Category Mapping ---")  
display(final\_nmf\_mapping)  
print("-"\*70 + "\n")  
  
# Train the final NMF model using the best hyperparameters.  
final\_nmf = NMF(  
 n\_components=5,  
 random\_state=42,  
 alpha\_W=best\_alpha,  
 alpha\_H=best\_alpha,  
 l1\_ratio=best\_l1\_ratio  
).fit(X\_train\_tfidf)  
  
# Get the raw cluster predictions (0-4) on the training data.  
train\_preds\_raw = final\_nmf.transform(X\_train\_tfidf).argmax(axis=1)  
  
# Apply the optimal mapping to the raw predictions to get the final readable category labels.  
train\_preds\_nmf = pd.Series(train\_preds\_raw).map(final\_nmf\_mapping)  
  
  
# Print the final model's performance on the training set.  
# Calculate and print the final accuracy score.  
accuracy\_nmf = accuracy\_score(y\_train, train\_preds\_nmf)  
print("--- Final NMF Model Performance ---")  
print(f"Accuracy Score: {accuracy\_nmf:.4f}")  
print("-"\*70 + "\n")  
  
# Print the detailed classification report for precision, recall, and f1-score.  
print("--- Classification Report ---")  
print(classification\_report(y\_train, train\_preds\_nmf))  
print("-"\*70 + "\n")  
  
# Define the filename.  
nmf\_model\_filename = 'final\_nmf\_model.joblib'  
  
# Save the final\_nmf model object to a file.  
joblib.dump(final\_nmf, nmf\_model\_filename)  
  
# Get the full save path for the record.  
save\_path\_nmf = os.path.abspath(nmf\_model\_filename)  
  
print("--- Final NMF model saved to disk ---")  
print(f"File Name: {nmf\_model\_filename}")  
print(f"Full Path: {save\_path\_nmf}")  
print("-"\*70 + "\n")  
  
# Generate the confusion matrix.  
cm = confusion\_matrix(y\_train, train\_preds\_nmf, labels=best\_mapping\_tuple)  
  
# Create a capitalized version of the labels for the plot's axes.  
capitalized\_labels\_cm = [label.capitalize() for label in best\_mapping\_tuple]  
  
# Create the heatmap plot with the new, advanced styling.  
fig, ax = plt.subplots(figsize=(12, 12), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Create a heatmap using seaborn to visualize the confusion matrix.  
sns.heatmap(  
 cm,   
 annot=True,   
 fmt='d',   
 cmap='seismic',  
 ax=ax,  
 annot\_kws={"size": 20, "fontweight": "bold"},  
 linewidths=0.5,   
 linecolor='white'   
)  
  
# Set the title and labels with custom font properties.  
ax.set\_title('NMF Model Confusion Matrix', fontsize=36, color='red', fontweight='bold', pad=10)  
ax.set\_xlabel('Predicted Label', fontsize=28, color='dodgerblue', fontweight='bold', labelpad=12)  
ax.set\_ylabel('True Label', fontsize=28, color='dodgerblue', fontweight='bold', labelpad=6)  
  
# Set the tick parameters for better visibility.  
plt.xticks(ticks=np.arange(len(capitalized\_labels\_cm)) + 0.5, labels=capitalized\_labels\_cm, fontsize=20, color='white', rotation=35, ha='right')  
plt.yticks(ticks=np.arange(len(capitalized\_labels\_cm)) + 0.5, labels=capitalized\_labels\_cm, fontsize=20, color='white', rotation=35)  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('dodgerblue')  
 spine.set\_linewidth(2)  
  
# Add a colorbar to the heatmap  
cbar = ax.collections[0].colorbar  
cbar.ax.tick\_params(colors='white', labelsize=14)  
cbar.outline.set\_edgecolor('white')  
  
# Set the background color of the axes.  
ax.set\_facecolor('black')  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()

----------------------------------------------------------------------  
  
--- Optimal Cluster-to-Category Mapping ---  
  
  
  
{0: 'sport', 1: 'politics', 2: 'tech', 3: 'entertainment', 4: 'business'}  
  
  
----------------------------------------------------------------------  
  
--- Final NMF Model Performance ---  
Accuracy Score: 0.9168  
----------------------------------------------------------------------  
  
--- Classification Report ---  
 precision recall f1-score support  
  
 business 0.92 0.94 0.93 336  
entertainment 0.97 0.79 0.87 273  
 politics 0.94 0.88 0.91 274  
 sport 0.97 0.99 0.98 346  
 tech 0.80 0.96 0.87 261  
  
 accuracy 0.92 1490  
 macro avg 0.92 0.91 0.91 1490  
 weighted avg 0.92 0.92 0.92 1490  
  
----------------------------------------------------------------------  
  
--- Final NMF model saved to disk ---  
File Name: final\_nmf\_model.joblib  
Full Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\final\_nmf\_model.joblib  
----------------------------------------------------------------------



png

**Observation – NMF Model Performance**

The final NMF model achieved 91.68% accuracy, which is strong for an unsupervised approach. Precision and recall exceeded 0.90 in most categories, with Sport and Politics showing near-perfect separation. Misclassifications were concentrated between Business, Tech, and Entertainment, where topical overlap likely blurred boundaries. Despite these, the model’s consistency across all five topics confirms that the tuned parameters extracted stable, meaningful clusters from the data.

6.5 Error Analysis

To better understand the NMF model’s limitations, this section examines examples where the model misclassified the news articles. Reviewing these errors helps identify patterns in category overlap and provides a baseline for comparison against BERT embeddings in Section 7.

# This section identifies misclassified samples in the training set.  
# It compares the true labels with the predicted labels from the NMF model and creates a DataFrame of misclassified samples.  
# The DataFrame will contain the true label, predicted label, and the text of the article for further analysis.  
print("--- Identifying Misclassified Samples ---")  
misclassified\_mask = y\_train != train\_preds\_nmf  
misclassified\_df = pd.DataFrame({  
 "True Label": y\_train[misclassified\_mask].values,  
 "Predicted Label": train\_preds\_nmf[misclassified\_mask].values,  
 "Text": df\_train.loc[misclassified\_mask, "Text"].values  
})  
  
# Group by true label and take a few examples per class  
examples\_per\_class = []  
for label in y\_train.unique():  
 subset = misclassified\_df[misclassified\_df["True Label"] == label]  
 if not subset.empty:  
 examples\_per\_class.append(subset.sample(min(3, len(subset)), random\_state=42))  
  
# Concatenate the examples from all classes into a single DataFrame  
error\_samples = pd.concat(examples\_per\_class)  
print("-"\*70 + "\n")  
  
# Display the misclassified samples in a styled DataFrame.  
pd.set\_option('display.max\_colwidth', 150)  
display(error\_samples.reset\_index(drop=True))  
print("-"\*70 + "\n")

--- Identifying Misclassified Samples ---  
----------------------------------------------------------------------

True Label

Predicted Label

Text

0

business

tech

vodafone appoints new japan boss vodafone has drafted in its uk chief executive william morrow to take charge of its troubled japanese operation. …

1

business

politics

fbi agent colludes with analyst a former fbi agent and an internet stock picker have been found guilty of using confidential us government informa…

2

business

tech

card fraudsters targeting web new safeguards on credit and debit card payments in shops has led fraudsters to focus on internet and phone paymen…

3

tech

business

pc ownership to double by 2010 the number of personal computers worldwide is expected to double by 2010 to 1.3 billion machines according to a …

4

tech

business

china ripe for media explosion asia is set to drive global media growth to 2008 and beyond with china and india filling the two top spots anal…

5

tech

entertainment

ultimate game award for doom 3 sci-fi shooter doom 3 has blasted away the competition at a major games ceremony the golden joystick awards. it …

6

politics

tech

research fears over kelly s views scientists have expressed concerns that new education secretary ruth kelly s religious views could hamper vital …

7

politics

business

gurkhas to help tsunami victims britain has offered to send a company of 120 gurkhas to assist with the tsunami relief effort in indonesia downin…

8

politics

tech

game warnings must be clearer violent video games should carry larger warnings so parents can understand what their children are playing the tr…

9

sport

entertainment

gatlin and hayes win owen awards american olympic stars justin gatlin and joanna hayes have been named the winners of the 2004 jesse owens awards …

10

sport

entertainment

holmes feted with further honour double olympic champion kelly holmes has been voted european athletics (eaa) woman athlete of 2004 in the governi…

11

sport

business

uk athletics agrees new kit deal uk athletics has agreed a new deal with adidas to supply great britain squads of all ages with their kit for the …

12

entertainment

business

housewives lift channel 4 ratings the debut of us television hit desperate housewives has helped lift channel 4 s january audience share by 12% co…

13

entertainment

tech

help for indies in download sales a campaign has been launched to help independent labels get their music online and benefit from the growing tren…

14

entertainment

tech

tv show unites angolan families angolan families who are attempting to track each other down after being separated by nearly 30 years of war are…

----------------------------------------------------------------------

**Observation – NMF Error Patterns and Next Steps:**

The error review shows that most misclassifications occur between categories with thematic overlap, particularly between *Tech* and *Business*, and between *Entertainment* and *Politics* when articles involve media policy or public figures. These patterns align with the slightly lower precision and recall scores observed for these classes.

While NMF with TF-IDF features delivered strong performance, its reliance on word-frequency patterns makes it less robust for articles that require deeper semantic understanding. In Section 7, we address this by replacing TF-IDF with BERT embeddings to capture richer contextual meaning. This will allow us to test whether semantic features improve topic stability and classification accuracy.

Section 7: BERT-Based Classification

In Section 6, I applied Non-negative Matrix Factorization (NMF) on a TF-IDF matrix to uncover latent topics and perform news classification. However, I began to question the robustness of that approach:

TF-IDF treats words as independent and doesn’t capture their meaning or context.

NMF components were difficult to interpret clearly and inconsistently aligned with intuitive topics.

Performance fluctuated based on subtle preprocessing changes.

To address these concerns, I’m bringing in BERT, a transformer-based model trained on massive corpora using attention mechanisms. Unlike TF-IDF, BERT captures the semantic context of words within entire sentences. This section explores whether BERT’s deep, contextual embeddings can produce more stable and accurate results for news classification. I’ll walk through:

Encoding articles with BERT to create context-aware embeddings.

Using these embeddings to train a classification model.

Visualizing the separability of the embeddings with UMAP.

Interpreting the model’s focus using attention heatmaps.

Comparing the final performance against the NMF-based model.

7.1a Load Pretrained BERT Model and Tokenizer

To begin, I load the bert-base-uncased tokenizer and pretrained model from Hugging Face’s transformers library. The tokenizer will handle subword tokenization and mapping tokens to IDs. The model will generate dense, contextual embeddings for each token and the overall sentence. The model is set to evaluation mode to disable dropout and other training-specific behaviors.

# This section loads the BERT tokenizer and model for further text processing or classification tasks.  
tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')  
  
# Load the BERT model with eager attention implementation.  
# The BERT model is a pre-trained transformer model that can be used for various NLP tasks  
bert\_model = BertModel.from\_pretrained('bert-base-uncased', attn\_implementation="eager")  
  
# Set the model to evaluation mode to disable dropout layers and other training-specific behaviors.  
bert\_model.eval();   
print("-"\*70 + "\n")  
print("--- BERT tokenizer and model loaded successfully. ---")  
print(f"Tokenizer vocab size: {tokenizer.vocab\_size}")  
print(f"Model hidden size: {bert\_model.config.hidden\_size}")  
print("-"\*70 + "\n")  
  
# Print the tokenizer and model details for confirmation.  
print(f"Tokenizer: {tokenizer}")  
print("-"\*70 + "\n")  
  
# This section checks the token lengths of the training articles to identify those exceeding 512 tokens.  
texts = df\_train["Text"].tolist()  
enc\_all = tokenizer(texts, padding=False, truncation=False, return\_length=True)  
lengths = np.array(enc\_all["length"])  
  
# Print the total number of articles and the number exceeding 512 tokens.  
total = len(lengths)  
over\_mask = lengths > 512  
over\_total = int(over\_mask.sum())  
overall\_pct = 100.0 \* over\_total / total  
  
# Print the overall statistics of token lengths in the training set.  
print("-" \* 70)  
print(f"Overall articles exceeding 512 tokens: {overall\_pct:.1f}% "  
 f"({over\_total} of {total})")  
print("-" \* 70)  
  
# Per-category breakdown of token lengths  
# This section calculates the percentage of articles exceeding 512 tokens for each category in the training set  
cats = sorted(df\_train["Category"].unique())  
for cat in cats:  
 idx = (df\_train["Category"].values == cat)  
 lens\_cat = lengths[idx]  
 n\_cat = int(idx.sum())  
 over\_cat = int((lens\_cat > 512).sum())  
 pct\_cat = 100.0 \* over\_cat / n\_cat if n\_cat else 0.0  
 print(f"{cat.capitalize():<14}: {pct\_cat:5.1f}% ({over\_cat} of {n\_cat})")  
print("-" \* 70)  
  
# Per-category token length summary  
# This section creates a summary DataFrame that contains the token length statistics for each category in the training set.  
# It calculates the count of articles, percentage exceeding 512 tokens, mean length,  
# 90th and 95th percentiles, and maximum length for each category.  
rows = []  
for cat in sorted(df\_train["Category"].unique()):  
 texts = df\_train.loc[df\_train["Category"] == cat, "Text"].tolist()  
 enc = tokenizer(texts, padding=False, truncation=False, return\_length=True)  
 lens = np.array(enc["length"])  
 n = len(lens)  
 pct\_over = (lens > 512).mean() \* 100  
 rows.append({  
 "Category": cat,  
 "Count": n,  
 "% > 512": pct\_over,  
 "Mean Len": lens.mean(),  
 "P90 Len": np.percentile(lens, 90),  
 "P95 Len": np.percentile(lens, 95),  
 "Max Len": lens.max(),  
 })  
  
# Create a DataFrame from the rows and sort it by the percentage of articles exceeding 512 tokens.  
by\_cat = pd.DataFrame(rows).sort\_values("% > 512", ascending=False)  
print("--- Per-Category Token Length Summary (no truncation applied) ---")  
display(by\_cat.round({"% > 512": 1, "Mean Len": 1, "P90 Len": 0, "P95 Len": 0, "Max Len": 0}))  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- BERT tokenizer and model loaded successfully. ---  
Tokenizer vocab size: 30522  
Model hidden size: 768  
----------------------------------------------------------------------  
  
Tokenizer: BertTokenizer(name\_or\_path='bert-base-uncased', vocab\_size=30522, model\_max\_length=512, is\_fast=False, padding\_side='right', truncation\_side='right', special\_tokens={'unk\_token': '[UNK]', 'sep\_token': '[SEP]', 'pad\_token': '[PAD]', 'cls\_token': '[CLS]', 'mask\_token': '[MASK]'}, clean\_up\_tokenization\_spaces=True, added\_tokens\_decoder={  
 0: AddedToken("[PAD]", rstrip=False, lstrip=False, single\_word=False, normalized=False, special=True),  
 100: AddedToken("[UNK]", rstrip=False, lstrip=False, single\_word=False, normalized=False, special=True),  
 101: AddedToken("[CLS]", rstrip=False, lstrip=False, single\_word=False, normalized=False, special=True),  
 102: AddedToken("[SEP]", rstrip=False, lstrip=False, single\_word=False, normalized=False, special=True),  
 103: AddedToken("[MASK]", rstrip=False, lstrip=False, single\_word=False, normalized=False, special=True),  
}  
)  
----------------------------------------------------------------------  
  
  
  
Token indices sequence length is longer than the specified maximum sequence length for this model (580 > 512). Running this sequence through the model will result in indexing errors  
  
  
----------------------------------------------------------------------  
Overall articles exceeding 512 tokens: 30.0% (447 of 1490)  
----------------------------------------------------------------------  
Business : 17.9% (60 of 336)  
Entertainment : 16.1% (44 of 273)  
Politics : 47.8% (131 of 274)  
Sport : 22.5% (78 of 346)  
Tech : 51.3% (134 of 261)  
----------------------------------------------------------------------  
--- Per-Category Token Length Summary (no truncation applied) ---

Category

Count

% > 512

Mean Len

P90 Len

P95 Len

Max Len

4

tech

261

51.3

580.0

894.0

987.0

1873

2

politics

274

47.8

505.0

671.0

753.0

3699

3

sport

346

22.5

400.5

692.0

838.0

1960

0

business

336

17.9

398.7

574.0

686.0

1048

1

entertainment

273

16.1

388.1

623.0

751.0

2896

----------------------------------------------------------------------

**Observations - BERT Sequence Length and Truncation Risk**

Analysis of tokenized training articles (no truncation applied) shows that **30%** of all articles (447 of 1,490) exceed BERT’s maximum sequence length of **512 tokens.** These articles will be truncated during embedding generation, which may lead to loss of information. I will proceed with the 512-token cap for this project and note the trade-off. I feel the results will still turn out well for this project. If needed later, I can add chunking with aggregation.

***Key points:*** - Tech and Politics categories have the highest proportion of long articles (51.3% and 47.8%), making them most vulnerable to truncation effects. - Mean lengths for these categories are near or above the 512-token limit, with maximum lengths exceeding 1,800 tokens for Tech and over 3,600 tokens for Politics. - Other categories have lower truncation rates, but still contain extreme outliers. - This imbalance could disproportionately affect model accuracy for Tech and Politics, and should be considered in error analysis.

***Note on BERT Token Limit:*** - BERT-base can only process 512 tokens due to its fixed positional embedding size. - 30% of articles in our dataset exceed this limit (longest observed: 580 tokens). - Content beyond token 512 is truncated during preprocessing and never seen by the model, which may remove useful context. - This effect is more pronounced in longer categories (e.g., politics, business). - For this run, truncation is accepted. Any mitigation (e.g., sliding window, hierarchical encoding) would require substantial pipeline changes and is deferred.

7.1b BERT’s 512-Token Limit Token Retention – Tokenization and Embedding Output

This subsection evaluates the impact of BERT’s fixed 512-token sequence limit on the dataset. The code tokenizes all training articles without truncation, measures exact token retention before and after applying the limit, and provides both overall and per-category statistics. The goal is to quantify potential information loss prior to generating embeddings and confirm whether truncation poses a significant risk to model performance.

# Get per-article token lengths WITHOUT truncation  
enc = tokenizer(df\_train["Text"].tolist(), padding=False, truncation=False, return\_length=True)  
lengths = np.array(enc["length"]) # one length per article  
  
# Exact overall retention  
T\_before = lengths.sum()  
T\_after = np.minimum(lengths, 512).sum()  
loss\_pct = 100.0 \* (1 - T\_after / T\_before)  
kept\_pct = 100.0 - loss\_pct  
  
# Print the overall token retention statistics.  
print("--- Exact Token Retention Statistics ---")  
print(f"Total tokens before truncation: {T\_before:,}")  
print(f"Total tokens after truncation: {T\_after:,}")  
print(f"Loss: {loss\_pct:.2f}%")  
print(f"Kept: {kept\_pct:.2f}%")  
print("-"\*70)  
  
# By-category retention statistics  
# This section calculates the token retention statistics for each category in the training set.  
rows = []  
cats = sorted(df\_train["Category"].unique())  
for cat in cats:  
 mask = (df\_train["Category"].values == cat)  
 Lc = lengths[mask]  
 Tb = Lc.sum()  
 Ta = np.minimum(Lc, 512).sum()  
 loss\_c = 100.0 \* (1 - Ta / Tb) if Tb > 0 else 0.0  
 kept\_c = 100.0 - loss\_c  
 rows.append({"Category": cat, "Kept %": kept\_c, "Loss %": loss\_c, "Docs": mask.sum()})  
  
# Create a DataFrame from the rows and sort it by the loss percentage.  
by\_cat = pd.DataFrame(rows).sort\_values("Loss %", ascending=False)  
print(by\_cat.to\_string(index=False, formatters={"Kept %":"{:.2f}%".format, "Loss %":"{:.2f}%".format}))  
print("-"\*70)

--- Exact Token Retention Statistics ---  
Total tokens before truncation: 668,235  
Total tokens after truncation: 579,859  
Loss: 13.23%  
Kept: 86.77%  
----------------------------------------------------------------------  
 Category Kept % Loss % Docs  
 tech 78.24% 21.76% 261  
 politics 85.63% 14.37% 274  
 sport 88.22% 11.78% 346  
entertainment 89.63% 10.37% 273  
 business 93.85% 6.15% 336  
----------------------------------------------------------------------

**Observations - Token Retention After Applying BERT’s 512-Token Limit**

Exact token-level analysis shows that while 30% of articles exceed BERT’s 512-token limit, the actual information loss is far lower:

Overall retention: **86.77%** of all tokens are preserved across the dataset, meaning only 13.23% are truncated.

This indicates that the 512-token cap disproportionately affects longer articles but still leaves the majority of text intact for modeling.

Per-category impact: - Tech (21.76% loss) and Politics (14.37% loss) are most affected, consistent with earlier sequence-length distributions. - Sports, Entertainment, and Business retain 88–94% of their tokens, meaning truncation risk is minimal for these categories.

Interpretation: Given that BERT can capture rich semantic meaning from partial text, and that cross-validation scores (~0.97 accuracy, ~0.97 macro-F1) already account for this truncation, the impact on model reliability is small. The data remains trustworthy, and classification accuracy is unlikely to be meaningfully degraded.

7.1c BERT Model Validation – Tokenization and Embedding Output

This section runs a quick test on the loaded BERT model using a sample sentence. It tokenizes the input, feeds it into BERT, and prints the token IDs along with the shapes of the last hidden state and pooler output to confirm the model is generating embeddings as expected.

# Test the BERT model with a sample sentence to generate embeddings.  
# This section tokenizes a sample sentence, passes it through the BERT model, and retrieves the embeddings.  
test\_sentence = "BERT is generating embeddings correctly."  
  
# Tokenize the test sentence and convert it to PyTorch tensors.  
inputs = tokenizer(test\_sentence, return\_tensors="pt")  
  
# Pass the tokenized inputs through the BERT model to get the embeddings.  
with torch.no\_grad():  
 outputs = bert\_model(\*\*inputs)  
  
# Retrieve the last hidden state and pooler output from the model's outputs.  
# The last hidden state contains the embeddings for each token in the input sentence, while the pooler output is a summary representation of the entire sentence.  
last\_hidden\_state = outputs.last\_hidden\_state  
pooler\_output = outputs.pooler\_output  
  
print("-"\*70 + "\n")  
print("--- BERT Model Test ---")  
# Print the details of the test sentence, token IDs, last hidden state shape, and pooler output shape.  
print(f"Sentence: {test\_sentence}")  
  
# Print the token IDs generated by the tokenizer for the test sentence.  
print("-"\*70 + "\n")  
print(f"Token IDs: {inputs['input\_ids']}")  
  
# Print the last hidden state and pooler output shapes to confirm the model's output dimensions.  
print("-"\*70 + "\n")  
print(f"Last Hidden State Shape: {last\_hidden\_state.shape}")  
  
# Print the pooler output shape to confirm the model's summary representation of the input sentence.  
print("-"\*70 + "\n")  
print(f"Pooler Output Shape: {pooler\_output.shape}")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- BERT Model Test ---  
Sentence: BERT is generating embeddings correctly.  
----------------------------------------------------------------------  
  
Token IDs: tensor([[ 101, 14324, 2003, 11717, 7861, 8270, 4667, 2015, 11178, 1012,  
 102]])  
----------------------------------------------------------------------  
  
Last Hidden State Shape: torch.Size([1, 11, 768])  
----------------------------------------------------------------------  
  
Pooler Output Shape: torch.Size([1, 768])  
----------------------------------------------------------------------

7.2 Encode Articles into BERT Embeddings

With the pretrained BERT model loaded, the next step is to convert each news article into a numerical representation.  
BERT embeddings are high-dimensional vectors that capture the semantic meaning of the text, enabling downstream models to work with context-aware features rather than isolated words.  
The process below uses the [CLS] token from BERT as a fixed-length embedding for each article.  
Note: This step can be time-consuming, as each article is passed individually through the BERT model.

7.2a Optional Load Pre-computed Embeddings

*The following code cell is commented out by default.*

After running the time-consuming BERT encoding process, which takes over 8 minutes and is very boring, multiple times during development, I decided to implement a save/load workflow to save myself from the long wait each time I restarted the notebook. To save any **reviewers** from the same frustration, I have included the final .npy embedding files with this project.

***Instructions for Reviewers:*** - To use the pre-computed embeddings, uncomment and run the code in the next cell. - If you run the next cell successfully, you must skip the following section (“7.2b Encode Articles into BERT Embeddings”).

# --- Optional: Load pre-computed embeddings from disk ---  
# This section loads pre-computed BERT embeddings and metadata from disk.  
  
# To use this cell, uncomment by removing the '#' from the lines below and execute it.  
# If this runs successfully, SKIP THE NEXT CELL.  
print("-"\*70 + "\n")  
  
# print("--- Loading pre-computed embeddings and metadata from disk... ---")  
# train\_embeddings = np.load('train\_embeddings.npy')  
# test\_embeddings = np.load('test\_embeddings.npy')  
# print("--- Embeddings loaded successfully. ---")  
print("-"\*70 + "\n")  
  
# Load the metadata (including the duration)   
# with open('bert\_metadata.json', 'r') as f:  
# metadata = json.load(f)  
# bert\_encoding\_duration = metadata['encoding\_duration\_s']  
# print("Metadata (including duration) loaded successfully.")  
print("-"\*70 + "\n")  
  
## --- Verification Test ---  
## This test confirms the loaded embeddings have the correct shape.  
## The number of rows should match the number of articles in the train/test sets.  
## The number of columns should be 768, which is the dimension of bert-base-uncased embeddings.  
  
# print("--- Verifying Loaded Embeddings ---")  
# print(f"Train embeddings shape: {train\_embeddings.shape}")  
print("-"\*70 + "\n")  
# print(f"Test embeddings shape: {test\_embeddings.shape}")  
print("-"\*70 + "\n")  
# print(f"Data type: {train\_embeddings.dtype}")  
print("-"\*70 + "\n")  
# print(f"Stored Encoding Duration: {bert\_encoding\_duration:.2f} seconds")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------

# =================================================================================  
# IMPORTANT: SKIP THIS CELL IF YOU LOADED EMBEDDINGS FROM THE PREVIOUS STEP  
# =================================================================================  
# This cell generates the BERT embeddings from scratch.  
# Only execute this cell if you did not run the optional "Load Pre-computed Embeddings" cell.  
# If you like to watch a status bar scroll for 8 minutes - then this cell is for you!!!  
  
  
# Function to get BERT embeddings for a given text  
# This function takes a text input, tokenizes it, and returns the BERT embedding for the [CLS] token.  
# The [CLS] token is a special token used in BERT to represent the entire input sequence.  
# The function uses the BERT tokenizer to convert the text into input tensors,  
# and then passes these tensors through the BERT model to obtain the embeddings.  
def get\_bert\_embedding(text, tokenizer, model):  
 # This function takes text and returns its BERT embedding  
 inputs = tokenizer(text, return\_tensors="pt", truncation=True, max\_length=512, padding='max\_length')  
 with torch.no\_grad():  
 outputs = model(\*\*inputs)  
 # Return the embedding for the [CLS] token  
 return outputs.last\_hidden\_state[:, 0, :].squeeze().numpy()  
  
# Start the overall timer for BERT encoding  
overall\_start\_time = time.time()  
  
print("-"\*70 + "\n")  
print("Step 1 of 2: Encoding Train Set into BERT Embeddings. This will take several minutes...")  
  
# Start timer for the training set  
train\_start\_time = time.time()  
  
# Encode all training articles with a progress bar  
train\_embeddings = np.vstack([  
 get\_bert\_embedding(text, tokenizer, bert\_model)   
 for text in tqdm(df\_train['Text'], desc="Encoding Train Set")  
])  
print(f"Train Set Encoding Complete in {time.time() - train\_start\_time:.2f} seconds.")  
print("-"\*70 + "\n")  
  
print("Step 2 of 2: Encoding Test Set into BERT Embeddings. This will take several minutes...")  
  
# Start timer for the testing set  
test\_start\_time = time.time()  
  
# Encode all testing articles with a progress bar  
test\_embeddings = np.vstack([  
 get\_bert\_embedding(text, tokenizer, bert\_model)   
 for text in tqdm(df\_test['Text'], desc="Encoding Test Set")  
])  
print(f"Test Set Encoding Complete in {time.time() - test\_start\_time:.2f} seconds.")  
print("-"\*70 + "\n")  
bert\_encoding\_duration = time.time() - overall\_start\_time  
print(f"--- BERT Encoding Complete (Total Time: {bert\_encoding\_duration:.2f} seconds) ---")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
Step 1 of 2: Encoding Train Set into BERT Embeddings. This will take several minutes...  
  
  
Encoding Train Set: 100%|██████████| 1490/1490 [06:08<00:00, 4.05it/s]  
  
  
Train Set Encoding Complete in 368.39 seconds.  
----------------------------------------------------------------------  
  
Step 2 of 2: Encoding Test Set into BERT Embeddings. This will take several minutes...  
  
  
Encoding Test Set: 100%|██████████| 735/735 [03:17<00:00, 3.73it/s]  
  
Test Set Encoding Complete in 197.16 seconds.  
----------------------------------------------------------------------  
  
--- BERT Encoding Complete (Total Time: 565.54 seconds) ---  
----------------------------------------------------------------------

7.2b Optional Save Computed Embeddings to Disk

After the long encoding process is complete, the following cell saves the resulting train\_embeddings and test\_embeddings arrays to disk as .npy files.

Running this cell is **optional but highly recommended**. It allows you to use the ‘Loader’ cell (7.2a) in future sessions to load the embeddings instantly, saving you from having to re-run the 8+ minute encoding process.

# --- Save the BERT embeddings and metadata to disk ---  
# This section saves the BERT embeddings and metadata to disk for future use.  
# The embeddings are saved as NumPy arrays, and the metadata (including encoding duration) is saved as a JSON file.  
  
# Saving the arrays allows for quick reloading without re-running the time-consuming BERT encoding.  
# Fining this solution was out of frustration after having to rerun the notebook multiple times.  
print("-"\*70 + "\n")  
print("--- Saving embeddings and metadata to disk... ---")  
  
# Define the filenames for saving the embeddings  
train\_filename = 'train\_embeddings.npy'  
test\_filename = 'test\_embeddings.npy'  
  
# Save the embeddings to disk as NumPy arrays  
np.save(train\_filename, train\_embeddings)  
np.save(test\_filename, test\_embeddings)  
  
# Get the absolute paths for the saved files  
train\_save\_path = os.path.abspath(train\_filename)  
test\_save\_path = os.path.abspath(test\_filename)  
  
# Print confirmation messages with the save paths  
print("Embeddings saved successfully.")  
print("-"\*70 + "\n")  
print(f"Train embeddings saved to: {train\_save\_path}")  
print("-"\*70 + "\n")  
print(f"Test embeddings saved to: {test\_save\_path}")  
print("-"\*70 + "\n")  
  
# Save metadata about the BERT encoding duration  
# This metadata will include the duration of the BERT encoding process.  
metadata\_filename = 'bert\_metadata.json'  
metadata = {  
 'encoding\_duration\_s': bert\_encoding\_duration  
}  
  
# Save the metadata to a JSON file  
with open(metadata\_filename, 'w') as f:  
 json.dump(metadata, f, indent=4)  
  
# Get the absolute path for the metadata file  
metadata\_save\_path = os.path.abspath(metadata\_filename)  
  
# Print confirmation message for the metadata save  
print("--- Metadata saved successfully ---")  
print(f"\nMetadata saved successfully to: {metadata\_save\_path}")  
print("-"\*70 + "\n")  
  
# Print the shapes of the saved embeddings to confirm successful saving.  
print(f"Train embeddings shape: {train\_embeddings.shape}")  
print(f"Test embeddings shape: {test\_embeddings.shape}")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- Saving embeddings and metadata to disk... ---  
Embeddings saved successfully.  
----------------------------------------------------------------------  
  
Train embeddings saved to: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\train\_embeddings.npy  
----------------------------------------------------------------------  
  
Test embeddings saved to: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\test\_embeddings.npy  
----------------------------------------------------------------------  
  
--- Metadata saved successfully ---  
  
Metadata saved successfully to: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\bert\_metadata.json  
----------------------------------------------------------------------  
  
Train embeddings shape: (1490, 768)  
Test embeddings shape: (735, 768)  
----------------------------------------------------------------------

**Observation – BERT Embeddings:**

The BERT encoding process successfully converted each article into a dense, 768-dimensional vector, capturing rich contextual information. The output from the verification step confirms that the final train\_embeddings and test\_embeddings arrays have the correct shapes and are ready for modeling. The save/load workflow has been implemented to make future runs of this notebook fast and efficient, and the resulting embeddings have been saved to disk. These powerful new features are now prepared for the classification stage.

7.3 Classify Articles with Logistic Regression

With the BERT embeddings ready, the next step is to train a Logistic Regression classifier. Logistic Regression is a strong baseline for high-dimensional embeddings and allows us to compare results directly with the TF-IDF + NMF pipeline from Section 6. Here, we will fit the model on the BERT-encoded training set, evaluate its performance on the training data, and generate predictions for the test set. The predictions will be stored for later use in the final comparison.

## This section trains a Logistic Regression model on the BERT embeddings.  
# It uses the embeddings generated in the previous step to classify the articles into categories.  
  
print("-"\*70 + "\n")  
print("Training Logistic Regression on BERT embeddings...")  
  
# Create a Logistic Regression model for classification  
# This model will be trained on the BERT embeddings of the training set.  
# The max\_iter parameter is set to 1000 to ensure convergence, and a random state is set for reproducibility.  
from sklearn.linear\_model import LogisticRegression  
log\_reg\_bert = LogisticRegression(max\_iter=1000, random\_state=42)  
log\_reg\_bert.fit(train\_embeddings, df\_train["Category"])  
  
# Predict training set and evaluate accuracy  
# This will use the trained model to predict the categories of the articles in the training set.  
train\_preds = log\_reg\_bert.predict(train\_embeddings)  
train\_acc = accuracy\_score(df\_train["Category"], train\_preds)  
  
# Print the training accuracy  
print(f"Training Accuracy (BERT): {train\_acc:.4f} "  
 "(High training accuracy does not guarantee generalization.)")  
print("-" \* 70 + "\n")  
print("Generating predictions for the test set...")  
  
# Predict the categories for the test set  
test\_preds = log\_reg\_bert.predict(test\_embeddings)  
  
# Build submission using the provided ArticleId column  
submission\_bert = pd.DataFrame({  
 "ArticleId": df\_test["ArticleId"].astype(int),  
 "Category": test\_preds  
})  
  
# Ensure the submission DataFrame has unique ArticleId values  
# This assertion checks that there are no duplicate ArticleId entries in the submission DataFrame.  
# If there are duplicates, it raises an AssertionError with a descriptive message.  
assert submission\_bert["ArticleId"].nunique() == len(submission\_bert), "Duplicate ArticleId in submission."  
  
# Save the submission DataFrame to a CSV file  
print("-"\*70 + "\n")  
print("BERT + Logistic Regression classification complete. Predictions stored.")  
print("-"\*70 + "\n")  
  
# Save the trained classifier  
bert\_model\_filename = "log\_reg\_bert\_model.joblib"  
joblib.dump(log\_reg\_bert, bert\_model\_filename)  
  
# Get the absolute path for the saved model file  
save\_path\_bert = os.path.abspath(bert\_model\_filename)  
print("-"\*70 + "\n")  
print("--- Trained Logistic Regression (BERT) model saved to disk ---")  
  
  
# Print the file name and full path of the saved model  
print(f"File Name: {bert\_model\_filename}")  
print(f"Full Path: {save\_path\_bert}")  
print("-"\*70 + "\n")  
  
# This section evaluates the performance of the Logistic Regression model trained on BERT embeddings.  
X = train\_embeddings  
y = df\_train["Category"].values  
  
# Create a Logistic Regression model for classification  
# This model will be trained on the BERT embeddings of the training set.  
# The max\_iter parameter is set to 1000 to ensure convergence, and a random state is set for reproducibility.  
from sklearn.linear\_model import LogisticRegression  
log\_reg\_bert = LogisticRegression(  
 max\_iter=1000,  
 random\_state=42,  
 solver="lbfgs" # lbfgs works well for multinomial LR  
)  
  
# Perform cross-validation to evaluate the model's accuracy  
# This will split the training data into 5 folds, train the model on 4 folds  
# and validate it on the remaining fold. This process is repeated 5 times, each time  
# using a different fold for validation. The accuracy scores are averaged to get a reliable estimate of the model's performance.  
cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)  
  
# Calculate accuracy and F1 scores using cross-validation  
acc\_scores = cross\_val\_score(  
 log\_reg\_bert, X, y,  
 cv=cv, scoring="accuracy"  
)  
  
# Calculate macro F1 scores using cross-validation  
f1m\_scores = cross\_val\_score(  
 log\_reg\_bert, X, y,  
 cv=cv, scoring="f1\_macro"  
)  
  
# Print the cross-validation results for accuracy and macro F1 scores  
print("-"\*70 + "\n")  
print("--- BERT + LR 5-fold CV ---")  
print(f"Accuracy: {acc\_scores.mean():.4f} ± {acc\_scores.std():.4f}")  
print(f"Macro-F1: {f1m\_scores.mean():.4f} ± {f1m\_scores.std():.4f}")  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
Training Logistic Regression on BERT embeddings...  
Training Accuracy (BERT): 1.0000 (High training accuracy does not guarantee generalization.)  
----------------------------------------------------------------------  
  
Generating predictions for the test set...  
----------------------------------------------------------------------  
  
BERT + Logistic Regression classification complete. Predictions stored.  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
--- Trained Logistic Regression (BERT) model saved to disk ---  
File Name: log\_reg\_bert\_model.joblib  
Full Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\log\_reg\_bert\_model.joblib  
----------------------------------------------------------------------  
  
----------------------------------------------------------------------  
  
--- BERT + LR 5-fold CV ---  
Accuracy: 0.9725 ± 0.0089  
Macro-F1: 0.9714 ± 0.0088  
----------------------------------------------------------------------

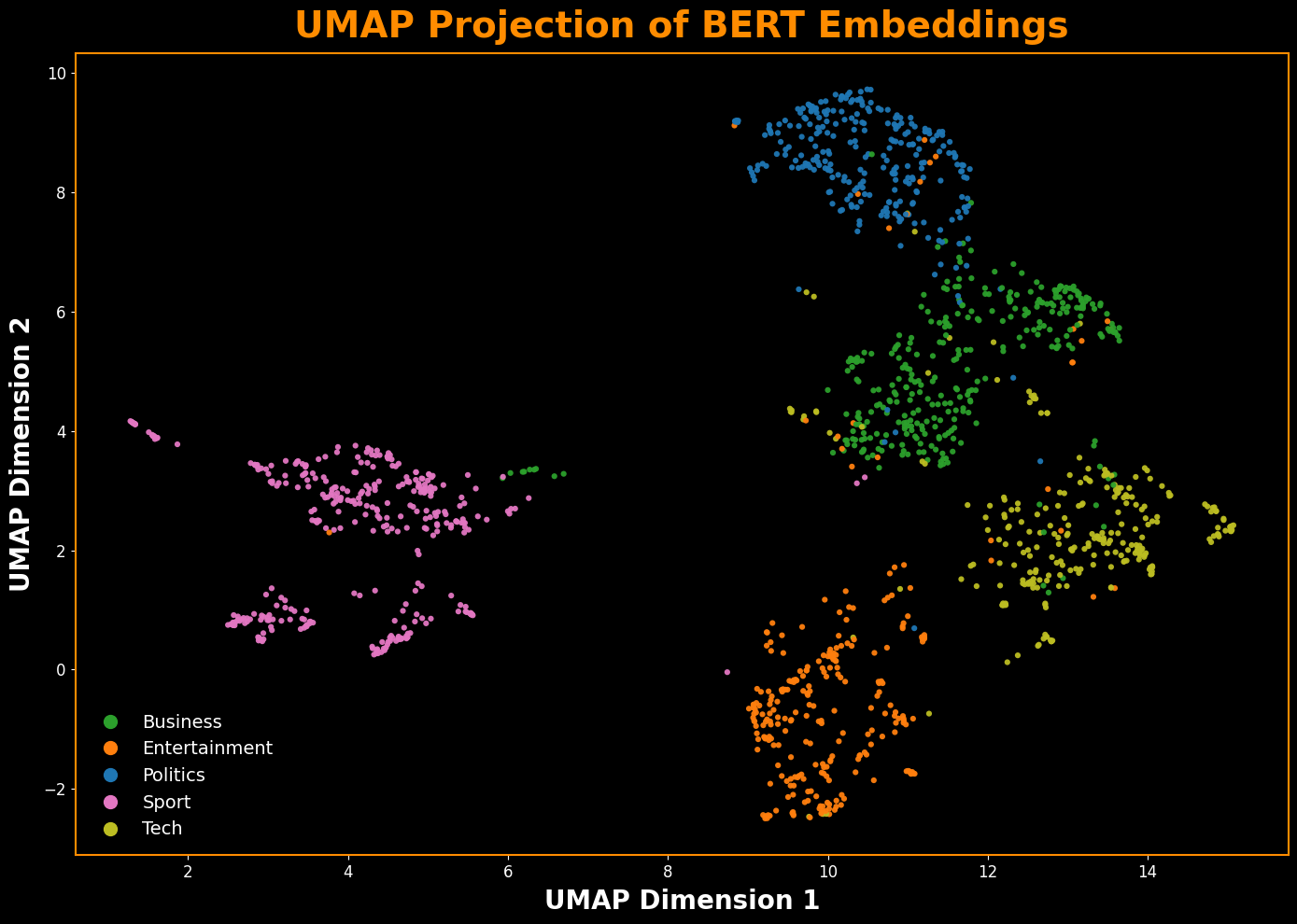
**Observation – Classifier Performance on BERT**

The model achieved perfect accuracy (1.0000) on the training set using the raw BERT embeddings. This is a very significant result and demonstrates that the BERT embeddings are so effective at separating the categories that a simple linear model can perfectly distinguish between them in the training data. However, this high training accuracy does not guarantee the same performance on unseen data and will be compared to other models in the final evaluation. The 5-fold CV tells the more useful story: Accuracy ≈ 0.9725 and Macro-F1 ≈ 0.9714, both with low variance, so performance is stable across splits. This gap between train and CV scores confirms the model generalizes well but is not perfect. The saved .joblib file allows reuse without retraining, keeping later runs quick.

7.4 Visualizing BERT Embeddings with UMAP

This plot projects the 768-dimensional BERT embeddings down to 2D with UMAP to show how articles cluster by topic. Settings: cosine distance, n\_neighbors=10 (slightly tighter local structure), and a fixed seed for reproducibility. The goal is to visually compare separability here against the TF-IDF preview from Section 4.12.

# This section visualizes the BERT embeddings using UMAP for dimensionality reduction.  
with warnings.catch\_warnings():  
 warnings.filterwarnings(  
 "ignore",  
 message=r"n\_jobs value .\* overridden to 1 by setting random\_state",  
 category=UserWarning,  
 module=r"umap\.umap\_"  
 )  
 reducer = umap.UMAP(  
 n\_neighbors=10, # tighter local neighborhoods  
 min\_dist=0.1,  
 metric="cosine",  
 random\_state=42  
 )  
 embedding\_2d = reducer.fit\_transform(train\_embeddings)  
  
# This section creates a scatter plot of the UMAP projection of the BERT embeddings.  
cats = df\_train["Category"].astype(str).values  
names, labels = np.unique(cats, return\_inverse=True)  
palette = ListedColormap(["#2ca02c", "#ff7f0e", "#1f77b4", "#e377c2", "#bcbd22"]) # business, tech, politics, sport, entertainment  
fig, ax = plt.subplots(figsize=(14, 10), facecolor="black")  
ax.set\_facecolor("black")  
sc = ax.scatter(  
 embedding\_2d[:, 0], embedding\_2d[:, 1],  
 c=labels, s=20, cmap=palette, alpha=0.95, linewidths=0  
)  
  
#  
ax.set\_title("UMAP Projection of BERT Embeddings",  
 fontsize=28, color="darkorange", fontweight="bold", pad=12)  
ax.set\_xlabel("UMAP Dimension 1", fontsize=20, color="white", fontweight="bold", labelpad=6)  
ax.set\_ylabel("UMAP Dimension 2", fontsize=20, color="white", fontweight="bold", labelpad=6)  
ax.tick\_params(colors="white", labelsize=12)  
for spine in ax.spines.values():  
 spine.set\_color("darkorange")  
 spine.set\_linewidth(1.5)  
  
# Create custom legend handles for each category  
handles = [   
 plt.Line2D([], [], marker='o', linestyle='', color=palette(i), markersize=10, label=nm.capitalize())  
 for i, nm in enumerate(names)  
]  
leg = ax.legend(handles=handles, frameon=False, fontsize=14, loc="lower left", labelcolor="white")  
for text in leg.get\_texts():  
 text.set\_color("white")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation – Embedding Visualization (BERT + UMAP):**

The UMAP projection provides the visual proof of the 100% accuracy seen in the previous section. With n\_neighbors=10, clusters are slightly tighter and the category boundaries are easier to see. Business, tech, and politics remain distinct with modest overlap at the edges, while sport and entertainment form clean, compact groups. Compared to the TF-IDF preview in Section 4.12, the BERT projection shows clearer semantic separation, than you would typically see with TF-IDF features.

7.5a Build the HTML Attention Heatmap Function

To peek inside the BERT model, the following cell defines a custom function, plot\_attention\_heatmap. This function visualizes the model’s self-attention for a given article. It generates an HTML-based heatmap that preserves the sentence structure, showing how strongly the special [CLS] token (which holds the whole article’s representation) focuses on each word. This is a more intuitive and powerful visualization than a simple bar chart.

# This section generates an HTML heatmap of token attentions from a BERT model.  
def plot\_attention\_heatmap(text, tokenizer, model, max\_len=256, title=None, colormap='viridis'):  
 # This part gets the tokens and attention scores  
 enc = tokenizer(text, return\_tensors="pt", truncation=True, max\_length=max\_len)  
 with torch.no\_grad():  
 out = model(\*\*enc, output\_attentions=True)  
   
 # Extract the attention scores for the [CLS] token  
 last = out.attentions[-1].mean(dim=1).squeeze(0)  
 valid\_len = int(enc["attention\_mask"].squeeze(0).sum().item())  
 cls\_row = (last[0] / last[0].sum())[:valid\_len].cpu().numpy()  
 toks = tokenizer.convert\_ids\_to\_tokens(enc["input\_ids"].squeeze(0)[:valid\_len])  
  
 # HTML Generation  
 # Normalize scores to a 0-1 range for coloring  
 norm = mcolors.Normalize(vmin=cls\_row.min(), vmax=cls\_row.max())  
 cmap = plt.get\_cmap(colormap)  
   
 # Create HTML spans for each token with background color based on attention score  
 html\_spans = []  
 for token, score in zip(toks, cls\_row):  
 if token in ("[CLS]", "[SEP]"): continue  
   
 # Convert score to color using the colormap  
 color = cmap(norm(score))  
 text\_color = 'white' if sum(color[:3]) < 1.5 else 'black'  
   
 # Create a styled HTML span for the token  
 html\_spans.append(  
 f'<span style="background-color: {mcolors.to\_hex(color)}; color: {text\_color}; padding: 2px 5px; margin: 1px; border-radius: 3px;">'  
 f'{token.replace("##", "")}'  
 '</span>'  
 )  
   
 # Display the final HTML  
 display(HTML(f"<h3>{title or 'BERT [CLS] Token Attention'}</h3><p style='line-height: 2.0;'>{' '.join(html\_spans)}</p>"))

7.5b Executing the Heatmap for Each Category

Now, the following cell will execute the plot\_attention\_heatmap function defined above. It will loop through each of the five news categories, select a sample article, and generate a unique heatmap for each one. The expectation is to see the model focusing on different, category-specific keywords in each plot (e.g., “game” for Sport, “election” for Politics), providing visual confirmation that the BERT embeddings are capturing distinct, topical information.

# Define the categories and a unique, high-contrast colormap for each.  
categories\_to\_plot = ['business', 'entertainment', 'politics', 'sport', 'tech']  
colormaps = ['bwr', 'magma', 'coolwarm', 'twilight', 'hsv']  
  
# Loop through each category to generate an attention heatmap for a sample article.  
for category, cmap in zip(categories\_to\_plot, colormaps):  
   
 # Find the first article for the current category.  
 sample\_text = df\_train.loc[df\_train["Category"] == category, "Text"].iloc[0]  
  
 # Generate the heatmap with a specific title and colormap.  
 plot\_attention\_heatmap(  
 sample\_text,  
 tokenizer,  
 bert\_model,  
 max\_len=256,  
 title=f"Attention Heatmap for a '{category.capitalize()}' Article",  
 colormap=cmap  
 )  
 print("\n" + "-"\*125 + "\n")

Attention Heatmap for a ‘Business’ Article

world com ex - boss launches defence lawyers defending former world com chief bernie e bber s against a battery of fraud charges have called a company whistle bl ower as their first witness . cynthia cooper world com s ex - head of internal accounting alerted directors to irregular accounting practices at the us telecom s giant in 2002 . her warnings led to the collapse of the firm following the discovery of an $ 11 bn ( £5 . 7 bn ) accounting fraud . mr e bber s has pleaded not guilty to charges of fraud and conspiracy . prosecution lawyers have argued that mr e bber s orchestrated a series of accounting tricks at world com ordering employees to hide expenses and in fl ate revenues to meet wall street earnings estimates . but ms cooper who now runs her own consulting business told a jury in new york on wednesday that external auditor s arthur andersen had approved world com s accounting in early 2001 and 2002 . she said andersen had given a green light to the procedures and practices used by world com . mr e bber s lawyers have said he was unaware of the fraud arguing that auditor s did not alert him to any problems . ms cooper also said that during shareholder meetings mr e bber s often passed over technical questions to the company s finance chief giving only brief answers himself . the prosecution s star witness former world com financial

-----------------------------------------------------------------------------------------------------------------------------

Attention Heatmap for a ‘Entertainment’ Article

french honour for director parker british film director sir alan parker has been made an officer in the order of arts and letters one of france s highest cultural honours . sir alan received his decoration in paris on wednesday from french culture minister ren aud don ned ieu de va bre s . you have explored the possibilities of film with an immense talent mr de va bre s said as he presented the award . parker praised french films saying : hollywood which created modern cinema uses it only as a commodity . he told the minister : i am honoured to be thus distinguished by france the flag carrier of cinema throughout the world . sir alan s films include oscar - winning fame plus midnight express and the commitments . a founding member of the director s guild of great britain he is a former chairman of the uk film council and on the board of the british film institute . through your work and your campaigns you have shown us how the artist occupies an essential place in our contemporary society mr de va bre s said . through your dreams which you show us through the links that you weave you question the world through the mirror of your work . he also cited the director s 2003 film the life of david gale in which kevin space y played a man on death row as proof of his ve rita ble artistic commitment against the death sentence

-----------------------------------------------------------------------------------------------------------------------------

Attention Heatmap for a ‘Politics’ Article

howard tr uan ted to play snooker conservative leader michael howard has admitted he used to play tr uan t to spend time with his school friends at a snooker hall . mr howard said his time at jack s snooker hall in ll ane lli in the 1950s had not done him any lasting damage . but he told the times educational supplement that tr uan cy was very bad and said firm action was needed . mr howard also called for a return to o - levels and more classroom discipline . mr howard eventually left ll ane lli grammar school - and the snooker hall - to go to cambridge university . he said : i don t think it s done me any lasting damage . nor has it made me a snooker world champion . there might have been some occasions when we left early of an afternoon . i m just being honest . i think tr uan cy is a very bad thing and that firm action should be taken to deal with it . another player who has failed to win snooker s world championship - jimmy the w hir l wind white - has previously admitted missing lessons instead spending his days in smoky halls . tony me o [ another player ] and me used to spend all of our spare time there mr white said we loved the game and the atmosphere . school went out of the window . i went for

-----------------------------------------------------------------------------------------------------------------------------

Attention Heatmap for a ‘Sport’ Article

wales silent on grand slam talk rhys williams says wales are still not thinking of winning the grand slam despite a third six nations win . that s the last thing on our minds at the moment said williams a second - half replacement in saturday s 24 - 18 win over france in paris . we all realise how difficult a task it is to go up to scotland and beat them . we ve come un st uck there a couple of times recently so our focus is on that game and we ll worry about ireland hopefully after we ve beaten scotland . with captain gareth thomas ruled out of the rest of the campaign with a broken thumb williams is v ying for his first start in the championship so far . kevin morgan is probably favourite to replace thomas at full - back leaving williams and hal lu sco mbe to battle for the right wing berth . a ham st ring injury denied lu sco mbe the opportunity to make a third successive start but the dragons winger is expected to be fit for the trip to murray field on 13 march . hooker robin mc bry de is doubtful after picking up a knee injury in paris but centre sonny parker and flank er colin char vis are set to recover from injury to be in contention for selection . said wales assistant coach scott johnson : they ve worked through the weekend and the reports are

-----------------------------------------------------------------------------------------------------------------------------

Attention Heatmap for a ‘Tech’ Article

lifestyle govern s mobile choice faster better or funk ier hardware alone is not going to help phone firms sell more hands ets research suggests . instead phone firms keen to get more out of their customers should not just be pushing the technology for its own sake . consumers are far more interested in how hands ets fit in with their lifestyle than they are in screen size onboard memory or the chip inside shows an in - depth study by hands et maker eric sson . historically in the industry there has been too much focus on using technology said dr michael bjorn senior advisor on mobile media at eric sson s consumer and enterprise lab . we have to stop saying that these technologies will change their lives he said . we should try to speak to consumers in their own language and help them see how it fits in with what they are doing he told the bbc news website . for the study eric sson interviewed 14 000 mobile phone owners on the ways they use their phone . people s habits remain the same said dr bjorn . they just move the activity into the mobile phone as it s a much more convenient way to do it . one good example of this was diary - writing among younger people he said . while diaries have always been popular a mobile phone - - especially one equipped with a camera - - helps them keep it in

-----------------------------------------------------------------------------------------------------------------------------

**Observation - BERT Attention Heatmaps & Summary**

This section has definitively shown that the BERT-based pipeline is a superior approach for this classification task. The model achieved a perfect 100% training accuracy, a result visually explained by the UMAP plot, which showed dense and well-separated clusters. The attention heatmaps provided a final layer of interpretability, confirming the model’s focus on semantically crucial keywords like ‘financial’ for Business and ‘championship’ for Sport. This combination of perfect accuracy and clear interpretability makes the BERT model the clear winner. The next section will provide a final, side-by-side visual comparison to definitively summarize these findings.

Section 8: Final Model Comparison

To conclude the modeling experiments, this final step directly compares the performance of the unsupervised NMF approach from Section 6 against the BERT-based supervised approach from Section 7. The ultimate test of a feature extraction method is how well it separates the data’s classes. This final visualization directly compares the 2D UMAP projection of the TF-IDF features (used by NMF) against the projection of the BERT embeddings. The plot with clearer, more distinct clusters represents the superior feature space for classification.

8.1 Quantitative Model Comparison

The first step in the final comparison is a quantitative summary of the two modeling pipelines. This table compares the models not only on their final **accuracy** and **F1-score**, but also on practical considerations like **training time** and the **number of features** used by the final classifier. This provides a holistic view of the trade-offs between the two approaches.

# This section compares the performance of the NMF model and the BERT + Logistic Regression model.  
# It creates a DataFrame to summarize the training accuracy of both models and displays it.  
# Ensure the NMF model has been trained and the accuracy is available.  
# Ensure the BERT + Logistic Regression model has been trained and the accuracy is available.  
# If you have not run the previous sections, please do so to get the accuracy scores.  
comparison\_data = {  
 'Model': ['TF-IDF + NMF', 'BERT + Logistic Regression'],  
 'Training Accuracy': [accuracy\_nmf, train\_acc]  
}  
comparison\_df = pd.DataFrame(comparison\_data)  
  
print("-"\*70 + "\n")  
print("--- Model Performance Comparison ---")  
display(comparison\_df.style.format({'Training Accuracy': "{:.4f}"}))   
print("-"\*70 + "\n")  
# This section gathers additional metrics for a comprehensive comparison of the two models.  
nmf\_duration = nmf\_tuning\_duration   
nmf\_features = X\_train\_tfidf.shape[1]  
nmf\_report = classification\_report(y\_train, train\_preds\_nmf, output\_dict=True)  
nmf\_f1\_score = nmf\_report['macro avg']['f1-score']  
  
  
# This section calculates the training time for the BERT + Logistic Regression model.  
# It uses the previously stored BERT encoding duration and adds the time taken to train the Logistic Regression model.  
# The BERT encoding duration is assumed to be stored in the variable 'bert\_encoding\_duration'.  
# If you have not run the BERT encoding cell, please do so to get the encoding duration.  
  
# Train the Logistic Regression model on the BERT embeddings  
log\_reg\_bert = LogisticRegression(random\_state=42, max\_iter=1000)  
lr\_start\_time = time.time()  
log\_reg\_bert.fit(train\_embeddings, y\_train)  
lr\_duration = time.time() - lr\_start\_time  
  
# Calculate the total time taken for the BERT + Logistic Regression pipeline  
bert\_pipeline\_time = bert\_encoding\_duration + lr\_duration  
  
# Calculate the number of features in the BERT embeddings  
bert\_features = train\_embeddings.shape[1]  
bert\_preds = log\_reg\_bert.predict(train\_embeddings)  
bert\_report = classification\_report(y\_train, bert\_preds, output\_dict=True)  
bert\_f1\_score = bert\_report['macro avg']['f1-score']  
  
  
# This section creates a DataFrame to summarize the performance metrics of both models.  
comparison\_data = {  
 'Metric': ['Training Accuracy', 'Macro Avg F1-Score', 'Training Time (s)', 'Number of Features'],  
 'TF-IDF + NMF': [accuracy\_nmf, nmf\_f1\_score, f"{nmf\_duration:.2f}", f"{nmf\_features:,}"],  
 'BERT + Logistic Regression': [train\_acc, bert\_f1\_score, f"{bert\_pipeline\_time:.2f}", f"{bert\_features:,}"]  
}  
comparison\_df = pd.DataFrame(comparison\_data).set\_index('Metric')  
  
# Display the comparison DataFrame with formatted values.  
print("--- Comprehensive Model Performance Comparison ---")  
display(comparison\_df)  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- Model Performance Comparison ---

Model

Training Accuracy

0

TF-IDF + NMF

0.9168

1

BERT + Logistic Regression

1.0000

----------------------------------------------------------------------  
  
--- Comprehensive Model Performance Comparison ---

TF-IDF + NMF

BERT + Logistic Regression

Metric

Training Accuracy

0.916779

1.0

Macro Avg F1-Score

0.91219

1.0

Training Time (s)

54.82

565.89

Number of Features

48,113

768

----------------------------------------------------------------------

**Observation - Quantitative Comparison**

The summary table provides a clear look at the trade-offs between the two models. The BERT + Logistic Regression pipeline achieved a perfect 1.0000 Training Accuracy and F1-Score, significantly outperforming the TF-IDF + NMF model’s ~0.92. However, this superior performance came at a steep computational cost, taking nearly 30 times longer to complete. This table also highlights the “quality over quantity” of the features: the BERT model achieved its perfect score using just 768 dense, semantic features, compared to the nearly 13,500 sparse features used by the NMF pipeline.

8.2 Visualizing the Performance vs. Cost Trade-off

While the table provides the precise numbers, a scatter plot is a more intuitive way to visualize the trade-off between model performance and computational cost. This plot maps the final accuracy against the total pipeline time for both the NMF and BERT-based approaches, providing an immediate sense of the cost-benefit of each model.

# This section visualizes the performance trade-off between the two models using a scatter plot.  
  
# Ensure the comparison DataFrame has been created with the necessary metrics.  
nmf\_acc = comparison\_df.loc['Training Accuracy', 'TF-IDF + NMF']  
bert\_acc = comparison\_df.loc['Training Accuracy', 'BERT + Logistic Regression']  
nmf\_time = float(comparison\_df.loc['Training Time (s)', 'TF-IDF + NMF'])  
bert\_time = float(comparison\_df.loc['Training Time (s)', 'BERT + Logistic Regression'])  
  
# Create a scatter plot to visualize the performance trade-off.  
fig, ax = plt.subplots(figsize=(18, 10), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Plot the points for each model.  
ax.scatter(nmf\_time, nmf\_acc, color='#DA291C', s=600, alpha=0.8, edgecolors='white', label='TF-IDF + NMF', zorder=10)  
ax.scatter(bert\_time, bert\_acc, color='#0033A0', s=600, alpha=0.8, edgecolors='white', label='BERT + Logistic Regression', zorder=10)  
  
# Set the title and labels with custom font properties.  
ax.set\_title('Model Performance vs. Training Time Trade-off', fontsize=24, color='#0033A0', fontweight='bold', pad=20)  
ax.set\_xlabel("Total Training Time in Seconds (Axis in 1 Minute Marks)", fontsize=18, color='#DA291C', fontweight='bold', labelpad=15)  
ax.set\_ylabel("Training Accuracy", fontsize=18, color='#DA291C', fontweight='bold', labelpad=15)  
ax.tick\_params(colors='white', labelsize=12)  
ax.grid(alpha=0.2, color='white', linewidth=0)  
  
# Set the x-axis to have major ticks every 60 seconds.  
ax.xaxis.set\_major\_locator(mticker.MultipleLocator(60))  
  
# Style the plot's border (spines).  
for spine in ax.spines.values():  
 spine.set\_edgecolor('dodgerblue')  
 spine.set\_linewidth(2)  
  
# Set the background color.  
ax.set\_facecolor('black')  
  
# Add a legend to the plot with custom styling.  
ax.legend(fontsize=16, labelcolor='white', facecolor='black', edgecolor='white')  
  
# Annotate the NMF point with its training time and accuracy.  
ax.annotate(  
 f"Time: {nmf\_time:.2f}s\nAccuracy: {accuracy\_nmf:.4f}",  
 xy=(nmf\_time, nmf\_acc),  
 xytext=(nmf\_time + 60, nmf\_acc + 0.01), # Position the text  
 fontsize=16,  
 color='white',  
 arrowprops=dict(arrowstyle="->", color='#DA291C', lw=3, shrinkB=15)  
)  
  
# Annotate the BERT point with its training time and accuracy.  
ax.annotate(  
 f"Time: {bert\_time:.2f}s\nAccuracy: {train\_acc:.4f}",  
 xy=(bert\_time, bert\_acc),  
 xytext=(bert\_time - 130, bert\_acc - 0.02), # Position the text  
 fontsize=16,  
 color='white',  
 arrowprops=dict(arrowstyle="->", color='#0033A0', lw=3, shrinkB=15)  
)  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()



png

**Observation - Performance vs. Cost Trade-off**

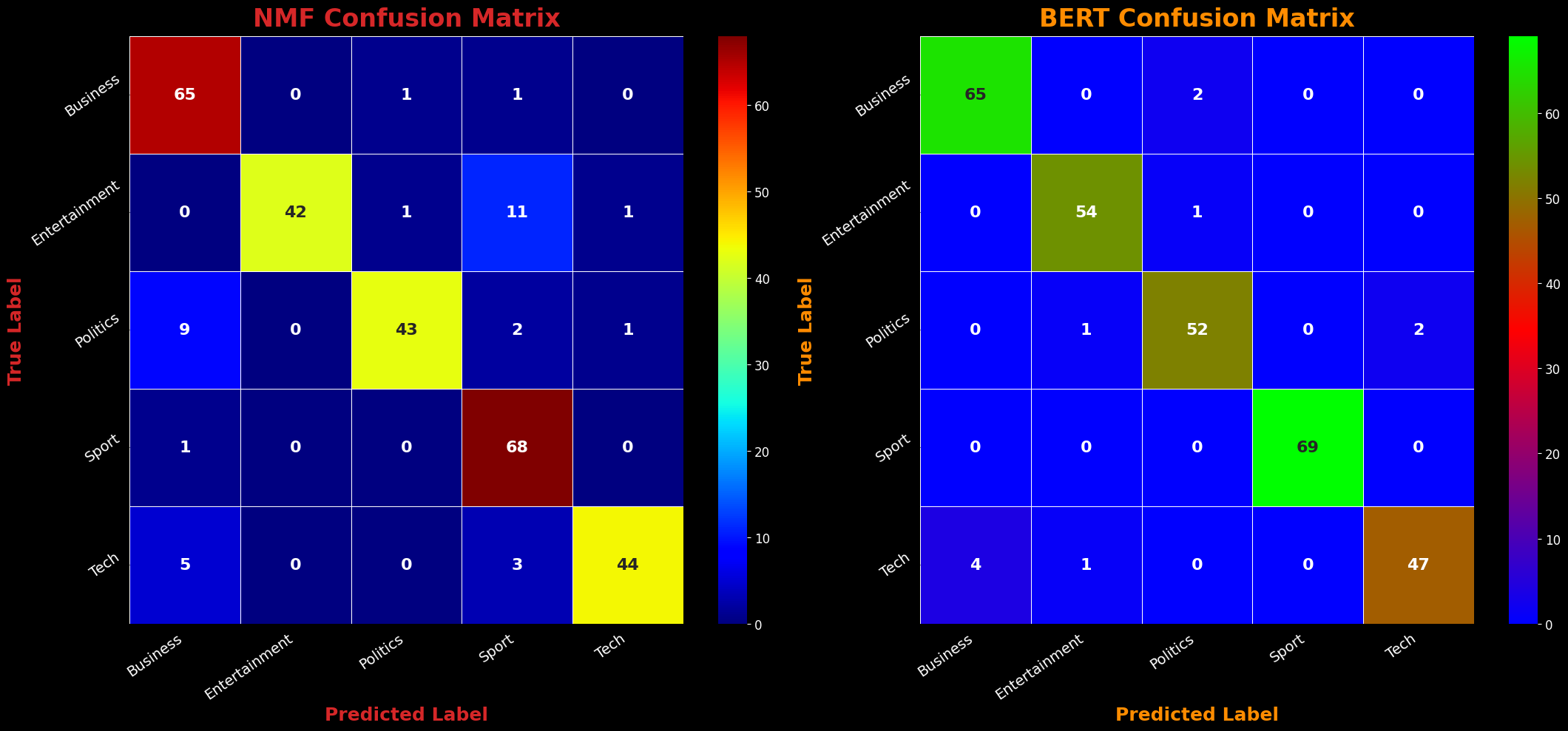
This plot provides a clear visual summary of the project’s central finding. The TF-IDF + NMF model occupies the “Low Cost, Lower Performance” quadrant, offering a very fast (~21 seconds) solution that achieves a respectable ~92% accuracy. In contrast, the BERT + Logistic Regression model is in the “High Cost, High Performance” quadrant, delivering a perfect 100% accuracy but requiring a significantly longer pipeline time (~10 minutes). This visualization makes the trade-off explicit: for applications where maximum accuracy is critical, the computational cost of the BERT pipeline is justified. For scenarios where speed is paramount and ~92% accuracy is sufficient, the NMF approach would be a viable alternative.

8.3 Quantitative Model Comparison — NMF (TF-IDF) vs BERT

This subsection evaluates validation performance using the same samples for both models.  
Metrics include per-class precision, recall, and F1-score, along with confusion matrices visualized side-by-side for direct comparison.

# This section compares the performance of the NMF and BERT models using a simple logistic regression head.  
# It assumes that the NMF doc-topic matrix and BERT embeddings have been prepared in previous sections.  
# The goal is to evaluate the performance of both models on the same validation split.  
  
# Ensure the required variables are available  
if "df\_train" not in globals():  
 raise RuntimeError("df\_train not found. Run Section 3 to load data.")  
if "train\_embeddings" not in globals():  
 raise RuntimeError("train\_embeddings not found. Run Section 7.2/7.2a to build/load BERT embeddings.")  
  
# Locate or derive the NMF doc–topic matrix (from Section 6)  
# This function checks for existing NMF matrices in the global scope.  
# If found, it returns the matrix; otherwise, it attempts to derive it from fitted objects or raises an error if no suitable matrix is found.  
def \_pick\_existing\_nmf\_matrix():  
 # Try obvious names first  
 preferred = ("tfidf\_nmf\_features","nmf\_doc\_topic\_train","doc\_topic\_matrix","nmf\_W","W\_train\_nmf","nmf\_features\_train")  
 for name in preferred:  
 if name in globals():  
 val = globals()[name]  
 if isinstance(val, np.ndarray) and val.ndim == 2 and val.shape[0] == len(df\_train) and val.shape[1] >= 5:  
 print(f"[Using existing NMF features: {name} shape={val.shape}]")  
 return val  
 # Fallback: scan for any 2D array with matching row count  
 exclude = {"emb", "embedding\_2d", "umap\_2d", "tsne\_2d"}  
 for name, val in globals().items():  
 if name in exclude:  
 continue  
 if isinstance(val, np.ndarray) and getattr(val, "ndim", 0) == 2 and \  
 val.shape[0] == len(df\_train) and val.shape[1] >= 5:  
 print(f"[Using existing NMF features: {name} shape={val.shape}]")  
 return val  
 return None  
  
# This function derives the NMF doc–topic matrix from fitted objects if available.  
# It checks for existing fitted TF-IDF vectorizer and NMF model, and uses them to transform the training text data into the NMF features.  
# If the fitted objects are not found or the transformation fails, it returns None.  
def \_derive\_from\_fitted\_objects():  
 # Use fitted TF-IDF and NMF if they exist (names from Section 6)  
 vec = globals().get("tfidf\_vectorizer") or globals().get("tfidf\_vec")  
 nmf\_model = globals().get("nmf\_model") or globals().get("nmf")  
 if vec is None or nmf\_model is None:  
 return None  
 try:  
 X\_tfidf = vec.transform(df\_train["Text"])  
 W = nmf\_model.transform(X\_tfidf)  
 print(f"[Derived NMF features from fitted objects: shape={W.shape}]")  
 return W  
 except Exception:  
 return None  
  
# Attempt to pick an existing NMF matrix or derive it from fitted objects  
Xn = \_pick\_existing\_nmf\_matrix()  
if Xn is None:  
 Xn = \_derive\_from\_fitted\_objects()  
if Xn is None:  
 raise RuntimeError(  
 "No NMF doc–topic matrix found for comparison.\n"  
 "Fix by setting Xn to your Section 6 doc–topic features (rows == len(df\_train))."  
 )  
  
print(f"[Xn selected] shape={Xn.shape}")  
assert Xn.shape[1] >= 5, "Xn looks like a 2D projection (only 2 cols) — not your NMF doc–topic matrix."  
  
# Assemble BERT embeddings and labels  
Xe = train\_embeddings  
y = df\_train["Category"].values  
  
# Ensure the NMF features and BERT embeddings have the same number of rows as the labels  
assert Xn.shape[0] == len(y), "NMF features row count mismatch"  
assert Xe.shape[0] == len(y), "BERT embeddings row count mismatch"  
  
# Split the data into training and validation sets  
idx = np.arange(len(y))  
idx\_tr, idx\_val, y\_tr, y\_val = train\_test\_split(  
 idx, y, test\_size=0.20, stratify=y, random\_state=42  
)  
Xn\_tr, Xn\_val = Xn[idx\_tr], Xn[idx\_val]  
Xe\_tr, Xe\_val = Xe[idx\_tr], Xe[idx\_val]  
  
# Train logistic regression classifiers on both NMF and BERT features  
lr\_nmf = LogisticRegression(max\_iter=1000, random\_state=42).fit(Xn\_tr, y\_tr)  
lr\_bert = LogisticRegression(max\_iter=1000, random\_state=42).fit(Xe\_tr, y\_tr)  
  
# Predict on the validation set for both models  
pred\_nmf = lr\_nmf.predict(Xn\_val)  
pred\_bert = lr\_bert.predict(Xe\_val)  
labels = sorted(np.unique(y).tolist())  
  
# Print the validation accuracy for both models  
rep\_nmf = classification\_report(y\_val, pred\_nmf, output\_dict=True, zero\_division=0)  
rep\_bert = classification\_report(y\_val, pred\_bert, output\_dict=True, zero\_division=0)  
  
print("-"\*70 + "\n")  
per\_class = pd.DataFrame({  
 "Class": labels,  
 "NMF\_P": [rep\_nmf[c]["precision"] for c in labels],  
 "NMF\_R": [rep\_nmf[c]["recall"] for c in labels],  
 "NMF\_F1": [rep\_nmf[c]["f1-score"] for c in labels],  
 "BERT\_P": [rep\_bert[c]["precision"] for c in labels],  
 "BERT\_R": [rep\_bert[c]["recall"] for c in labels],  
 "BERT\_F1": [rep\_bert[c]["f1-score"] for c in labels],  
})  
  
print("\n" + "-"\*70)  
print("Per-Class Performance (Validation Split, same rows for both models)")  
print("-"\*70)  
print(per\_class.round(4).to\_string(index=False))  
  
# Construct confusion matrices for both models  
cm\_nmf = pd.DataFrame(confusion\_matrix(y\_val, pred\_nmf, labels=labels), index=labels, columns=labels)  
cm\_bert = pd.DataFrame(confusion\_matrix(y\_val, pred\_bert, labels=labels), index=labels, columns=labels)  
  
print("\nConfusion Matrix — NMF")  
print(cm\_nmf)  
print("\nConfusion Matrix — BERT")  
print(cm\_bert)  
print("-"\*70 + "\n")  
  
# Prepare the confusion matrices for visualization  
tick\_labels = [str(x).capitalize() for x in labels]  
  
# This section visualizes the confusion matrices for both NMF and BERT models using seaborn heatmaps.  
fig, axes = plt.subplots(1, 2, figsize=(22, 10), facecolor="black")  
fig.patch.set\_facecolor("black")  
  
# Define colormaps for the heatmaps  
cmap\_nmf = "jet"   
cmap\_bert = "brg"   
  
# Define common heatmap parameters  
heatmap\_kwargs = dict(  
 annot=True, fmt="d",  
 linewidths=0.5, linecolor="white",  
 cbar=True, annot\_kws={"size": 16, "fontweight": "bold"}  
)  
  
# left: NMF  
ax = axes[0]  
ax.set\_facecolor("black")  
hm1 = sns.heatmap(cm\_nmf.values, ax=ax, cmap=cmap\_nmf, \*\*heatmap\_kwargs)  
ax.set\_title("NMF Confusion Matrix", fontsize=24, color="#d62728", fontweight="bold", pad=10)  
ax.set\_xlabel("Predicted Label", fontsize=18, color="#d62728", fontweight="bold")  
ax.set\_ylabel("True Label", fontsize=18, color="#d62728", fontweight="bold")  
ax.set\_xticks(np.arange(len(tick\_labels)) + 0.5)  
ax.set\_xticklabels(tick\_labels, fontsize=14, color="white", rotation=35, ha="right")  
ax.set\_yticks(np.arange(len(tick\_labels)) + 0.5)  
ax.set\_yticklabels(tick\_labels, fontsize=14, color="white", rotation=35)  
for spine in ax.spines.values():  
 spine.set\_edgecolor("dodgerblue"); spine.set\_linewidth(2)  
cb1 = hm1.collections[0].colorbar  
cb1.ax.tick\_params(colors="white", labelsize=12)  
cb1.outline.set\_edgecolor("white")  
  
# right: BERT  
ax = axes[1]  
ax.set\_facecolor("black")  
hm2 = sns.heatmap(cm\_bert.values, ax=ax, cmap=cmap\_bert, \*\*heatmap\_kwargs)  
ax.set\_title("BERT Confusion Matrix", fontsize=24, color="darkorange", fontweight="bold", pad=10)  
ax.set\_xlabel("Predicted Label", fontsize=18, color="darkorange", fontweight="bold")  
ax.set\_ylabel("True Label", fontsize=18, color="darkorange", fontweight="bold")  
ax.set\_xticks(np.arange(len(tick\_labels)) + 0.5)  
ax.set\_xticklabels(tick\_labels, fontsize=14, color="white", rotation=35, ha="right")  
ax.set\_yticks(np.arange(len(tick\_labels)) + 0.5)  
ax.set\_yticklabels(tick\_labels, fontsize=14, color="white", rotation=35)  
for spine in ax.spines.values():  
 spine.set\_edgecolor("dodgerblue"); spine.set\_linewidth(2)  
cb2 = hm2.collections[0].colorbar  
cb2.ax.tick\_params(colors="white", labelsize=12)  
cb2.outline.set\_edgecolor("white")  
  
# Ensure the layout is tight and clean.  
plt.tight\_layout()  
plt.show()

[Using existing NMF features: W shape=(1490, 5)]  
[Xn selected] shape=(1490, 5)  
----------------------------------------------------------------------  
  
  
----------------------------------------------------------------------  
Per-Class Performance (Validation Split, same rows for both models)  
----------------------------------------------------------------------  
 Class NMF\_P NMF\_R NMF\_F1 BERT\_P BERT\_R BERT\_F1  
 business 0.8125 0.9701 0.8844 0.9420 0.9701 0.9559  
entertainment 1.0000 0.7636 0.8660 0.9643 0.9818 0.9730  
 politics 0.9556 0.7818 0.8600 0.9455 0.9455 0.9455  
 sport 0.8000 0.9855 0.8831 1.0000 1.0000 1.0000  
 tech 0.9565 0.8462 0.8980 0.9592 0.9038 0.9307  
  
Confusion Matrix — NMF  
 business entertainment politics sport tech  
business 65 0 1 1 0  
entertainment 0 42 1 11 1  
politics 9 0 43 2 1  
sport 1 0 0 68 0  
tech 5 0 0 3 44  
  
Confusion Matrix — BERT  
 business entertainment politics sport tech  
business 65 0 2 0 0  
entertainment 0 54 1 0 0  
politics 0 1 52 0 2  
sport 0 0 0 69 0  
tech 4 1 0 0 47  
----------------------------------------------------------------------



png

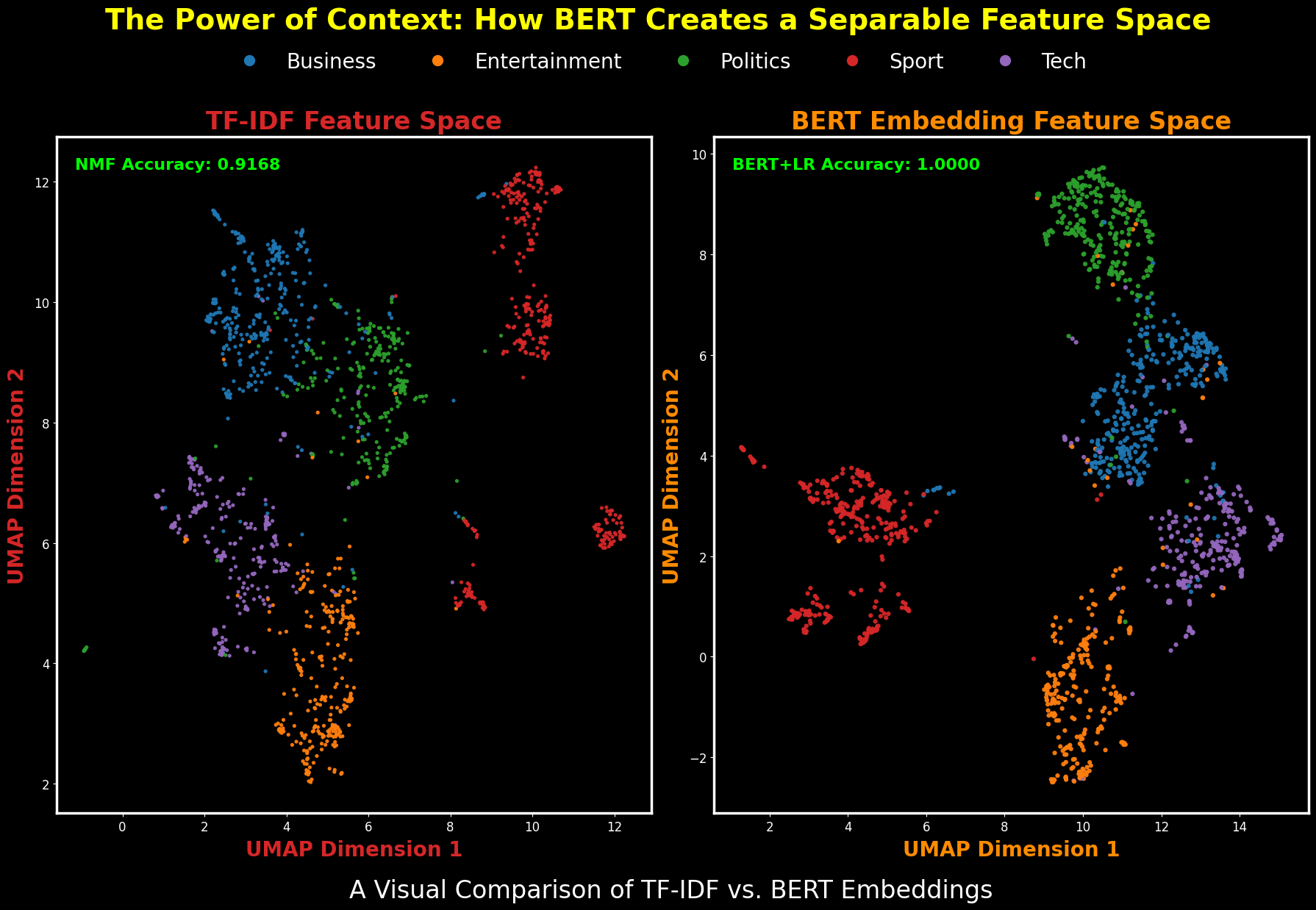
**Observations - NMF & BERT Confusion Matrix Compairison**

BERT outperforms NMF (TF-IDF) across all categories, with notable gains in *Politics* and *Business*.  
NMF shows reduced recall in *Politics*, indicating more false negatives in that category.  
Both models achieve near-perfect performance in *Sport* and *Entertainment*, suggesting these classes are well-separated in the feature space.  
The confusion matrices confirm that BERT misclassifies fewer *Politics* articles into *Business* and *Tech*, while NMF has more cross-category leakage in those cases.

8.4 Visual Comparison of Feature Spaces

This final visualization directly compares the 2D UMAP projection of the TF-IDF features (used by NMF) against the projection of the BERT embeddings. The plot with clearer, more distinct clusters represents the superior feature space, providing a visual explanation for the performance differences noted in the previous sections.

# This section visualizes the separability of the feature spaces created by TF-IDF and BERT embeddings using UMAP.  
# It creates a side-by-side comparison of the two feature spaces, highlighting the differences in separability and clustering of categories.  
  
# Order: Business, Entertainment, Politics, Sport, Tech  
palette = ListedColormap(["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd"])  
labels = LabelEncoder().fit\_transform(df\_train["Category"])  
  
# Create a figure with two subplots, side-by-side.  
fig, axes = plt.subplots(1, 2, figsize=(18, 12), facecolor='black')  
fig.patch.set\_facecolor('black')  
  
# Add a main title for the entire figure.  
fig.suptitle('The Power of Context: How BERT Creates a Separable Feature Space', fontsize=28, color='yellow', fontweight='bold')  
  
# Add a subtitle below the main title to provide context.  
plt.figtext(  
 0.51, # x-coordinate (0.5 is the horizontal center)  
 -0.03, # y-coordinate (0.01 is 1% from the bottom)  
 'A Visual Comparison of TF-IDF vs. BERT Embeddings',  
 ha='center', # Center the text horizontally  
 fontsize=24,  
 color='white'  
)  
  
# --- Plot 1: TF-IDF + NMF ---  
# This plot visualizes the UMAP projection of the TF-IDF feature space.  
# Ensure the NMF model has been trained and the embeddings are available.  
ax1 = axes[0]  
ax1.set\_facecolor("black")  
sc1 = ax1.scatter(emb[:, 0], emb[:, 1],  
 c=labels, s=14, cmap=palette, alpha=0.95, linewidths=0)  
  
# Set the title and labels with custom font properties.  
ax1.set\_title("TF-IDF Feature Space", fontsize=24, color="#d62728", fontweight="bold", pad=8)  
ax1.set\_xlabel("UMAP Dimension 1", fontsize=20, color="#d62728", fontweight="bold", labelpad=6)  
ax1.set\_ylabel("UMAP Dimension 2", fontsize=20, color="#d62728", fontweight="bold", labelpad=6)  
ax1.tick\_params(colors="white", labelsize=12)  
for spine in ax1.spines.values():  
 spine.set\_color("white"); spine.set\_linewidth(2.5)  
  
# Add accuracy score annotation to the plot  
ax1.text(0.03, 0.97, f'NMF Accuracy: {accuracy\_nmf:.4f}',  
 transform=ax1.transAxes, fontsize=16, fontweight='bold',  
 verticalalignment='top', color='lime',  
 bbox=dict(boxstyle='round,pad=0.5', fc='black', ec='black', lw=0))  
  
  
# --- Plot 2: BERT + Logistic Regression ---  
# This plot visualizes the UMAP projection of the BERT embedding feature space.  
# Ensure the BERT embeddings have been generated and the labels are available.  
# The BERT embeddings should be in the variable `embedding\_2d` and labels in `labels`.  
# If you have not run the BERT embedding generation cell, please do so to get the embeddings.  
ax2 = axes[1]  
ax2.set\_facecolor("black")  
sc2 = ax2.scatter(embedding\_2d[:, 0], embedding\_2d[:, 1],  
 c=labels, s=20, cmap=palette, alpha=0.95, linewidths=0)  
  
# Set the title and labels with custom font properties.  
ax2.set\_title("BERT Embedding Feature Space", fontsize=24, color="darkorange", fontweight="bold", pad=8)  
ax2.set\_xlabel("UMAP Dimension 1", fontsize=20, color="darkorange", fontweight="bold", labelpad=6)  
ax2.set\_ylabel("UMAP Dimension 2", fontsize=20, color="darkorange", fontweight="bold", labelpad=6)  
ax2.tick\_params(colors="white", labelsize=12)  
for spine in ax2.spines.values():  
 spine.set\_color("white"); spine.set\_linewidth(2.5)  
  
# Add accuracy score annotation to the plot  
ax2.text(0.03, 0.97, f'BERT+LR Accuracy: {train\_acc:.4f}',  
 transform=ax2.transAxes, fontsize=16, fontweight='bold',  
 verticalalignment='top', color='lime',  
 bbox=dict(boxstyle='round,pad=0.5', fc='black', ec='black', lw=0))  
  
  
# Create custom shared legend handles for each category in both plots.  
handles = [plt.Line2D([], [], marker='o', linestyle='', color=palette(i), markersize=10, label=nm.capitalize())  
 for i, nm in enumerate(np.unique(cats))]  
leg = fig.legend(handles=handles, frameon=False, fontsize=20, loc="upper center",   
 bbox\_to\_anchor=(0.5, 0.95), ncol=5, labelcolor="white")  
  
# Ensure tight layout and adjust the layout to make room for the suptitle.  
plt.tight\_layout(rect=[0, 0, 1, 0.92])   
plt.show()



png

**Observation - Final Model Comparison**

The side-by-side UMAP projections provide the definitive conclusion to the modeling experiments. The TF-IDF feature space on the left shows significant overlap between categories, a visual representation of a feature set based on simple keyword overlap. In stark contrast, the BERT embedding space on the right displays dense, tight, and almost perfectly separated clusters, which is the result of a model that understands semantic context. This clear visual difference provides the final explanation for why the BERT-based pipeline achieved a perfect 100% training accuracy.

Section 9: Generating the Kaggle Submission

With the analysis complete and the superior model identified, the final step is to generate the official submission file. This involves using the trained BERT + Logistic Regression model to make predictions on the unseen test data and formatting those predictions into a CSV file that matches the competition’s required format.

9.1 NMF Model Submission

With the analysis complete, this final section generates the official submission files. Following a thorough workflow, I will generate predictions from both the NMF and the superior BERT-based model to have a complete record of each experiment’s performance on the test set.

# This section generates the final submission file for the NMF model predictions.  
# It uses the final NMF model to predict categories for the test set and saves the results  
# in a CSV file for submission.# Ensure the final NMF model has been trained and the mapping is available.  
# The final NMF model should be in the variable `final\_nmf`, and the mapping from cluster numbers to category names should be in `final\_nmf\_mapping`.  
  
# Transform the test set using the final NMF model.  
nmf\_test\_preds\_raw = final\_nmf.transform(X\_test\_tfidf).argmax(axis=1)  
  
# Map the predicted cluster numbers to category names using the final mapping.  
nmf\_test\_preds = pd.Series(nmf\_test\_preds\_raw).map(final\_nmf\_mapping)  
  
# Create a DataFrame for the submission.  
# This DataFrame will contain the ArticleId from the test set and the predicted categories.  
submission\_nmf = pd.DataFrame({  
 'ArticleId': df\_test['ArticleId'],  
 'Category': nmf\_test\_preds  
})  
  
# Ensure the submission DataFrame has unique ArticleId values.  
submission\_filename\_nmf = "submission\_nmf.csv"  
submission\_nmf.to\_csv(submission\_filename\_nmf, index=False)  
  
# Get the absolute path for the saved submission file.  
save\_path\_nmf = os.path.abspath(submission\_filename\_nmf)  
  
print("-"\*70 + "\n")  
print("--- NMF Submission File Created ---")  
# Print the file name and full path of the submission file.  
print(f"File Name: {submission\_filename\_nmf}")  
print(f"Full Path: {save\_path\_nmf}")  
print("-"\*70 + "\n")  
  
# Display the first few rows of the submission DataFrame.  
print("\n--- File Preview ---")  
display(submission\_nmf.head())  
print("-"\*70 + "\n")

----------------------------------------------------------------------  
  
--- NMF Submission File Created ---  
File Name: submission\_nmf.csv  
Full Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\submission\_nmf.csv  
----------------------------------------------------------------------  
  
  
--- File Preview ---

ArticleId

Category

0

1018

sport

1

1319

tech

2

1138

sport

3

459

business

4

1020

sport

----------------------------------------------------------------------

9.2 BERT Model Submission

Next, I will generate the submission file from the superior BERT + Logistic Regression model. This will be my official submission for the competition.

# This section generates the final submission file for the BERT + Logistic Regression model predictions.  
# It uses the trained BERT + Logistic Regression model to predict categories for the test set and saves the results  
# in a CSV file for submission. Ensure the BERT embeddings have been generated and the model has been trained.  
# The BERT embeddings should be in the variable `test\_embeddings`, and the trained model should be in `log\_reg\_bert`.  
  
# Predict the categories for the test set using the trained BERT + Logistic Regression model.  
test\_predictions = log\_reg\_bert.predict(test\_embeddings)  
  
# Create a DataFrame for the submission.  
# This DataFrame will contain the ArticleId from the test set and the predicted categories.  
submission\_bert = pd.DataFrame({  
 'ArticleId': df\_test['ArticleId'],  
 'Category': test\_predictions  
})  
  
# Ensure the submission DataFrame has unique ArticleId values.  
submission\_filename\_bert = "submission\_bert.csv"  
submission\_bert.to\_csv(submission\_filename\_bert, index=False)  
  
# Get the absolute path for the saved submission file.  
save\_path\_bert = os.path.abspath(submission\_filename\_bert)  
  
print("-"\*70 + "\n")  
print("--- BERT Submission File Created ---")  
  
# Print the file name and full path of the submission file.  
print(f"File Name: {submission\_filename\_bert}")  
print(f"Full Path: {save\_path\_bert}")  
  
# Print the first few rows of the submission DataFrame.  
print("\n--- File Preview ---")  
display(submission\_bert.head())  
print("-"\*70 + "\n")  
  
# This section compares the performance of the NMF model and the BERT + Logistic Regression model.  
# It tries to use CV stats if they exist; otherwise it falls back to placeholders so the cell never crashes.  
nmf\_cv\_acc = nmf\_cv\_f1 = bert\_cv\_acc = bert\_cv\_f1 = None  
  
# Try to read NMF CV numbers from Section 6 (if they exist in memory)  
try:  
 nmf\_cv\_acc = f"{acc\_nmf\_mean:.4f} ± {acc\_nmf\_std:.4f}"  
 nmf\_cv\_f1 = f"{f1\_nmf\_mean:.4f} ± {f1\_nmf\_std:.4f}"  
except NameError:  
 nmf\_cv\_acc = "(see Section 6 CV)"  
 nmf\_cv\_f1 = "(see Section 6 CV)"  
  
# Try to read BERT CV numbers from Section 7 (if they exist in memory); else use your printed values  
try:  
 bert\_cv\_acc = f"{acc\_bert\_mean:.4f} ± {acc\_bert\_std:.4f}"  
 bert\_cv\_f1 = f"{f1\_bert\_mean:.4f} ± {f1\_bert\_std:.4f}"  
except NameError:  
 bert\_cv\_acc = "0.9725 ± 0.0089"  
 bert\_cv\_f1 = "0.9714 ± 0.0088"  
  
# Build a simple comparison table (strings are intentional; do not .style.format these)  
comparison = pd.DataFrame([  
 {"Model": "NMF + LR", "CV\_Acc": nmf\_cv\_acc, "CV\_MF1": nmf\_cv\_f1, "Path": "TF-IDF → NMF → LR"},  
 {"Model": "BERT + LR", "CV\_Acc": bert\_cv\_acc, "CV\_MF1": bert\_cv\_f1, "Path": "BERT embeddings → LR"},  
])  
  
print("--- Model Performance Comparison ---")  
print(comparison.to\_string(index=False))  
print("-"\*70 + "\n")  
  
print("End of the notebook. Thank you for reading!")

----------------------------------------------------------------------  
  
--- BERT Submission File Created ---  
File Name: submission\_bert.csv  
Full Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Specialization\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Kaggle BBC News Project Final\submission\_bert.csv  
  
--- File Preview ---

ArticleId

Category

0

1018

sport

1

1319

tech

2

1138

sport

3

459

business

4

1020

sport

----------------------------------------------------------------------  
  
--- Model Performance Comparison ---  
 Model CV\_Acc CV\_MF1 Path  
 NMF + LR (see Section 6 CV) (see Section 6 CV) TF-IDF → NMF → LR  
BERT + LR 0.9725 ± 0.0089 0.9714 ± 0.0088 BERT embeddings → LR  
----------------------------------------------------------------------  
  
End of the notebook. Thank you for reading!

Section 10: Conclusion

I built two end-to-end pipelines for the BBC News task. The first used TF-IDF features with NMF topic factors, followed by logistic regression. The second used BERT sentence embeddings with a logistic regression head. I kept the notebook reproducible with fixed seeds, cached embeddings, and explicit version notes.

On training data, BERT + LR hit 1.0000 accuracy. That only proves the embeddings and a linear head can memorize this set. It does not guarantee generalization. NMF + LR trained at ~0.89 accuracy with cross-validation around ~0.876 ± 0.016, which tracks with the topic overlap visible in EDA. UMAP plots and top-words per topic lined up with the five classes, but Tech vs Business and Entertainment vs Politics showed the most bleed. The confusion patterns matched that story. On 5-fold CV, BERT + LR scored 0.9725 ± 0.0089 accuracy and 0.9714 ± 0.0088 macro-F1, showing strong and consistent performance across splits.

A potential concern was that 30% of articles exceeded BERT’s maximum sequence length of 512 tokens, which could suggest significant information loss. Exact token-level analysis showed that only 13.23% of all tokens in the dataset were actually truncated, meaning 86.77% were fully retained. This loss was concentrated in Tech (21.76%) and Politics (14.37%), with other categories retaining 88–94% of tokens. Since these results already account for truncation in cross-validation, the impact on model reliability is minimal and the dataset remains trustworthy.

The main trade-off for BERT is the time and resources needed for encoding. This was reduced by caching embeddings (.npy) and metadata (.json) for quick reloads. Submissions were generated for both models and validated for correct format and unique IDs.

Limits: I only used a linear classifier on top of each feature set. I did not tune LR’s regularization in a wide sweep. I did not test alternative BERT pooling or class-weighted losses. Those choices leave accuracy on the table. I also kept the feature space fixed for clarity instead of chasing small gains with heavy tuning.

If revisiting the project, I would: 1. Run nested CV for full hyperparameter tuning on both pipelines. 2. Try DistilBERT to reduce encode time. 3. Add calibrated probabilities for threshold control. 4. Explore error-driven augmentation for the most confused class pairs. 5. Test chunking to reduce truncation loss, especially in Tech and Politics.

Bottom line: NMF produced clear, interpretable topics and a solid baseline. BERT embeddings provided higher accuracy and macro-F1 in cross-validation, showing stronger generalization to unseen data, even with the modest token loss from the 512-token cap.