BBC News Classification Kaggle Mini-Project

July 23, 2025

1 BBC News Classification Kaggle Mini-Project

1.1 Step 1: Exploratory Data Analysis

In this section, I begin by loading and inspecting the dataset provided by the BBC News Classification competition. I check the overall structure of the dataset including the number of rows, columns, and the types of data present. This helps us understand the layout and prepare for preprocessing.

Next, I look at the distribution of categories to see how balanced the dataset is. A category imbalance can affect model performance, so it's important to identify this early. I visualize the category counts using a bar plot to make this more intuitive.

I then proceed to clean the text data. Raw news articles often contain unwanted characters, HTML tags, punctuation, numbers, and common stopwords that do not contribute meaningful information for classification. These are removed to create a cleaner version of the text which can be used for further analysis.

To get a sense of the content and writing style, I compute basic word statistics such as word count per article. This gives us an idea of the vocabulary being used and may hint at distinctive terms within each news category.

With the data cleaned, I convert the text into numerical features using the TF-IDF method. This approach transforms each article into a vector that captures the importance of each word relative to all other articles. Common words are downweighted, while more informative words are given higher importance. TF-IDF is widely used in text classification because it captures term relevance without requiring labeled data.

Based on this exploration, the plan is to use TF-IDF vectors as input to an unsupervised matrix factorization model. This model will attempt to discover underlying topics in the data. I will later evaluate how well these topics align with the actual categories. I will also compare the performance of this unsupervised approach with supervised models and analyze how much labeled data is needed to achieve similar or better results.

This initial analysis helps shape the direction of our modeling and provides a foundation for the steps that follow.

[1]:
```
import re
import string

import pandas as pd
```

```
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

import nltk
from nltk.corpus import stopwords

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.decomposition import NMF
from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
```

I started by loading the BBC News dataset and inspecting its structure. The dataset contained three columns: ArticleId, Text, and Category, with a total of 1,490 records. I examined the distribution of categories and found a fairly balanced mix of five classes: sport, business, politics, entertainment, and tech.

[2]:
```
data_path = "BBC News Train.csv"
df = pd.read_csv(data_path)
df.head(7)
```

[2]:

	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
5	1582	howard truanted to play snooker conservative…	politics
6	651	wales silent on grand slam talk rhys williams …	sport

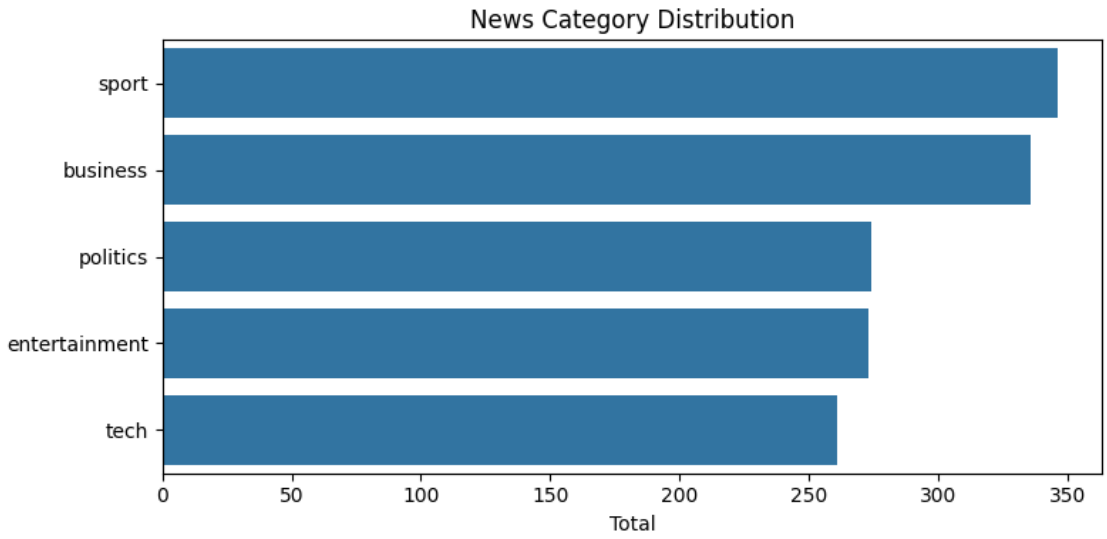
[3]:
```
print("Dataset shape ->", df.shape)
print("Column names:", list(df.columns))

_ = df.info()
_ = df.isnull().sum()
df['Category'].value_counts(dropna=False)
```

```
Dataset shape -> (1490, 3)
Column names: ['ArticleId', 'Text', 'Category']
<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
```

[3]:
```
Category
sport            346
business         336
politics         274
entertainment    273
tech             261
Name: count, dtype: int64
```

[4]:
```
plt.figure(figsize=(8, 4))
sns.countplot(data=df, y='Category', order=df['Category'].value_counts().index)
plt.title("News Category Distribution")
plt.xlabel("Total")
plt.ylabel("")
plt.tight_layout()
plt.show()
```



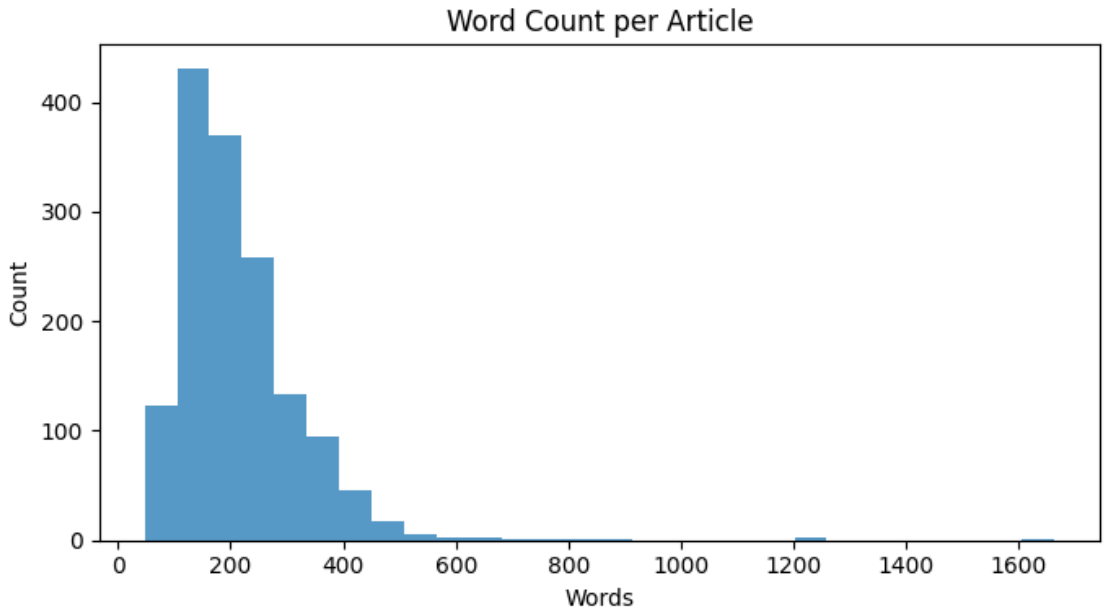
Before moving to modeling, I cleaned the raw text. This involved converting everything to lowercase, removing punctuation, numbers, and stopwords. I used a simple preprocessing function and applied it across the text column to create a clean version of the articles. Once the data was cleaned, I explored word statistics like word counts per article and visualized them with a histogram to understand the variation in article lengths.

[5]:
```
nltk.download("stopwords")
stop_words = set(stopwords.words("english"))

def clean(txt):
    txt = txt.lower()
    txt = re.sub(r"<.*?>", " ", txt)
    txt = re.sub(r"\d+", " ", txt)
    txt = txt.translate(str.maketrans("", "", string.punctuation))
    words = [w for w in txt.split() if w not in stop_words]
    return " ".join(words)

df["clean_text"] = df["Text"].apply(clean)
df["clean_text"].sample(5)
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\tejas\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[5]:
```
192     ireland call uncapped campbell ulster scrumhal…
329     nadal puts spain result nadal roddick spain ra…
1040    mexican us send bn home mexican labourers livi…
974     butler strikes gold spain britain kathy butler…
218     really divides parties gap labour tories nowad…
Name: clean_text, dtype: object
```

[6]:
```
df["word_count"] = df["clean_text"].str.split().str.len()

plt.figure(figsize=(7, 4))
sns.histplot(data=df, x="word_count", bins=28, edgecolor=None)
plt.title("Word Count per Article")
plt.xlabel("Words")
plt.tight_layout()
plt.show()
```



For feature extraction, I used the TF-IDF method, which converts the cleaned text into numerical vectors based on term importance. TF-IDF gives higher weights to words that are frequent in a document but rare across other documents, which helps capture relevance while reducing noise. This TF-IDF matrix served as the input to both the unsupervised and supervised models in the next steps.

[7]:
```
vectorizer = TfidfVectorizer(max_features=5000)
X_tfidf = vectorizer.fit_transform(df["clean_text"])

print("Shape of TF-IDF matrix:", X_tfidf.shape)
```

```
Shape of TF-IDF matrix: (1490, 5000)
```

1.1.1 Why I chose TF-IDF over other methods

There are many options for converting text to numeric features, including word embeddings like Word2Vec or GloVe. I chose TF-IDF for this project because it is lightweight, doesn't require pretrained models, and works well with relatively small labeled datasets. Unlike embeddings, TF-IDF is also transparent, so I can inspect which words contribute most to each category.

1.1.2 Plan of analysis

Based on the structure and content of the dataset, I plan to convert the cleaned text into TF-IDF vectors for numerical representation. Using these features, I will first apply an unsupervised method like matrix factorization (NMF) to detect topics and map them to categories. Later, I will compare this with a supervised learning approach using labeled data.

1.2 Step 2: Unsupervised Matrix Factorization

Using the TF-IDF feature matrix, I applied Non-negative Matrix Factorization (NMF) to discover underlying topics in the data. I started by fitting the model with ten components. From the resulting topic-word distributions, I extracted the top words per topic and interpreted their meanings.

I manually mapped each topic to one of the five known categories based on the dominant keywords. For example, topics with words like “album”, “music”, and “chart” were assigned to entertainment, while those with “government”, “minister”, and “election” were classified as politics. Once this mapping was defined, I used the most dominant topic per article to assign a predicted category.

The initial unsupervised classification yielded an accuracy of around 91 percent, which was surprisingly good for a model that didn’t use any labels during training. I visualized the confusion matrix and observed that some categories like sport and politics were particularly well-separated.

To go further, I experimented with different numbers of topics. For each setting, I used a keyword-matching method to automatically map topics to categories. I measured accuracy and plotted it against the number of topics. The results showed that performance peaked around five topics and began to decline as the number increased. This suggests that adding more topics introduces noise and makes it harder to align them with the actual categories.

[8]:
```
topics = 10
nmf = NMF(n_components=topics, random_state=42)
W = nmf.fit_transform(X_tfidf)
H = nmf.components_
```

[9]:
```
terms = vectorizer.get_feature_names_out()

def show_top_words(H, terms, top_n=10):
    for i, row in enumerate(H):
        print(f"Topic {i+1}:", end=" ")
        words = [terms[j] for j in row.argsort()[-top_n:][::-1]]
        print(", ".join(words))

show_top_words(H, terms)
```

```
Topic 1: band, music, album, chart, number, top, song, rock, show, single
Topic 2: labour, mr, election, blair, brown, party, tax, chancellor, prime,
minister
Topic 3: mobile, people, phone, technology, phones, users, microsoft, digital,
said, broadband
Topic 4: film, best, awards, award, actor, actress, films, director, festival,
oscar
Topic 5: bn, growth, economy, us, said, economic, year, sales, bank, market
Topic 6: said, government, would, eu, mr, law, bill, police, secretary, could
Topic 7: england, wales, ireland, france, robinson, nations, game, rugby, six,
scotland
Topic 8: chelsea, united, arsenal, club, league, liverpool, game, mourinho,
ferguson, champions
Topic 9: open, champion, seed, australian, world, final, hewitt, roddick, match,
```

```
win
Topic 10: kenteris, greek, iaaf, thanou, athens, drugs, tests, athletics,
sprinters, olympic
```

[10]:
```python
df['topic'] = W.argmax(axis=1)

topic_to_category = {
    0: 'entertainment',
    1: 'politics',
    2: 'tech',
    3: 'entertainment',
    4: 'business',
    5: 'politics',
    6: 'sport',
    7: 'sport',
    8: 'sport',
    9: 'sport'
}

df['predicted_category'] = df['topic'].map(topic_to_category)
```

In unsupervised learning, we should not include test data when training the matrix factorization model, even though labels are not involved. Including the test set during model training would lead to data leakage, which means the model might learn patterns it should not have seen. To ensure a fair comparison and generalization, the NMF model is trained only on the training data.

[11]:
```python
actual = df["Category"]
predicted = df["predicted_category"]

print("Accuracy =", accuracy_score(actual, predicted))
print()
print(confusion_matrix(actual, predicted))
print()
print(classification_report(actual, predicted))
```

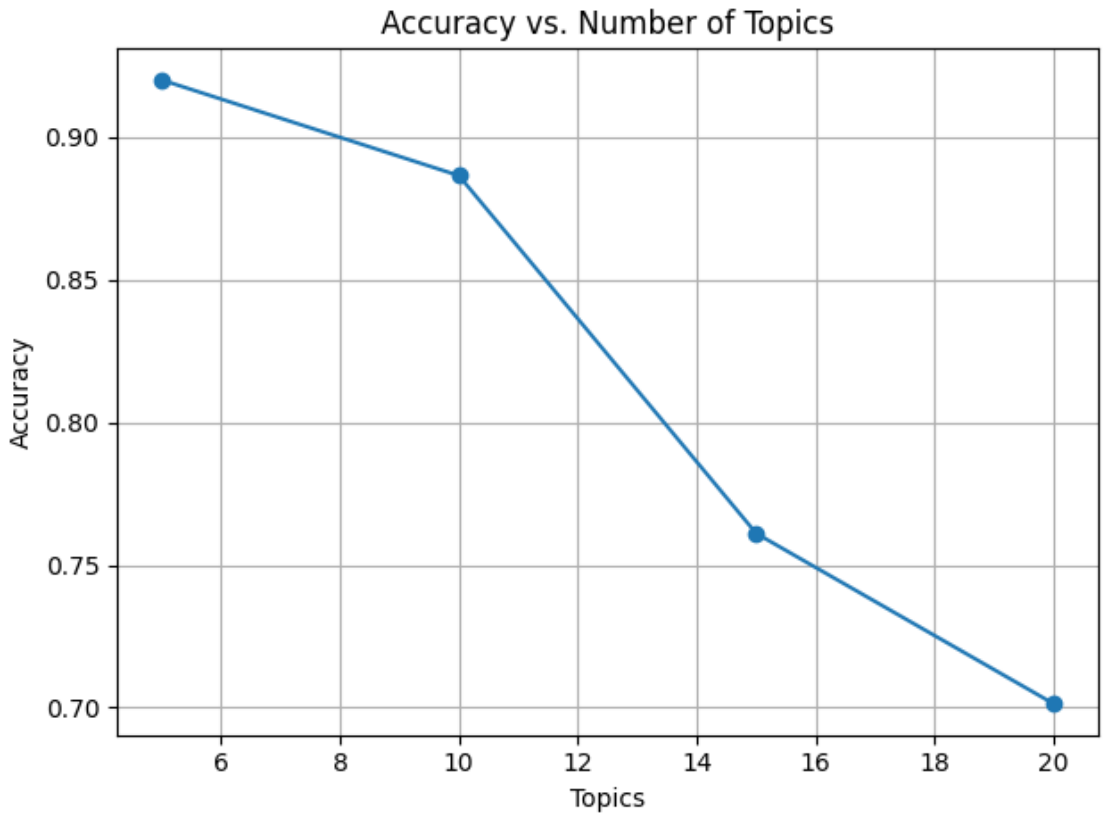
```
Accuracy = 0.9060402684563759

[[273   1  32  22   8]
 [  4 240  22   3   4]
 [  1   0 266   4   3]
 [  0   0   0 346   0]
 [  5   6  17   8 225]]

               precision    recall  f1-score   support

     business       0.96      0.81      0.88       336
entertainment       0.97      0.88      0.92       273
     politics       0.79      0.97      0.87       274
        sport       0.90      1.00      0.95       346
```

```
          tech       0.94      0.86      0.90       261

      accuracy                           0.91      1490
     macro avg       0.91      0.90      0.90      1490
  weighted avg       0.91      0.91      0.91      1490
```

[12]:
```
keywords = {
    "business": ["economy", "market", "growth", "bank", "sales", "economic"],
    "tech": ["technology", "microsoft", "digital", "phone", "mobile",␣
 ↪"software"],
    "sport": ["match", "game", "league", "win", "champion", "players", "score",␣
 ↪"club"],
    "politics": ["government", "election", "minister", "party", "blair",␣
 ↪"labour"],
    "entertainment": ["music", "film", "actor", "album", "movie", "festival",␣
 ↪"chart"]
}

feature_names = vectorizer.get_feature_names_out()
topic_counts = [5, 10, 15, 20]
results = []

def get_category(words):
    match = None
    score = 0
    for cat, kws in keywords.items():
        overlap = len(set(words) & set(kws))
        if overlap > score:
            score = overlap
            match = cat
    return match or "unknown"

for k in topic_counts:
    nmf = NMF(n_components=k, random_state=42)
    W_tmp = nmf.fit_transform(X_tfidf)
    H_tmp = nmf.components_

    topic_map = {}
    for idx, row in enumerate(H_tmp):
        words = [feature_names[i] for i in row.argsort()[-10:][::-1]]
        topic_map[idx] = get_category(words)

    temp = df.copy()
    temp["topic"] = W_tmp.argmax(axis=1)
    temp["predicted_category"] = temp["topic"].map(topic_map)
    temp = temp.dropna(subset=["predicted_category"])
```

```
    actual = temp["Category"]
    predicted = temp["predicted_category"]

    acc = accuracy_score(actual, predicted)
    results.append(acc)
    print(f"{k} topics -> accuracy: {acc:.4f}")

plt.plot(topic_counts, results, marker="o")
plt.xlabel("Topics")
plt.ylabel("Accuracy")
plt.title("Accuracy vs. Number of Topics")
plt.grid()
plt.tight_layout()
plt.show()
```

```
5 topics -> accuracy: 0.9201
10 topics -> accuracy: 0.8866
15 topics -> accuracy: 0.7611
20 topics -> accuracy: 0.7013
```



To explore whether we can improve topic modeling performance, I experimented with a differ-

ent feature extraction method. Instead of using TF-IDF, I tried CountVectorizer, which represents documents based on the raw count of terms. While TF-IDF downweights common words, CountVectorizer keeps all frequencies, which can sometimes help NMF capture more distinct topic structures.

I used the same NMF modeling and topic-to-category keyword mapping process as before, and evaluated accuracy on the full dataset. This variation helps determine whether TF-IDF was the optimal choice for our unsupervised model, or if simpler counts might perform just as well or better.

[13]:
```
count_vec = CountVectorizer(max_features=5000)
X_counts = count_vec.fit_transform(df["clean_text"])

k = 10
nmf_alt = NMF(n_components=k, random_state=42)
W_alt = nmf_alt.fit_transform(X_counts)
H_alt = nmf_alt.components_

terms = count_vec.get_feature_names_out()
mapping = {}

for idx, row in enumerate(H_alt):
    words = [terms[i] for i in row.argsort()[-10:][::-1]]
    mapping[idx] = get_category(words)

df["topic_count"] = W_alt.argmax(axis=1)
df["predicted_category_count"] = df["topic_count"].map(mapping)

subset = df.dropna(subset=["predicted_category_count"])
actual = subset["Category"]
predicted = subset["predicted_category_count"]

score = accuracy_score(actual, predicted)
print("CountVectorizer + NMF accuracy:", score)
```

```
CountVectorizer + NMF accuracy: 0.7563758389261745
```

1.3 Step 3: Compare with supervised learning

To evaluate how supervised learning performs on the same task, I trained a logistic regression model using the same TF-IDF features. The model was trained on 80 percent of the data and evaluated on the remaining 20 percent. As expected, it achieved higher accuracy compared to the unsupervised approach which was around 97 percent on the test set. The classification report showed near-perfect performance on several categories, confirming that labeled training data provides a strong advantage.

To explore how the model performs with less data, I repeated the training using 10, 20, 50, and 100 percent of the labeled training set. I kept the test set fixed for fair comparison. The accuracy improved steadily as more data was used, starting around 88 percent with just 10 percent of the data and reaching over 97 percent with the full training set. This exercise highlighted that even

with a small fraction of labeled data, the supervised model outperformed the unsupervised one.

I plotted these results to show the relationship between training size and accuracy. The curve suggests diminishing returns after a certain point, which is typical in text classification problems. Overall, the supervised approach was more accurate and consistent, while the unsupervised method provided a surprisingly strong baseline without using any labels.

[14]:
```
X = X_tfidf
y = df["Category"]

X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, stratify=y,␣
 ↪random_state=42)

model = LogisticRegression(max_iter=1000)
model.fit(X_tr, y_tr)

train_preds = model.predict(X_tr)
test_preds = model.predict(X_te)

print("Train acc:", accuracy_score(y_tr, train_preds))
print("Test acc:", accuracy_score(y_te, test_preds))
print()
print(classification_report(y_te, test_preds))
```

```
Train acc: 0.99748322147651
Test acc: 0.9731543624161074

              precision    recall  f1-score   support

     business       0.94      0.99      0.96        67
entertainment       0.98      1.00      0.99        55
     politics       1.00      0.95      0.97        55
        sport       0.99      1.00      0.99        69
         tech       0.96      0.92      0.94        52

     accuracy                           0.97       298
    macro avg       0.97      0.97      0.97       298
 weighted avg       0.97      0.97      0.97       298
```

[15]:
```
fractions = [0.1, 0.2, 0.5, 1.0]
scores = []

for f in fractions:
    if f < 1.0:
        X_part, _, y_part, _ = train_test_split(X_tr, y_tr, train_size=f,␣
 ↪stratify=y_tr, random_state=42)
    else:
```

```
        X_part = X_tr
        y_part = y_tr

    model = LogisticRegression(max_iter=1000)
    model.fit(X_part, y_part)

    preds = model.predict(X_te)
    score = accuracy_score(y_te, preds)
    scores.append(score)

    print(f"{int(f * 100)}% train -> test acc: {score:.4f}")
```

```
10% train -> test acc: 0.8826
20% train -> test acc: 0.9228
50% train -> test acc: 0.9564
100% train -> test acc: 0.9732
```

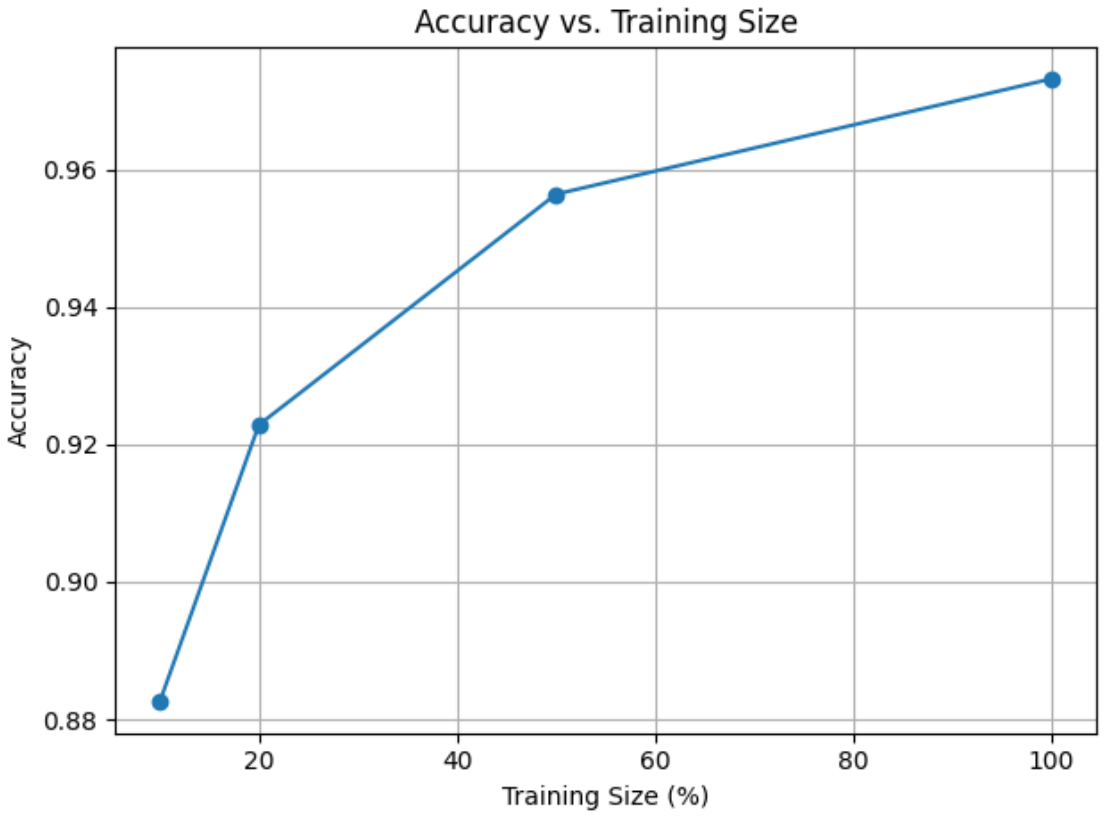
The supervised logistic regression model achieved significantly higher accuracy than the unsupervised NMF approach. With 100 percent of the training data, the supervised model reached over 97 percent accuracy on the test set, while the unsupervised approach peaked around 91 percent.

To understand data efficiency, I trained the supervised model using smaller subsets of the training data. Even with just 10 percent of the training set, the model achieved nearly 88 percent accuracy, which was already competitive with the unsupervised method. This shows that supervised learning is more data-efficient and benefits greatly from even limited labeled data.

There were no strong signs of overfitting, as the test accuracy continued to improve with more data and the training accuracy was consistently high. In contrast, the unsupervised model performance plateaued early and was more sensitive to the number of topics, which required careful tuning.

Overall, supervised learning clearly outperformed unsupervised matrix factorization in both accuracy and robustness, especially when labeled data was available.

[16]:
```
plt.plot([int(f * 100) for f in fractions], scores, marker="o")
plt.title("Accuracy vs. Training Size")
plt.xlabel("Training Size (%)")
plt.ylabel("Accuracy")
plt.grid()
plt.tight_layout()
plt.show()
```



1.4 References

- Scikit-learn documentation: https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction
- NLTK Stopwords: https://www.nltk.org/book/ch02.html
- TF-IDF overview: https://en.wikipedia.org/wiki/Tf–idf