Business problem:

- The goal of this project is to use a bunch of news articles extracted from the companies' internal database and categorize them into several categories like politics, technology, sports, business and entertainment based on their content.

Data load and basic non-graphical EDA

```
from google.colab import drive
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import ttest_ind
from scipy.stats import ttest_rel
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from statsmodels.graphics.gofplots import qqplot
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.compat import lzip
import statsmodels.stats.api as sms
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    classification_report,
    confusion_matrix
)
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
import string
# Setting up the NLTK to pre-processing textual data
import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import PorterStemmer, LancasterStemmer, SnowballStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize, sent_tokenize
nltk.download('punkt') # What does this do? nltk.download('punkt') downloads the necessary data files for NLTK's Punkt sentence tokenizer, enabling you to use functions like nltk.sent_tokenize() to accurately split text into individual sentences. You typically only need to run this command once per Python environment where you intend to use NLTK's sentence tokenization.
nltk.download('stopwords') # What does this do? nltk.download('stopwords') downloads the list of common stop words (like "and", "the", etc.) used in various languages, which can be used to filter out these words from your text data during preprocessing.
nltk.download('wordnet') # What does this do? nltk.download('wordnet') downloads the WordNet lexical database, which is used for tasks like lemmatization and synonym lookup in natural language processing.
nltk.download('averaged_perceptron_tagger') # What does this do? nltk.download('averaged_perceptron_tagger') downloads the pre-trained part-of-speech tagger model, which is used to assign grammatical tags (like noun, verb, etc.) to words in a sentence.
nltk.download('tagsets') # What does this do? nltk.download('tagsets') downloads the tagset documentation for the part-of-speech tags used in NLTK, providing a reference for understanding the tags assigned by the tagger.
nltk.download('universal_tagset') # What does this do? nltk.download('universal_tagset') downloads the universal tageset, which is a simplified set of part-of-speech tags that can be used across different languages, making it easier to understand and work with POS tagging in NLTK.
nltk.download('treebank')        # What does this do? nltk.download('treebank') downloads the Treebank corpus, which is a collection of annotated text used for training and evaluating natural language processing models, particularly in tasks like part-of-speech tagging and parsing.
nltk.download('punkt_tab')


sns.set_theme(style="darkgrid")
pd.set_option("display.max_columns", 100)
%matplotlib inline

import re
import spacy
import contractions
```

```
!pip install contractions
!pip install -q gdown
!gdown --id 1-LWa9iJf6m2PqNWXY8vvKB9JzpHElHcc -O Flipitnews-data.csv
```

```
from google.colab import drive

pd.set_option('display.max_columns', None)

data = pd.read_csv('Flipitnews-data.csv')
data.head()
```

```
data.info()
```

```
data.describe(include = 'object')
```

```
data.drop_duplicates(inplace = True)
data.reset_index(inplace = True, drop = True)
```

```
data.info()
```

```
data
```

```
data[data.Category == 'Technology'].shape
```

```
data[data.Category == 'Sports'].shape
```

```
sns.countplot(data['Category'])
```

Observations:

There are 2126 rows in the data

There are in total 5 unique categories thata article belongs to.

The maximum number of Articles(500 Articles) belong to Category Business and Sports, followed by Politics. Category Technology beloongs to least number of Articles i.e. around 350 Articles.

Text preprocessing

- Convert to lowercase
- Text lemmatization
- Remove stop words
- Remove all non-letter characters
- Encoding target variable
- Remove words with 3 characters or less

```
def process_sentence(sentence, nlp_object):
    # Convert to lowercase
    sentence = sentence.lower()

    # Expanding contractions
    sentence = contractions.fix(sentence)

    # Lemmatization and removing stopwords
    doc = nlp_object(sentence)
    sentence = " ".join([token.lemma_ for token in doc if not token.is_stop])

    # Remove all non-letter characters
    sentence = re.sub(r'[^a-zA-Z\s]', ' ', sentence)  # Keep only letters and whitespace
    sentence = re.sub(r'\s+', ' ', sentence)  # Normalize whitespace

    return sentence.strip()
```

```
from tqdm.notebook import tqdm
# tqdm to see real time progress
tqdm.pandas()

nlp = spacy.load('en_core_web_sm') # English pipeline optimized for CPU

def process_article(article_text, nlp_object):
    processed_article_sentences = []
    # using nltk sentence tokenizer
    for sentence in sent_tokenize(article_text):
        # preprocessing each sentence using our process_sentence function
        processed_article_sentences.append(process_sentence(sentence, nlp_object))
    # joining preprocessed sentence as a complete paragrams of the article
    return " ".join(processed_article_sentences)

data["processed_Article"] = data["Article"].progress_apply(lambda x : process_article(x, nlp))
```

```
data.head()
```

```
print('Article before preprocessing: ',data.Article.loc[0])
print(' ')
print('Article after preprocessing: ',data.processed_Article.loc[0])
```

```
le = LabelEncoder()

# Fit and transform the 'Category' column
data['Category_encoded'] = le.fit_transform(data['Category'])
data.head()
```

```
data['tokens'] = data['processed_Article'].apply(word_tokenize)
```

```
data.head()
```

```
data['processed_Article'] = data['processed_Article'].apply(
    lambda text: ' '.join([word for word in text.split() if len(word) > 3])
)

data = data.dropna(subset=['processed_Article'])
data.reset_index(inplace= True, drop = True)
```

Vectorize the text using either BOW or tfidf

```
def vectorize_text_data(df, text_column, method='tfidf'):
    """
    Vectorizes text data using Bag of Words or TF-IDF.

    Parameters:
        df (pd.DataFrame): DataFrame containing text data.
        text_column (str): Column name with the text to vectorize.
        method (str): 'bow' for Bag of Words, 'tfidf' for TF-IDF.

    Returns:
        pd.DataFrame: Vectorized representation of text.
    """
    if method == 'bow':
        vectorizer = CountVectorizer()
    elif method == 'tfidf':
        vectorizer = TfidfVectorizer()
    else:
        raise ValueError("Invalid method. Choose 'bow' or 'tfidf'.")

    X = vectorizer.fit_transform(df[text_column])
    vectorized_df = pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names_out())
    return vectorized_df
```

With BOW embeddings

```
vectorized_df = vectorize_text_data(data, text_column='processed_Article', method='bow')  # or method='bow'
vectorized_df.head()
```

```
data.head()
```

```
vectorized_df_1 = pd.concat([vectorized_df, data['Category_encoded'].reset_index(drop=True)], axis=1)
vectorized_df_1.head()
```

Train-test split

```
# Assume vectorized_df has features and 'Category_encoded'
X = vectorized_df_1.drop(columns=['Category_encoded'])
y = vectorized_df_1['Category_encoded']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

```
# Assume vectorized_df has features and 'Category_encoded'
X_dummy = vectorized_df_1.drop(columns=['Category_encoded'])
y_dummy = vectorized_df_1['Category_encoded']

# Train-test split
X_train_dummy, X_test_dummy, y_train_dummy, y_test_dummy = train_test_split(
    X_dummy, y_dummy, test_size=0.25, random_state=42
)
```

```
X_train_dummy.shape, X_test_dummy.shape
```

Training the classifier model

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, classification_report, confusion_matrix
)
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# ========================
# STEP 1: Split the dataset
# ========================
def split_data(df, target_col='Category_encoded', test_size=0.2):
    X = df.drop(columns=[target_col])
    y = df[target_col]
    return train_test_split(X, y, test_size=test_size, random_state=42)

# ====================================
# STEP 2: Evaluate Model on Given Dataset
# ====================================
def evaluate_model(model, X, y, dataset_name='Dataset'):
    y_pred = model.predict(X)
    acc = accuracy_score(y, y_pred)
    prec = precision_score(y, y_pred, average='weighted', zero_division=0)
    rec = recall_score(y, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y, y_pred, average='weighted', zero_division=0)

    print(f"\n📊 Metrics for {dataset_name}:")
    print(f"✅ Accuracy:  {acc:.4f}")
    print(f"✅ Precision: {prec:.4f}")
    print(f"✅ Recall:    {rec:.4f}")
    print(f"✅ F1-score:  {f1:.4f}")
    print("\n📄 Classification Report:\n", classification_report(y, y_pred, zero_division=0))

    cm = confusion_matrix(y, y_pred)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Purples')
    plt.title(f"{dataset_name} - Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.tight_layout()
    plt.show()

# =============================
# STEP 3: Train & Evaluate Model
# =============================
def train_and_evaluate_model(model, param_grid, model_name, X_train, y_train, X_test, y_test):
    print(f"\n🔧 Training {model_name}...")
    grid = GridSearchCV(model, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
    grid.fit(X_train, y_train)
    best_model = grid.best_estimator_
    print(f"✅ Best Parameters for {model_name}:", grid.best_params_)

    # Evaluate on Train and Test
    evaluate_model(best_model, X_train, y_train, dataset_name=f"{model_name} - Train Set")
    evaluate_model(best_model, X_test, y_test, dataset_name=f"{model_name} - Test Set")

# ===================
# STEP 4: Run Everything
# ===================

# 1. Naive Bayes
train_and_evaluate_model(
    MultinomialNB(),
    param_grid={'alpha': [0.01, 0.1, 0.5, 1.0, 2.0, 5.0]},
    model_name='Multinomial Naive Bayes',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 2. Decision Tree
train_and_evaluate_model(
    DecisionTreeClassifier(random_state=42),
    param_grid={'max_depth': [5, 10, 20, None], 'min_samples_split': [2, 5, 10]},
    model_name='Decision Tree',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 3. K-Nearest Neighbors
train_and_evaluate_model(
    KNeighborsClassifier(),
    param_grid={'n_neighbors': [3, 5, 7], 'weights': ['uniform', 'distance']},
    model_name='K-Nearest Neighbors',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 4. Random Forest
train_and_evaluate_model(
    RandomForestClassifier(random_state=42),
    param_grid={'n_estimators': [50, 100], 'max_depth': [None, 10], 'min_samples_split': [2, 5]},
    model_name='Random Forest',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)
```

✅ 1. Multinomial Naive Bayes Train Accuracy: 99.65% Test Accuracy: 97.42%

🔍 Observations:

Strong generalization: Only a ~2.2% drop from train to test, indicating minimal overfitting.

Consistent performance across all classes — especially class 3 and 1 with near-perfect recall and F1.

Slight performance dip on class 0 and 4 in test set but still within high-performance range.

Ideal for high-dimensional, sparse data such as text classification with TF-IDF or Bag-of-Words.

🧠 Verdict: ✅ Reliable and efficient model — extremely strong for baseline or even production use in text-based pipelines. Lightweight yet powerful.

⚠️ 2. Decision Tree Train Accuracy: 98.82% Test Accuracy: 84.27%

🔍 Observations:

Overfitting evident: Nearly perfect on training set but a ~14.5% drop on test performance.

Precision and recall are reasonable but vary significantly between classes.

Class 3 is well-handled; class 0 and 4 underperform on test set, contributing to the drop.

Despite hyperparameter tuning (min_samples_split=5), lack of max_depth caused deep tree growth.

🧠 Verdict: ⚠️ Overfit model — useful for interpretability or as a base estimator but not reliable without pruning or regularization.

❌ 3. K-Nearest Neighbors (KNN) Train Accuracy: 100.00% Test Accuracy: 74.18%

🔍 Observations:

Severe overfitting: The model perfectly memorizes training data but generalizes poorly to test set (~26% drop).

Recall is erratic — class 3 gets 100% recall, but class 4 only 51%.

Performance highly unstable across classes; influenced by class imbalance or data density.

Computationally expensive for large datasets, especially during inference.

🧠 Verdict: ❌ Not suitable for this task — overfits, unstable class-wise performance, and not scalable for production in high-dimensional settings.

✅ 4. Random Forest Train Accuracy: 100.00% Test Accuracy: 95.54%

🔍 Observations:

Strong performance and generalization — only ~4.5% test-train gap, indicating acceptable variance.

Excellent recall and F1 across all classes, especially for class 3 and 2.

Slight weakness in class 4 (recall = 0.81) suggests possible class imbalance impact.

Ensemble nature reduces variance compared to Decision Tree; generalizes much better.

🧠 Verdict: ✅ Best overall performer — robust, accurate, and less prone to overfitting. A top pick for balanced performance across all classes.

Model	Train Accuracy	Train Recall	Train F1	Test Accuracy	Test Recall	Test F1	Overfitting?	Notes
Naive Bayes	99.65%	99.65%	99.65%	97.42%	97.42%	97.41%	❌ No	Lightweight, fast, excellent generalization
Decision Tree	98.82%	98.82%	98.82%	84.27%	84.27%	84.24%	✅ Yes	Overfits, needs depth/leaf regularization
KNN	100.00%	100.00%	100.00%	74.18%	74.18%	74.41%	✅ Severe	Highly overfit, unstable generalization
Random Forest	100.00%	100.00%	100.00%	95.54%	95.54%	95.47%	⚠️ Slight	Excellent accuracy, best overall performer

With TFIDF embeddings

```
vectorized_df = vectorize_text_data(data, text_column='processed_Article', method='tfidf')  # or method='bow'
vectorized_df.head()
```

```
data.head()
```

```
vectorized_df_1 = pd.concat([vectorized_df, data['Category_encoded'].reset_index(drop=True)], axis=1)
vectorized_df_1.head()
```

Train-test split

```
# Assume vectorized_df has features and 'Category_encoded'
X = vectorized_df_1.drop(columns=['Category_encoded'])
y = vectorized_df_1['Category_encoded']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Training the classifier model

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, classification_report, confusion_matrix
)
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# ========================
# STEP 1: Split the dataset
# ========================
def split_data(df, target_col='Category_encoded', test_size=0.2):
    X = df.drop(columns=[target_col])
    y = df[target_col]
    return train_test_split(X, y, test_size=test_size, random_state=42)

# ====================================
# STEP 2: Evaluate Model on Given Dataset
# ====================================
def evaluate_model(model, X, y, dataset_name='Dataset'):
    y_pred = model.predict(X)
    acc = accuracy_score(y, y_pred)
    prec = precision_score(y, y_pred, average='weighted', zero_division=0)
    rec = recall_score(y, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y, y_pred, average='weighted', zero_division=0)

    print(f"\n📊 Metrics for {dataset_name}:")
    print(f"✅ Accuracy:  {acc:.4f}")
    print(f"✅ Precision: {prec:.4f}")
    print(f"✅ Recall:    {rec:.4f}")
    print(f"✅ F1-score:  {f1:.4f}")
    print("\n📄 Classification Report:\n", classification_report(y, y_pred, zero_division=0))

    cm = confusion_matrix(y, y_pred)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Purples')
    plt.title(f"{dataset_name} - Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.tight_layout()
    plt.show()

# =============================
# STEP 3: Train & Evaluate Model
# =============================
def train_and_evaluate_model(model, param_grid, model_name, X_train, y_train, X_test, y_test):
    print(f"\n🔧 Training {model_name}...")
    grid = GridSearchCV(model, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
    grid.fit(X_train, y_train)
    best_model = grid.best_estimator_
    print(f"✅ Best Parameters for {model_name}:", grid.best_params_)

    # Evaluate on Train and Test
    evaluate_model(best_model, X_train, y_train, dataset_name=f"{model_name} - Train Set")
    evaluate_model(best_model, X_test, y_test, dataset_name=f"{model_name} - Test Set")

# ===================
# STEP 4: Run Everything
# ===================

# 1. Naive Bayes
train_and_evaluate_model(
    MultinomialNB(),
    param_grid={'alpha': [0.01, 0.1, 0.5, 1.0, 2.0, 5.0]},
    model_name='Multinomial Naive Bayes',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 2. Decision Tree
train_and_evaluate_model(
    DecisionTreeClassifier(random_state=42),
    param_grid={'max_depth': [5, 10, 20, None], 'min_samples_split': [2, 5, 10]},
    model_name='Decision Tree',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 3. K-Nearest Neighbors
train_and_evaluate_model(
    KNeighborsClassifier(),
    param_grid={'n_neighbors': [3, 5, 7], 'weights': ['uniform', 'distance']},
    model_name='K-Nearest Neighbors',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)

# 4. Random Forest
train_and_evaluate_model(
    RandomForestClassifier(random_state=42),
    param_grid={'n_estimators': [50, 100], 'max_depth': [None, 10], 'min_samples_split': [2, 5]},
    model_name='Random Forest',
    X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test
)
```

✅ 1. Multinomial Naive Bayes Train Accuracy: 99.47% Test Accuracy: 98.36%

🔍 Observations: Strong generalization: There's only a ~1% drop from train to test, which indicates the model is not overfitting and generalizes well.

Performs consistently well across all classes.

Excellent choice for text classification tasks, especially when using bag-of-words or TF-IDF features.

🧠 Verdict: ✅ Reliable baseline model with excellent performance. Especially strong considering its simplicity.

✅ 2. Decision Tree Train Accuracy: 100.00% Test Accuracy: 83.10%

🔍 Observations: Clear overfitting: The model perfectly memorizes the training data but fails to generalize well to the test set.

Significant drop (~17%) in accuracy, precision, recall, and f1-score from train to test.

Class 3 performs well on the test set, but others show inconsistency, particularly class 0 and 4.

🧠 Verdict: ⚠️ Overfit model. Avoid using Decision Trees without pruning or regularization. Needs stronger regularization (e.g., max_depth, min_samples_leaf).

✅ 3. K-Nearest Neighbors (KNN) Train Accuracy: 95.71% Test Accuracy: 93.90%

🔍 Observations: Small drop between training and test accuracy (~1.8%), suggesting good generalization.

Performs very well on all classes — precision, recall, and f1-scores are high and balanced.

May be computationally expensive on large datasets.

🧠 Verdict: ✅ Robust performer with strong generalization. Great choice if computation cost isn't a bottleneck.

✅ 4. Random Forest Train Accuracy: 100.00% Test Accuracy: 95.31%

🔍 Observations: Like Decision Tree, Random Forest perfectly fits training data — but generalizes far better.

High accuracy and balanced metrics across all classes.

Still slightly overfitting, but acceptable due to high test performance.

🧠 Verdict: ✅ Best overall performer — powerful, high test accuracy, robust across all classes, and less prone to overfitting due to ensembling.

Model	Train Accuracy	Train Recall	Train F1	Test Accuracy	Test Recall	Test F1	Overfitting?	Notes
Naive Bayes	99.47%	99.47%	99.47%	98.36%	98.36%	98.36%	❌ No	Lightweight, fast, and well-balanced
Decision Tree	100.00%	100.00%	100.00%	83.10%	83.10%	82.99%	✅ Yes	Severely overfitting — needs pruning
KNN	95.71%	95.71%	95.71%	93.90%	93.90%	93.92%	❌ Minimal	Accurate, slower on large datasets
Random Forest	100.00%	100.00%	100.00%	95.31%	95.31%	95.23%	⚠️ Slight	Best performer overall, but large size

Questionaire

Q1. How many news articles are present in the dataset that we have?

A1. There are 2225 total and 2126 unique news articles.

Q2. Most of the news articles are from _____ category.

A2. Sports with total 501 articles after de-duplication

Q3. Only ___ no. of articles belong to the 'Technology' category.

A3. 347 articles after de-duplication

Q4. What are Stop Words and why should they be removed from the text data?

A4. Stop words are common words like "the", "is", and "in" that carry little meaningful information. Removing them helps reduce noise and improves model performance by focusing on the more important words in the text.

Q5. Explain the difference between Stemming and Lemmatization.

A5. Stemming crudely chops off word endings to reduce words to their root form (e.g., "running" → "run"), often resulting in non-real words. Lemmatization is more accurate, reducing words to their base form using vocabulary and grammar (e.g., "better" → "good").

Q6. Which of the techniques Bag of Words or TF-IDF is considered to be more efficient than the other?

A6. TF-IDF is generally more efficient than Bag of Words as it not only considers word frequency but also reduces the weight of common words across documents, making it better at highlighting important terms. Bag of Words treats all words equally, often leading to less meaningful representations.

Q7. What's the shape of train & test data sets after performing a 75:25 split.

A7. The shape of train dataset is (1594, 20858) and that of test dataset is (532, 20858) after preprocessing of data and performing the 75:25 split.

Q8. Which of the following is found to be the best performing model..a. Random Forest b. Nearest Neighbors c. Naive Bayes

A8. Naive Bayes performs the best for our given data with below train and test performance for tfidf embeddings

Model	Train Accuracy	Train Recall	Train F1	Test Accuracy	Test Recall	Test F1	Overfitting?	Notes
Naive Bayes	99.47%	99.47%	99.47%	98.36%	98.36%	98.36%	❌ No	Lightweight, fast, and well-balanced

Q9. According to this particular use case, both precision and recall are equally important. (T/F)

A9. T