Advanced Natural Language Processing (968G5) Assignment

Table of Contents

1. Introduction 2
2. Methods 3
2.1 Environment and Libraries 3
2.2 Description of Datasets 4
2.3 Data Preprocessing 4
2.3 Handling Class Imbalance 5
2.4 Hyperparameter Tuning 5
2.5 Cross-Validation 5
2.6 Evaluation 5
2.7 Task Handling and Approaches 5
2.7.1 Task 1: Propaganda Detection 5
Approach 1: Naive Bayes Model 5
Approach 2: Logistic Regression 6
2.7.2 Task 2: Propaganda Technique Classification 7
Approach 1: Multinomial Logistic Regression 7
Approach 2: LSTM with Word2Vec Embeddings 8
3. Results and Analyses 9
3.1 Comparative Evaluation of Methods for Task 1: Propaganda Detection 9
Model Performance Summary 9
Approach 1: Naive Bayes Approach with TF-IDF Vectorizer 9
Naive Bayes Confusion Matrix Analysis 10
Approach 2: Logistic Regression Approach with TF-IDF Vectorizer 11
Analysis 12
5.2 Comparative Evaluation of Methods for Task 2: Propaganda Technique Classification 12
Model Performance Summary 12
LSTM with Word2Vec 12
3.2.1 Multinomial Logistic Regression Approaches 13
3.2.2 LSTM with Word2Vec 14
Analysis 15
4. Conclusion 15
5. References 16
6. Appendix 17
Task 1: Propaganda Detection 17
Approach 1: Naive Bayes Model 17
Approach 2: Logistic Regression 21
Task 2: Propaganda Technique Classification 24
Approach 1: Multinomial Logistic Regression 24
Approach 2: LSTM with Word2Vec Embeddings 33

Techniques for Propaganda Detection and Classification in Text: A Comparative Analysis of Machine Learning and Deep Learning Approaches

---

1. Introduction

Propaganda, the strategic dissemination of biased or misleading information, has become a pervasive issue in the modern digital era, shaping public opinion and manipulating societal beliefs **[1]**. With the proliferation of online platforms and social media, the ability to detect and classify propaganda techniques has emerged as a critical task in the field of natural language processing (NLP) and information analysis. This study aims to address the problem of propaganda detection and classification through the exploration and evaluation of various machine learning and deep learning approaches.

Significant work has been done on propaganda classification in text, with one notable study employing BERT for this task. The tool is PROTECT, developed by Vorakit Vorakitphan, Elena Cabrio, and Serena Villata, designed to automatically detect and classify propaganda in texts **[2]**. PROTECT identifies propaganda snippets and classifies them into various techniques using a pipeline architecture. The system leverages BERT and RoBERTa models for detection and classification, achieving state-of-the-art performance on standard benchmarks.

This study tackles two primary objectives: binary classification of text as propaganda or non-propaganda, and multi-class classification of specific propaganda techniques employed. The dataset used in this research is derived from the Propaganda Techniques Corpus **[9]**, containing labeled training and testing data with sentences annotated for various propaganda techniques, such as flag waving, appeal to fear, causal simplification, and loaded language, as well as instances of non-propaganda text.

The methodologies employed in this study encompass a range of techniques, each with its own theoretical underpinnings and practical applications. For the binary classification task, two main approaches are investigated: Naive Bayes and Logistic Regression. Naive Bayes, a probabilistic classifier based on Bayes' theorem, is known for its simplicity and efficiency in text classification tasks **[3]**. The study utilizes a pipeline integrating text preprocessing, TF-IDF vectorization, and classification using a Multinomial Naive Bayes model, with hyperparameters tuned for optimal performance.

Logistic Regression, a discriminative classifier that models class membership probabilities as a logistic function of features, is also explored for binary classification **[4]**. This method is particularly effective in handling imbalanced datasets and offers inherent regularization capabilities, potentially improving performance in text classification scenarios. The study employs Logistic Regression with TF-IDF vectorization and hyperparameter tuning using RandomizedSearchCV to enhance model accuracy.

For the multi-class classification task of identifying specific propaganda techniques, the study explores three approaches using Multinomial Logistic Regression with different feature representations: Word2Vec embeddings, TF-IDF vectorization, and a combination of both. These configurations aim to capture both the semantic and statistical properties of the text, potentially enhancing classification accuracy. The decision to use

multinomial logistic regression was informed by the book *Speech and Language Processing* **[4]**, which highlights the model's effectiveness in handling multiclass classification problems.

Additionally, a Long Short-Term Memory (LSTM) neural network architecture with Word2Vec embeddings is employed to capture sequential dependencies and contextual information in the text data. LSTM models are particularly well-suited for NLP tasks due to their ability to retain information over long sequences **[8]**, providing a deep learning perspective on the problem.

The report is structured to provide an overview of the environment, libraries, datasets, and preprocessing steps employed, as well as the strategies used to handle class imbalance, hyperparameter tuning, and cross-validation methods. The results and analyses section presents a comprehensive evaluation of the methods employed for both tasks, including performance metrics such as accuracy, precision, recall, and F1-score, enabling a comparative analysis of the strengths and weaknesses of each approach. The primary objective of this study is to identify the most effective approach for each task, considering not only the overall performance but also the specific requirements and characteristics of propaganda detection.

2. Methods

This section provides an overview of the environment, libraries, data preprocessing steps, handling of class imbalance, hyperparameter tuning, and cross-validation methods used in the study. It also details the approaches taken for each task, setting the stage for the presentation of results and evaluation in the following sections.

2.1 Environment and Libraries

The development and experiments for this study were conducted on Google Colab, utilizing its T4 GPU to expedite the training and evaluation processes for the models. Google Colab provides a powerful and flexible environment for developing machine learning and deep learning models due to its pre-installed libraries and GPU support.
Libraries Used:

- Natural Language Processing:
    - NLTK: The Natural Language Toolkit (NLTK) was used for basic text preprocessing tasks such as tokenization and stopword removal
    - SpaCy: SpaCy was employed for more advanced NLP tasks including part-of-speech (POS) tagging and named entity recognition (NER), which are crucial for understanding the grammatical structure and identifying important entities in the text.
- Machine Learning:
    - Scikit-learn: This library was utilized for building and evaluating machine learning models. It was also used for hyperparameter tuning and cross-validation to ensure robust model performance.
    - Imbalanced-learn: Specifically, the SMOTE (Synthetic Minority Over-sampling Technique) function from this library was used to handle class imbalance by generating synthetic examples of the minority class.
    - Gensim: Gensim's Word2Vec model was trained to generate word embeddings from the text data, which captures the semantic meaning of words and improves the performance of machine learning models.
- Deep Learning:

    - Keras and TensorFlow: These libraries were used for constructing and training deep learning models, particularly LSTM (Long Short-Term Memory) networks.
- Utilities:
    - Pandas: Employed for data manipulation and analysis, enabling efficient handling of the dataset.
    - NumPy: Used for numerical computations and array operations, essential for processing data and model outputs.
    - Matplotlib: Utilized for visualizing results and performance metrics, aiding in the analysis and presentation of the findings.

2.2 Description of Datasets

The datasets provided for this study consist of two files, one for training and one for testing, both in tab-separated-value (tsv) format. Each file contains two columns: the first column specifies the label, and the second column contains the corresponding sentence or chunk of text. The labels are drawn from a set of nine categories, including eight propaganda techniques (flag waving, appeal to fear/prejudice, causal simplification, doubt, exaggeration/minimisation, loaded language, name calling/labeling, and repetition) and one non-propaganda label ("not propaganda"). The sentences are annotated with the beginning (<BOS>) and end (<EOS>) tokens to mark the span of text associated with the given label. These datasets were obtained from the Propaganda Techniques Corpus **[9]** and were preprocessed to include tokenization, removal of stopwords, and normalization to lowercase. This structured format allows for both binary classification (propaganda vs. non-propaganda) and multi-class classification (identifying specific propaganda techniques).

2.3 Data Preprocessing

Data preprocessing is crucial for improving the performance of machine learning models. The preprocessing steps for this study included:

- Text Cleaning: Each sentence in the dataset was cleaned to remove unwanted characters and specific tokens such as <BOS> and <EOS>, which were included in the original dataset to mark the beginning and end of text spans.
- Tokenization: Sentences were tokenized into individual words using NLTK and SpaCy. This process involves breaking down the text into smaller units (tokens) that can be more easily analyzed.
- Stopword Removal: Common English stopwords, which do not contribute significant meaning to the text, were removed to reduce noise. This helps in focusing the model on the more informative parts of the text.
- POS Tagging and Entity Recognition: Using SpaCy, each token was tagged with its part of speech, and named entities were recognized. These tags and entities were then included as additional features for the models.

Custom Transformers: Custom transformers were implemented to standardize these preprocessing steps and integrate them seamlessly into the machine learning pipelines. This ensures that the same preprocessing is applied consistently across all data used in training and evaluation.

2.3 Handling Class Imbalance

Class imbalance is a common issue in machine learning, where some classes are underrepresented compared to others. This can lead to biased models that perform poorly on the minority classes. To address this, SMOTE (Synthetic Minority Over-sampling Technique) was employed **[5].** SMOTE generates synthetic examples of the

minority class by interpolating between existing examples, thus balancing the class distribution and improving model performance.
For this study, SMOTE was primarily used because the propaganda classes were less numerous than the not_propaganda classes. Although the total number of instances did not differ significantly, SMOTE was applied when necessary and omitted from the model pipelines if it did not improve the model's performance.

2.4 Hyperparameter Tuning

Hyperparameter tuning is the process of selecting the best set of parameters for a machine learning model to optimize its performance. In this study, RandomizedSearchCV was used for this purpose. This technique searches over specified parameter values by sampling a fixed number of parameter settings from the given distributions **[6]**. It was particularly useful for tuning parameters in Naive Bayes and Logistic Regression models because implementing it on an LSTM model was computationally intensive and caused the code to run for extended periods.

2.5 Cross-Validation

Cross-validation is a method for evaluating the performance of a model **[7]**. In this study, 5-fold cross-validation was used. This involves splitting the dataset into five subsets, training the model on four subsets, and validating it on the remaining subset. This process is repeated five times, with each subset serving as the validation set once. Cross-validation provides a more reliable estimate of model performance by ensuring that the model is tested on multiple different subsets of data.

2.6 Evaluation

The performance of all models was evaluated using various metrics, including accuracy, precision, recall, and F1-score, as reported in the classification report. For the Naive Bayes model, a confusion matrix was employed to further understand the classification results.

2.7 Task Handling and Approaches

2.7.1 Task 1: Propaganda Detection

This task involves classifying whether a given sentence contains propaganda. Two approaches were used. For this task, the various propaganda techniques were consolidated into a single propaganda label, resulting in a binary classification: sentences were labeled either as "propaganda" or "not_propaganda."

Approach 1: Naive Bayes Model

Pipeline: The study incorporated text preprocessing, vectorization using TfidfVectorizer, and classification using a Multinomial Naive Bayes classifier. SMOTE was employed to address class imbalance, and hyperparameters were tuned using RandomizedSearchCV. A confusion matrix was plotted at the end to compare the performance on the validation set and test set.

Design Decisions:

- Naive Bayes: Naive Bayes was chosen for its simplicity and effectiveness in text classification tasks, particularly when used with TF-IDF features. Its probabilistic nature makes it well-suited for handling

high-dimensional data **[3]**, such as text, and it performs well even with relatively small datasets. Additionally, Naive Bayes is computationally efficient, allowing for quick training and prediction.

- SMOTE (Synthetic Minority Over-sampling Technique): SMOTE was employed with a random state of 42 to address the class imbalance issue, where the propaganda class was less frequent than the non-propaganda class.

- Hyperparameters selection to be Tuned:

  vectorizer__max_features': [1000, 5000, 10000],
  'vectorizer__ngram_range': [(1, 1), (1, 2), (1, 3)],
  'classifier__alpha': [0.1, 0.5, 1.0]

In this study, hyperparameters were carefully selected and tuned to optimize the performance of a Naive Bayes classifier with text preprocessing. The `max_features` parameter in TfidfVectorizer was set to 1000, 5000, and 10000 to balance feature richness and model complexity. The `ngram_range` parameter was chosen as (1,1), (1,2), and (1,3) to capture varying levels of word context. The `alpha` parameter for the Multinomial Naive Bayes classifier was adjusted to 0.1, 0.5, and 1.0 to control smoothing. These selections were made to analyze performance across a range of parameters.

Approach 2: Logistic Regression

Pipeline: The study included text preprocessing, vectorization using TfidfVectorizer, and classification using a Logistic Regression classifier. RandomizedSearchCV was employed for hyperparameter tuning. The performance was evaluated by comparing the classification report after cross-validation with the classification report of the test set.

Design Decisions:

- Logistic regression: Logistic regression is highly effective for binary classification tasks **[4]**, particularly due to its ability to adjust decision thresholds. It often outperforms Naive Bayes, making it a preferred choice for drawing comparisons in this study.

- SMOTE was not added in this approach primarily for comparison purposes and because Logistic Regression Can handle imbalanced datasets more robustly through techniques such as class weighting and threshold adjustment. This allows it to give more attention to the minority class.

- Hyperparameters selection to be Tuned:

  max_features: 1000, 5000, 10000
  ngram_range: (1,1), (1,2)
  C: 0.1, 1.0, 10.0

The max_features parameter was adjusted to 1000, 5000, and 10000 to find the best balance between model complexity and performance, with lower values reducing overfitting and higher values capturing more nuanced

patterns. The C parameter in Logistic Regression controls the inverse of the regularization strength. Tuning this parameter helps find the optimal balance between underfitting and overfitting. 0.1 means strong regularization and 10.0 means weak regularization, allowing the model more flexibility to fit the training data closely, which can be beneficial for capturing complex patterns but increases the risk of overfitting.

2.7.2 Task 2: Propaganda Technique Classification

This task involves identifying specific propaganda techniques used in given text spans. A label encoder was used to encode the propaganda techniques in the dataset. Two approaches were employed for this task:

Approach 1: Multinomial Logistic Regression

Three configurations were explored under this approach:

a) Multinomial Logistic Regression with Word2Vec

Pipeline:

```
('word2vec_transformer', word2vec_transformer),
('smote', SMOTE(random_state=42)),
('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))
```

Design Decisions:

-Hyperparameters to be tuned:
classifier__C: [0.01, 0.1, 1, 10, 100]
classifier__max_iter: [1000, 1500]

The C parameter in Logistic Regression controls the inverse of regularization strength. By tuning this parameter across a wide range of values, from very strong regularization (0.01) to very weak regularization (100), we aim to find the optimal balance between underfitting and overfitting, ensuring the model generalizes well to unseen data.
The max_iter parameter sets the maximum number of iterations for the solver to converge. A value of 300 was tested where the model failed to converge. Increasing the iteration limit ensures that the solver has enough opportunities to converge, particularly important for complex datasets where convergence might be slower.

b) Multinomial Logistic Regression with TF-IDF

Pipeline:

```
('tfidf_vectorizer', tfidf_vectorizer),
('smote', SMOTE(random_state=42)),
('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))
```

Design Decisions:
The hyperparameters for configuration two were kept the same as in configuration one for comparative purposes.

c) Multinomial Logistic Regression with Word2Vec+TF-IDF

Pipeline:

('combined_transformer', combined_transformer),
('smote', SMOTE(random_state=42)),
('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))

Design Decisions:

The combination of Word2Vec and TF-IDF features increases the computational complexity and time required for hyperparameter tuning. Given the extensive time required, this configuration was tested with default parameters to maintain computational feasibility.

Approach 2: LSTM with Word2Vec Embeddings

Pipeline: Text data was preprocessed and tokenized, followed by embedding with Word2Vec. An LSTM model with dropout and dense layers was then used for classification, with the model compiled using the Adam optimizer and trained with a validation split.

LSTM Model Architecture and Design Decisions:

LSTM Layers: The LSTM model included two layers: the first with 128 units and the second with 64 units. The first layer with 128 units was designed to capture complex patterns in the text, while the second layer with 64 units refined these patterns, balancing model capacity and computational efficiency. Both LSTM layers incorporated a dropout rate of 20% to prevent overfitting by randomly dropping units during training, ensuring the model generalized well. Additionally, recurrent dropout was applied to the LSTM units to further regularize the model.

Dense Layer: The dense layer contained 64 units, providing sufficient capacity to learn from the features extracted by the LSTM layers. This layer introduced non-linearity, enhancing the model's learning capability. A dropout rate of 50% was applied to this layer to further prevent overfitting, especially in the final stages of the network where overfitting is more likely.

Model Training: The Adam optimizer was chosen for its adaptive learning rate and efficient handling of sparse gradients, which is particularly suitable for NLP tasks. The categorical cross entropy loss function was used for multi-class classification, effectively measuring the model's performance in predicting the correct class probabilities. Accuracy was tracked as the primary metric to evaluate the model's performance in correctly classifying the text. The model was trained for 10 epochs, creating a balance between sufficient training and computational feasibility, ensuring the model had enough iterations to learn without overfitting. A batch size of 32 was chosen to provide a good trade-off between training speed and convergence stability. Lastly, a validation split of 20% was used to monitor the model's performance on unseen data during training, helping to prevent overfitting and ensuring generalization.

3. Results and Analyses

3.1 Comparative Evaluation of Methods for Task 1: Propaganda Detection

Model Performance Summary

Models	Best Hyperparameters:	Cross-Validation Scores	Mean CV Accuracy	Validation Set Performance Accuracy	Test Set Performance Accuracy
Naive Bayes Approach with TF-IDF Vectorizer	vectorizer__ngram_range: (1, 1) vectorizer__max_features: 10000 classifier__alpha: 1.0	[0.61904762, 0.6252588, 0.5942029, 0.64182195, 0.63278008]	0.6226222691	62%	65%
Logistic Regression Approach with TF-IDF Vectorizer	vectorizer__ngram_range: (1, 1) vectorizer__max_features: 5000 classifier__C: 1.0	[0.64082687, 0.65633075, 0.71502591, 0.71761658, 0.67098446]	0.6801569131488399	67%	69%

Approach 1: Naive Bayes Approach with TF-IDF Vectorizer

The Naive Bayes approach utilized a TF-IDF Vectorizer for feature extraction and Multinomial Naive Bayes for classification. The hyperparameters were optimized using RandomizedSearchCV, and cross-validation was employed to ensure robustness. These hyperparameters suggest that unigram features were the most effective, with a limit of 10,000 features, and the smoothing parameter (alpha) for the Naive Bayes classifier was set to 1.0. Unigram features capture the relevant information for this task, while limiting the number of features helps to reduce dimensionality and noise. The alpha value of 1.0 provides a moderate amount of smoothing, preventing overfitting. The cross-validation scores indicate a moderate performance with some variance, but the model maintained a consistent accuracy around 62%. This suggests that the model is reasonably robust and can generalize to unseen data.

The validation set results showed balanced performance across both classes with an overall accuracy of 62%. The precision, recall, and F1-scores for both classes are relatively similar, indicating that the model does not have a significant bias toward either class.

Validation Set Classification Report for TfidfVectorizer:

	precision	recall	f1-score	support
not_propaganda	0.62	0.61	0.61	241
propaganda	0.62	0.63	0.62	242
accuracy			0.62	483
macro avg	0.62	0.62	0.62	483
weighted avg	0.62	0.62	0.62	483

Figure 1. Validation Set Classification report For Naive Bayes+TF-IDF Model

On the test set, the model achieved a slightly higher accuracy of 65%, indicating generalizability beyond the training data. The classification report would provide further insights into the model's performance on the test set, including precision, recall, and F1-scores for each class.

Test Set Classification Report:

	precision	recall	f1-score	support
not_propaganda	0.68	0.63	0.65	301
propaganda	0.63	0.67	0.65	279
accuracy			0.65	580
macro avg	0.65	0.65	0.65	580
weighted avg	0.65	0.65	0.65	580

Figure 2. Test Set Classification report For Naive Bayes+TF-IDF Model

Naive Bayes Confusion Matrix Analysis

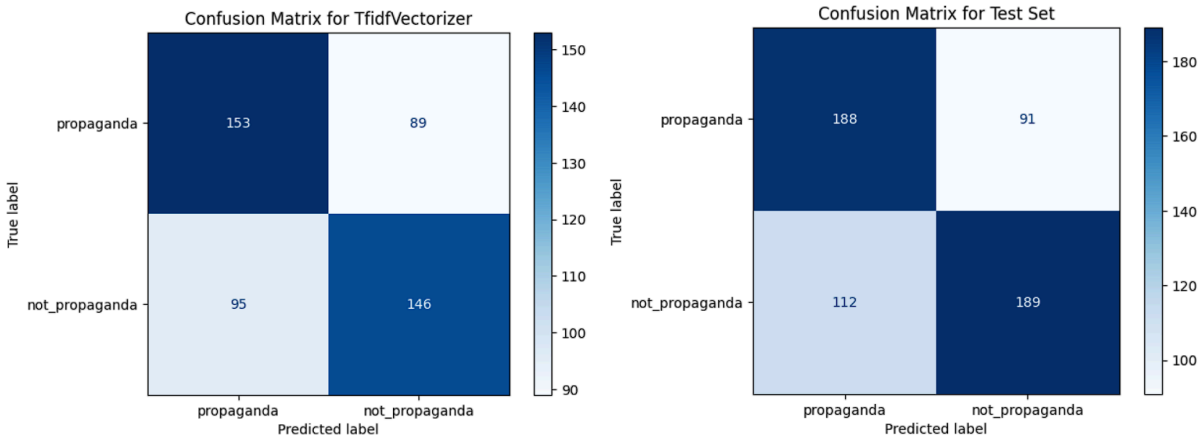

Figure 3. Confusion Matrix For Naive Bayes Model. **(a)** Confusion matrix for Validation Set (left), **(b)** Confusion matrix for test Set (right)

The confusion matrices for the Naive Bayes model using TF-IDF vectorization provide insights into the model's performance in detecting propaganda.

In the validation set, the confusion matrix reveals that the model correctly identified 153 instances of propaganda (true positives) and 146 instances of non-propaganda (true negatives). However, it also incorrectly

classified 89 instances as propaganda when they were not (false positives) and missed 95 instances of actual propaganda (false negatives). These results translate to a precision of 0.63 and a recall of 0.62 for propaganda, with an overall F1-score of 0.62. This balanced performance indicates that while the model is moderately effective, there is room for improvement, particularly in reducing the false positive and false negative rates.

The test set confusion matrix shows a similar pattern, with the model correctly identifying 188 instances of propaganda and 189 instances of non-propaganda. The false positives and false negatives were 91 and 112, respectively. The precision for propaganda improved to 0.67, and the recall was slightly lower at 0.63, resulting in an F1-score of 0.65. These results suggest that the model maintains consistent performance across different datasets, with a slight improvement in precision on the test set.

Comparing the validation and test set results, the model demonstrates a consistent ability to identify true positives and true negatives while maintaining balanced precision and recall metrics. The higher number of true positives in the test set (188 compared to 153 in the validation set) indicates an improved detection capability for propaganda. Similarly, the true negatives are also higher in the test set (189 compared to 146), suggesting better identification of non-propaganda instances.

However, the model still struggles with false negatives and false positives. The increase in false negatives in the test set (112 compared to 95) highlights a need for better recall, while the false positives remain relatively constant, indicating a persistent issue with misclassifying non-propaganda as propaganda. This balanced performance, as indicated by the closely aligned F1-scores, precision, and recall metrics across both sets, underscores the model's generalizability but also points to areas for enhancement.
In conclusion, the Naive Bayes model using TF-IDF vectorization demonstrates a reasonable performance in propaganda detection.

Approach 2: Logistic Regression Approach with TF-IDF Vectorizer

The Logistic Regression approach also utilized a TF-IDF Vectorizer for feature extraction. The cross-validation scores for Logistic Regression show higher performance compared to Naive Bayes, with an average accuracy around 68%. This improved performance can be attributed to the ability of Logistic Regression to handle correlated features and its inherent regularization capability **[4]**.

The validation set results indicate a better balance and slightly higher performance compared to the Naive Bayes model. The precision, recall, and F1-scores for both classes are relatively similar, with a slight advantage for the 'not_propaganda' class.

Validation Set Classification Report:

	precision	recall	f1-score	support
not_propaganda	0.69	0.64	0.66	242
propaganda	0.66	0.71	0.69	241
accuracy			0.67	483
macro avg	0.68	0.68	0.67	483
weighted avg	0.68	0.67	0.67	483

Figure 4. Validation Set Classification report For Logistic Regression Model

The Logistic Regression model achieved an accuracy of 69% on the test set, demonstrating improved generalizability and robustness compared to Naive Bayes. The classification report would provide further insights into the model's performance on the test set, including precision, recall, and F1-scores for each class.

```
Test Set Classification Report:
                precision    recall  f1-score   support

not_propaganda       0.72      0.66      0.69       301
    propaganda       0.66      0.72      0.69       280

      accuracy                           0.69       581
     macro avg       0.69      0.69      0.69       581
  weighted avg       0.69      0.69      0.69       581
```

Figure 5. Test Set Classification report For Logistic Regression Model

Analysis

Naive Bayes has strengths in its simplicity and computational efficiency, as well as its ability to perform well with a large number of features. However, its key weakness lies in the assumption of independence between features, which is often not the case in text data, leading to lower performance compared to Logistic Regression for this task. On the other hand, Logistic Regression's strengths include robustness to feature correlations, higher accuracy and better performance metrics, and more flexibility due to the regularization parameter. While it is computationally more intensive than Naive Bayes and requires careful tuning of hyperparameters, Logistic Regression outperformed Naive Bayes in terms of accuracy, precision, recall, and F1-score.

5.2 Comparative Evaluation of Methods for Task 2: Propaganda Technique Classification

Model Performance Summary

Model	Best Hyperparameters	Cross-Validation Scores	Mean CV F1 Score	Overall Accuracy
Multinomial Logistic Regression with Word2Vec	classifier__solver: 'lbfgs' classifier__max_iter: 1000 classifier__C: 100	[0.10828554, 0.12634774, 0.12623642, 0.13570348, 0.15668283]	0.13065120050062634	21%
Multinomial Logistic Regression with TF-IDF	classifier__max_iter: 1000 classifier__C: 0.1	[0.24802822, 0.27107111, 0.2512497, 0.26318388, 0.26558263]	0.2598231097121964	37%
Multinomial Logistic Regression with Word2Vec+TFIDF	RandomizedSearchCV not implemented	[0.23633106, 0.25589136, 0.27113702, 0.28221189, 0.2455685]	0.25822796693613465	47%
LSTM with Word2Vec	RandomizedSearchCV not implemented	Cross_val_score not Implemented	–	40%

3.2.1 Multinomial Logistic Regression Approaches

a) Word2Vec Configuration

This configuration had the lowest performance with a mean CV F1 Score of 0.13 and a test accuracy of 21%. Word2Vec captures semantic relationships between words, which is beneficial for understanding the context in text. This strength allows Word2Vec to model the meaning and relationships between words effectively. However, its performance was generally low in this study due to its reliance on context, which might not be sufficiently captured for all propaganda techniques. Additionally, Word2Vec demonstrated poor generalization on rare classes, as evidenced by low precision and recall scores, indicating that it struggled to accurately classify less common propaganda techniques.

.

Test Set Classification Report:

	precision	recall	f1-score	support
appeal_to_fear_prejudice	0.10	0.09	0.10	43
causal_oversimplification	0.08	0.16	0.11	31
doubt	0.16	0.26	0.20	38
exaggeration,minimisation	0.03	0.04	0.03	28
flag_waving	0.19	0.44	0.26	39
loaded_language	0.00	0.00	0.00	37
name_calling,labeling	0.12	0.29	0.17	31
not_propaganda	0.65	0.25	0.36	301
repetition	0.04	0.06	0.05	32
accuracy			0.21	580
macro avg	0.15	0.18	0.14	580
weighted avg	0.38	0.21	0.24	580

Figure 6. Test Set Classification report For Multinomial Logistic Regression + word2vec Model

b) TF-IDF Configuration

This configuration showed improved performance with a mean CV F1 Score of 0.26 and a test accuracy of 37%. TF-IDF likely provided better feature representation by focusing on term frequency and inverse document frequency, which helped in distinguishing between different propaganda techniques.

Test Set Classification Report:

	precision	recall	f1-score	support
appeal_to_fear_prejudice	0.24	0.28	0.26	43
causal_oversimplification	0.13	0.23	0.17	31
doubt	0.17	0.29	0.22	38
exaggeration,minimisation	0.23	0.32	0.27	28
flag_waving	0.42	0.59	0.49	39
loaded_language	0.05	0.08	0.06	37
name_calling,labeling	0.09	0.13	0.11	31
not_propaganda	0.82	0.46	0.59	301
repetition	0.20	0.28	0.23	32
accuracy			0.37	580
macro avg	0.26	0.30	0.27	580
weighted avg	0.52	0.37	0.42	580

Figure 7. Test Set Classification report For Multinomial Logistic Regression + TF-IDF Model

c) Multinomial Logistic Regression + word2vec+TF-IDF

The combination of Word2Vec and TF-IDF resulted in the highest performance among the logistic regression approaches, with a mean CV F1 Score of 0.26 and a test accuracy of 47%. This suggests that combining semantic embeddings with term frequency-based features can enhance the model's ability to capture both contextual and statistical properties of the text.

Test Set Classification Report:

	precision	recall	f1-score	support
appeal_to_fear_prejudice	0.29	0.30	0.30	43
causal_oversimplification	0.15	0.16	0.16	31
doubt	0.27	0.29	0.28	38
exaggeration,minimisation	0.22	0.29	0.25	28
flag_waving	0.49	0.56	0.52	39
loaded_language	0.05	0.08	0.06	37
name_calling,labeling	0.14	0.10	0.11	31
not_propaganda	0.74	0.65	0.69	301
repetition	0.26	0.28	0.27	32
accuracy			0.47	580
macro avg	0.29	0.30	0.29	580
weighted avg	0.50	0.47	0.48	580

Figure 8. Test Set Classification report For Multinomial Logistic Regression + word2vec+TF-IDF Model

3.2.2 LSTM with Word2Vec

The LSTM model achieved an accuracy of 40%. While LSTM can capture sequential dependencies and contextual information, the imbalance in the dataset likely affected its performance. The model performed well on the 'not_propaganda' class but struggled with other classes, indicating the need for class balancing techniques.

The classification report reveals significant challenges in the model's ability to accurately identify various propaganda techniques. The overall accuracy of 40% highlights a general difficulty in classification, particularly

with rare techniques. For instance, "appeal to fear/prejudice" and "loaded language" both show very low precision, recall, and F1-scores, indicating the model's poor performance in detecting these instances.

	precision	recall	f1-score	support
appeal_to_fear_prejudice	0.05	0.02	0.03	43
causal_oversimplification	0.17	0.19	0.18	31
doubt	0.33	0.08	0.13	38
exaggeration,minimisation	0.07	0.04	0.05	28
flag_waving	0.23	0.46	0.31	39
loaded_language	0.05	0.05	0.05	37
name_calling,labeling	0.19	0.16	0.18	31
not_propaganda	0.70	0.63	0.66	301
repetition	0.07	0.19	0.10	32
accuracy			0.40	580
macro avg	0.21	0.20	0.19	580
weighted avg	0.44	0.40	0.41	580

Figure 9. Test Set Classification report for LSTM + word2vec Model

Analysis

The logistic regression model, despite being a simpler approach, shows a moderate overall performance with an accuracy of 47%. The precision and recall for 'not_propaganda' are significantly higher, indicating better performance in identifying non-propaganda instances. However, it struggles with most other propaganda techniques, particularly 'loaded_language' and 'name_calling,labeling,' which have very low precision and recall values.

On the other hand, the LSTM model demonstrates a notably lower overall accuracy of 40%, highlighting substantial challenges in classification, especially for rare propaganda techniques. Techniques such as 'appeal to fear/prejudice' and 'loaded_language' show very low precision, recall, and F1-scores, indicating a significant difficulty in detecting these instances. While 'doubt' has relatively higher precision, its extremely low recall points to inconsistencies in correct identification. The 'not_propaganda' class, which is the most prevalent, shows better performance, underscoring the model's bias towards more frequent classes and its struggle with class imbalance. This may have resulted due to not using techniques such as SMOTE which help handle class imbalance.

4. Conclusion

In the pursuit of effective propaganda detection, this study explored and evaluated several machine learning and deep learning approaches for two critical tasks: binary classification of propaganda versus non-propaganda text, and multi-class classification of specific propaganda techniques.

For the binary classification task, the Logistic Regression approach with TF-IDF vectorization emerged as the superior method, outperforming the Naive Bayes approach in terms of accuracy, precision, recall, and F1-score. The Logistic Regression model achieved an impressive accuracy of 69% on the test set, demonstrating its robustness and generalizability. The model's ability to handle correlated features and its inherent regularization capability contributed to its strong performance in this text classification task.

The hyperparameters tuned for the Logistic Regression model played a crucial role in its success. The optimal configuration included unigram features with a feature limit of 5000, and a regularization parameter (C) of 1.0. These settings allowed the model to capture the most relevant information while reducing noise and overfitting. The cross-validation scores and consistent performance across validation and test sets further reinforced the model's reliability.

For the multi-class classification task of identifying specific propaganda techniques, the Multinomial Logistic Regression approach with a combination of Word2Vec embeddings and TF-IDF features demonstrated the highest performance. This combined approach leveraged both semantic and statistical features, enhancing the model's ability to capture contextual and term frequency-based properties of the text. Although the overall performance was lower compared to the binary classification task, the combined approach achieved a mean cross-validation F1 score of 0.26 and a test accuracy of 47%, outperforming the other configurations explored.

It is worth noting that the multi-class classification task presented a greater challenge due to the increased complexity of distinguishing between multiple propaganda techniques. The LSTM model with Word2Vec embeddings, while capable of capturing sequential dependencies and contextual information, struggled with class imbalance, highlighting the importance of incorporating class balancing techniques for improved performance. The low precision, recall, and F1-scores for most rare propaganda techniques indicate that the model lacks the ability to generalize and accurately classify these specific categories. The high support for the "not_propaganda" class skewing the model's performance further illustrates the need for improved handling of class imbalance and better feature extraction methods to enhance the detection of these nuanced propaganda techniques.

Additionally, neural networks have great potential , particularly when used with pretrained models such as BERT. This study attempted to integrate BERT as a vectorizer with LSTM and pretrained Word2vec for enhanced contextual understanding. However, due to long computing hours and limited computer RAM resources, these experiments could not be included in the final report. Ensemble methods, specifically the combination of Word2Vec and TF-IDF with Multinomial Logistic Regression, were employed for classifying propaganda techniques. Initially, LSTM was considered for this task, but as suggested by literature in chapter 9 of the book *Speech and language processing* **[8]**, neural networks tend to perform better with dense embedding layers like those provided by Word2Vec, which could not be achieved with TF-IDF alone.

Further research should explore the use of advanced pretrained models like BERT and their integration with neural networks for propaganda detection. Additionally, incorporating class balancing techniques and enhancing feature extraction methods can improve the detection of rare propaganda techniques. Experimenting with ensemble methods and leveraging more sophisticated contextual information will also be crucial in advancing the performance of these models.

In conclusion, this study has provided a comprehensive evaluation of various machine learning and deep learning approaches for propaganda detection. The Logistic Regression model with TF-IDF vectorization emerged as the most effective method for binary classification, while the Multinomial Logistic Regression approach with a combination of Word2Vec and TF-IDF features showed promise for multi-class classification.

5. References

[1] Haavard Koppang. 2009. Social influence by manipulation:A definition and case of propaganda. Middle East Critique, 18:117 – 143.

[2] Vorakit Vorakitphan, Elena Cabrio, Serena Villata. PROTECT: A Pipeline for Propaganda Detection and Classification. CLiC-it 2021- Italian Conference on Computational Linguistics, Jan 2022, Milan, Italy. pp.352-358, 10.4000/books.aaccademia.10884. hal-03417019v2

[3] Singh, N. (n.d.). Naive Bayes Classifier : Definition, Applications and Examples. Analytics Vidhya. Retrieved May 20, 2024, from https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/

[4] Jurafsky, D., & Martin, J. H. (2023). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition (3rd ed., Chapter 5) [Draft of January 7, 2023]. Stanford University; University of Colorado at Boulder.

[5] Brownlee, J. (2021, March 17). SMOTE for Imbalanced Classification with Python - MachineLearningMastery.com. Machine Learning Mastery. Retrieved May 20, 2024, from https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/

[6] sklearn.model_selection.RandomizedSearchCV — scikit-learn 1.4.2 documentation. (n.d.). Scikit-learn. Retrieved May 20, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

[7] sklearn.model_selection.cross_val_score — scikit-learn 1.4.2 documentation. (n.d.). Scikit-learn. Retrieved May 20, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

[8] Jurafsky, D., & Martin, J. H. (2023). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition (3rd ed., Chapter 9) [Draft of January 7, 2023]. Stanford University; University of Colorado at Boulder.

[9] Giovanni Da San Martino, Alberto Barr´on-Cede˜no, Henning Wachsmuth, Rostislav Petrov, and Preslav Nakov. 2020. SemEval-2020 task 11: Detection of propaganda techniques in news articles.

6. Appendix

Report Word count: 4937

Code

Task 1: Propaganda Detection

Approach 1: Naive Bayes Model

```
# Install necessary libraries
!pip install nltk spacy scikit-learn
!python -m spacy download en_core_web_sm
```

```
import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV, cross_val_score
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline


# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')


import matplotlib.pyplot as plt

# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()

# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])

# Remove the first row of both training and testing data
train_data = train_data.drop(0).reset_index(drop=True)
test_data = test_data.drop(0).reset_index(drop=True)

# Display the first few rows of the training data
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Convert labels to binary: 'propaganda' or 'not_propaganda'
train_data['binary_label'] = train_data['label'].apply(lambda x: 'not_propaganda' if x ==
'not_propaganda' else 'propaganda')
test_data['binary_label'] = test_data['label'].apply(lambda x: 'not_propaganda' if x ==
'not_propaganda' else 'propaganda')
```

```
# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()

# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])

# Verify the new columns
print("\nTraining Data with Cleaned Sentences:")
print(train_data.head(10))
print(train_data.columns)

print("\nTesting Data with Cleaned Sentences:")
print(test_data.head(10))
print(test_data.columns)

# Define the hyperparameter grid for TfidfVectorizer
param_grid = {
    'vectorizer__max_features': [1000, 5000, 10000],
```

```
    'vectorizer__ngram_range': [(1, 1), (1, 2), (1, 3)],
    'classifier__alpha': [0.1, 0.5, 1.0]
}

# Define a function to create a pipeline and perform hyperparameter tuning and evaluation
def evaluate_model(vectorizer):
    # Create a pipeline with SMOTE and Naive Bayes
    pipeline = ImbPipeline(steps=[
        ('preprocessor', text_preprocessor),
        ('vectorizer', vectorizer),
        ('smote', SMOTE(random_state=42)),
        ('classifier', MultinomialNB())
    ])

    # Create RandomizedSearchCV object
    random_search = RandomizedSearchCV(pipeline, param_distributions=param_grid, n_iter=10,
scoring='accuracy', cv=5, random_state=42)

    # Split the training data into training and testing sets
    X_train, X_val, y_train, y_val = train_test_split(train_data['cleaned_sentence'],
train_data['binary_label'], test_size=0.2, random_state=42)

    # Fit RandomizedSearchCV to the training data
    random_search.fit(X_train, y_train)

    # Print the best hyperparameters found
    print(f"Best Hyperparameters for {vectorizer.__class__.__name__}:",
random_search.best_params_)

    # Apply cross-validation to the entire training set using the best pipeline
    cv_scores = cross_val_score(random_search.best_estimator_, train_data['cleaned_sentence'],
train_data['binary_label'], cv=5, scoring='accuracy')

    # Print the cross-validation scores
    print(f"Cross-Validation Scores for {vectorizer.__class__.__name__}:", cv_scores)
    print(f"Mean CV Accuracy for {vectorizer.__class__.__name__}:", cv_scores.mean())

    # Predict the labels on the validation set using the best estimator from RandomizedSearchCV
    val_predictions = random_search.best_estimator_.predict(X_val)

    # Print out the classification report for the validation set
    print(f"Validation Set Classification Report for {vectorizer.__class__.__name__}:")
    print(classification_report(y_val, val_predictions))

    # Compute and plot the confusion matrix for the validation set
    cm = confusion_matrix(y_val, val_predictions, labels=['propaganda', 'not_propaganda'])
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['propaganda',
'not_propaganda'])
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f"Confusion Matrix for {vectorizer.__class__.__name__}")
    plt.show()

    return random_search.best_estimator_



# Evaluate using TfidfVectorizer (TF-IDF)
best_tfidf_model = evaluate_model(TfidfVectorizer())
```

```
#predict model on the test set
test_predictions = best_tfidf_model.predict(test_data['cleaned_sentence'])

# Print out the classification report for the test set
print("Test Set Classification Report:")
print(classification_report(test_data['binary_label'], test_predictions))

# Compute and plot the confusion matrix for the test set
cm = confusion_matrix(test_data['binary_label'], test_predictions, labels=['propaganda',
'not_propaganda'])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['propaganda',
'not_propaganda'])
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix for Test Set")
plt.show()
```

Approach 2: Logistic Regression

```
# Install necessary libraries
!pip install nltk spacy scikit-learn
!python -m spacy download en_core_web_sm

import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')


# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()

# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])


# Display the first few rows of the training data
```

```
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Convert labels to binary: 'propaganda' or 'not_propaganda'
train_data['binary_label'] = train_data['label'].apply(lambda x: 'not_propaganda' if x ==
'not_propaganda' else 'propaganda')
test_data['binary_label'] = test_data['label'].apply(lambda x: 'not_propaganda' if x ==
'not_propaganda' else 'propaganda')


# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()

# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
```

```
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])


# Define the hyperparameter grid for RandomizedSearchCV
param_grid = {
    'vectorizer__max_features': [1000, 5000, 10000],
    'vectorizer__ngram_range': [(1, 1), (1, 2)],
    'classifier__C': [0.1, 1.0, 10.0]
}

# Create a pipeline with text preprocessing, TF-IDF vectorization, and Logistic Regression
pipeline = Pipeline([
    ('vectorizer', TfidfVectorizer()),
    ('classifier', LogisticRegression(max_iter=300))
])

# Create RandomizedSearchCV object
random_search = RandomizedSearchCV(pipeline, param_distributions=param_grid, n_iter=10,
scoring='accuracy', cv=5, random_state=42)

# Split the training data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(train_data['cleaned_sentence'],
train_data['binary_label'], test_size=0.2, random_state=42)

# Fit RandomizedSearchCV to the training data
random_search.fit(X_train, y_train)

# Print the best hyperparameters found
print("Best Hyperparameters:", random_search.best_params_)

# Apply cross-validation to the entire training set using the best pipeline
cv_scores = cross_val_score(random_search.best_estimator_, X_train, y_train, cv=5,
scoring='accuracy')

# Print the cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Accuracy:", cv_scores.mean())

# Predict the labels on the validation set using the best estimator from RandomizedSearchCV
val_predictions = random_search.best_estimator_.predict(X_val)

# Print out the classification report for the validation set
print("Validation Set Classification Report:")
print(classification_report(y_val, val_predictions))


# Use the best model to predict on the test set
test_predictions = random_search.best_estimator_.predict(test_data['cleaned_sentence'])


# Print out the classification report for the test set
print("Test Set Classification Report:")
print(classification_report(test_data['binary_label'], test_predictions))
```

Task 2: Propaganda Technique Classification

Approach 1: Multinomial Logistic Regression

Configuration 1: Multinomial Logistic Regression with Word2Vec

```
import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from gensim.models import Word2Vec
import numpy as np

# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')

# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()


# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])

# Remove the first row of both training and testing data
train_data = train_data.drop(0).reset_index(drop=True)
test_data = test_data.drop(0).reset_index(drop=True)

# Display the first few rows of the training data
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Encode labels as integers representing each of the nine classes
```

```
label_encoder = LabelEncoder()
train_data['encoded_label'] = label_encoder.fit_transform(train_data['label'])
test_data['encoded_label'] = label_encoder.transform(test_data['label'])

# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()


# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])

# Verify the new columns
print("\nTraining Data with Cleaned Sentences:")
print(train_data.head(10))
print(train_data.columns)

print("\nTesting Data with Cleaned Sentences:")
print(test_data.head(10))
```

```
print(test_data.columns)

# Train a Word2Vec model on the cleaned sentences
sentences = [sentence.split() for sentence in train_data['cleaned_sentence']]
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Custom transformer to generate Word2Vec embeddings
class Word2VecTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, word2vec_model):
        self.word2vec_model = word2vec_model

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        word2vec_features = np.array([
            np.mean([self.word2vec_model.wv[word] for word in sentence.split() if word in
self.word2vec_model.wv]
                    or [np.zeros(self.word2vec_model.vector_size)], axis=0)
            for sentence in X
        ])
        return word2vec_features

# Initialize the custom transformer
word2vec_transformer = Word2VecTransformer(word2vec_model)

pipeline = ImbPipeline(steps=[
    ('word2vec_transformer', word2vec_transformer),
    ('smote', SMOTE(random_state=42)),
    ('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))
])

# Prepare data for the pipeline
X_train = train_data['cleaned_sentence']
y_train = train_data['encoded_label']
X_test = test_data['cleaned_sentence']
y_test = test_data['encoded_label']

# Define the parameter grid for RandomizedSearchCV
param_grid = {

    'classifier__C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'classifier__max_iter': [1000,1500]  # Different maximum iterations
}

# Perform randomized search for hyperparameter tuning
random_search = RandomizedSearchCV(pipeline, param_distributions=param_grid, n_iter=10, cv=5,
scoring='f1_macro', random_state=42)
random_search.fit(X_train, y_train)

# Print the best hyperparameters found
print("Best Hyperparameters:", random_search.best_params_)

# Apply cross-validation to the entire training set using the best pipeline
cv_scores = cross_val_score(random_search.best_estimator_, X_train, y_train, cv=5,
scoring='f1_macro')

# Print the cross-validation scores
print("Cross-Validation Scores:", cv_scores)
```

```
print("Mean CV F1 Score:", cv_scores.mean())


# Use the best model to predict on the test set
best_model = random_search.best_estimator_
test_predictions = best_model.predict(X_test)


# Print out the classification report for the test set
print("Test Set Classification Report:")
print(classification_report(y_test, test_predictions, target_names=label_encoder.classes_))

Result Without Cross Validation

# Print out the classification report
print(classification_report(y_test, predictions, target_names=label_encoder.classes_))
```

Configuration 2: Multinomial Logistic Regression with TF-IDF

```
import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split, RandomizedSearchCV, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
import numpy as np

# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')

# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()


# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])

# Remove the first row of both training and testing data
```

```
train_data = train_data.drop(0).reset_index(drop=True)
test_data = test_data.drop(0).reset_index(drop=True)

# Display the first few rows of the training data
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Encode labels as integers representing each of the nine classes
label_encoder = LabelEncoder()
train_data['encoded_label'] = label_encoder.fit_transform(train_data['label'])
test_data['encoded_label'] = label_encoder.transform(test_data['label'])

# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()
```

```
# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])

# Verify the new columns
print("\nTraining Data with Cleaned Sentences:")
print(train_data.head(10))
print(train_data.columns)

print("\nTesting Data with Cleaned Sentences:")
print(test_data.head(10))
print(test_data.columns)

# Initialize the TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 3))

# Create a pipeline with TF-IDF and Logistic Regression
pipeline = ImbPipeline(steps=[
    ('tfidf_vectorizer', tfidf_vectorizer),
    ('smote', SMOTE(random_state=42)),
    ('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))
])

# Prepare data for the pipeline
X_train = train_data['cleaned_sentence']
y_train = train_data['encoded_label']
X_test = test_data['cleaned_sentence']
y_test = test_data['encoded_label']

# Define the parameter grid for RandomizedSearchCV
param_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'classifier__max_iter': [1000, 1500]  # Different maximum iterations
}

# Perform randomized search for hyperparameter tuning
random_search = RandomizedSearchCV(pipeline, param_distributions=param_grid, n_iter=10, cv=5,
scoring='f1_macro', random_state=42)
random_search.fit(X_train, y_train)

# Print the best hyperparameters found
print("Best Hyperparameters:", random_search.best_params_)

# Apply cross-validation to the entire training set using the best pipeline
cv_scores = cross_val_score(random_search.best_estimator_, X_train, y_train, cv=5,
scoring='f1_macro')

# Print the cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean CV F1 Score:", cv_scores.mean())


# Use the best model to predict on the test set
best_model = random_search.best_estimator_
test_predictions = best_model.predict(X_test)


# Print out the classification report for the test set
```

```
print("Test Set Classification Report:")
print(classification_report(y_test, test_predictions, target_names=label_encoder.classes_))
```

Configuration 3: Multinomial Logistic Regression with Word2Vec+TF-IDF

```
import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from gensim.models import Word2Vec
import numpy as np

# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')

# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()


# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])

# Remove the first row of both training and testing data
train_data = train_data.drop(0).reset_index(drop=True)
test_data = test_data.drop(0).reset_index(drop=True)

# Display the first few rows of the training data
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Encode labels as integers representing each of the nine classes
```

```
label_encoder = LabelEncoder()
train_data['encoded_label'] = label_encoder.fit_transform(train_data['label'])
test_data['encoded_label'] = label_encoder.transform(test_data['label'])

# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()


# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])

# Verify the new columns
print("\nTraining Data with Cleaned Sentences:")
print(train_data.head(10))
print(train_data.columns)

print("\nTesting Data with Cleaned Sentences:")
print(test_data.head(10))
```

```
print(test_data.columns)

# Train a Word2Vec model on the cleaned sentences
sentences = [sentence.split() for sentence in train_data['cleaned_sentence']]
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Custom transformer to combine TF-IDF features with Word2Vec embeddings
class TfidWord2VecTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, word2vec_model):
        self.word2vec_model = word2vec_model
        self.vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 3))

    def fit(self, X, y=None):
        self.vectorizer.fit(X)
        return self

    def transform(self, X, y=None):
        tfidf_features = self.vectorizer.transform(X)
        word2vec_features = np.array([
            np.mean([self.word2vec_model.wv[word] for word in sentence.split() if word in
self.word2vec_model.wv]
                    or [np.zeros(self.word2vec_model.vector_size)], axis=0)
            for sentence in X
        ])
        combined_features = np.hstack((tfidf_features.toarray(), word2vec_features))
        return combined_features
# Initialize the custom transformer
combined_transformer = TfidWord2VecTransformer(word2vec_model)


# Create a pipeline with SMOTE and Logistic Regression
pipeline = ImbPipeline(steps=[
    ('combined_transformer', combined_transformer),
    ('smote', SMOTE(random_state=42)),
    ('classifier', LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial'))
])

# Prepare data for the pipeline
X_train = train_data['cleaned_sentence']
y_train = train_data['encoded_label']
X_test = test_data['cleaned_sentence']
y_test = test_data['encoded_label']

# Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Apply cross-validation to the entire training set using the best pipeline
cv_scores = cross_val_score(pipeline, X_train, y_train, cv=5, scoring='f1_macro')

# Print the cross-validation scores
print("Cross-Validation Scores:", cv_scores)
print("Mean CV F1 Score:", cv_scores.mean())

# Predict the labels on the test set
test_predictions = pipeline.predict(X_test)

# Print out the classification report for the test set
print("Test Set Classification Report:")
print(classification_report(y_test, test_predictions, target_names=label_encoder.classes_))
```

Approach 2: LSTM with Word2Vec Embeddings

```
import nltk
from nltk.corpus import stopwords
import spacy
import string
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Model
from keras.layers import Input, Embedding, LSTM, Dense, Dropout
from keras.utils import to_categorical
from gensim.models import Word2Vec
import numpy as np

# Download the stopwords from NLTK
nltk.download('stopwords')
nltk.download('punkt')

# Load the spaCy model for NER and POS tagging
nlp = spacy.load('en_core_web_sm')

# Upload datasets
from google.colab import files

# Prompt to upload the unzipped file
uploaded = files.upload()


# Load the training data
train_data = pd.read_csv('propaganda_train.tsv', sep='\t', header=None, names=['label',
'sentence'])

# Load the testing data
test_data = pd.read_csv('propaganda_val.tsv', sep='\t', header=None, names=['label', 'sentence'])

# Remove the first row of both training and testing data
train_data = train_data.drop(0).reset_index(drop=True)
test_data = test_data.drop(0).reset_index(drop=True)

# Display the first few rows of the training data
print("Training Data:")
print(train_data.head(10))

# Display the first few rows of the testing data
print("\nTesting Data:")
print(test_data.head(10))

# Encode labels as integers representing each of the nine classes
```

```
label_encoder = LabelEncoder()
train_data['encoded_label'] = label_encoder.fit_transform(train_data['label'])
test_data['encoded_label'] = label_encoder.transform(test_data['label'])

# Custom transformer for text preprocessing
class TextPreprocessor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))
        self.punctuation = set(string.punctuation)

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.apply(self._preprocess)

    def _preprocess(self, text):
        # Lowercasing
        text = text.lower()

        # Remove <BOS> and <EOS> tokens
        text = re.sub('<BOS>|<EOS>', '', text)

        # Apply spaCy pipeline
        doc = nlp(text)

        # Remove stopwords and punctuation, and retain only relevant words (nouns, verbs,
adjectives, etc.)
        words = [
            f"{token.text}_{token.pos_}" for token in doc
            if token.pos_ in {'NOUN', 'VERB', 'ADJ', 'ADV'}
            and token.text.lower() not in self.stop_words
            and token.text not in self.punctuation
        ]

        # Include named entities in the text
        entities = [f"{ent.text}_ENTITY" for ent in doc.ents]

        # Combine words and entities
        processed_text = words + entities

        # Return preprocessed text
        return ' '.join(processed_text)

# Initialize the text preprocessor
text_preprocessor = TextPreprocessor()


# Apply text preprocessing to the training and testing data
train_data['cleaned_sentence'] = text_preprocessor.transform(train_data['sentence'])
test_data['cleaned_sentence'] = text_preprocessor.transform(test_data['sentence'])

# Verify the new columns
print("\nTraining Data with Cleaned Sentences:")
print(train_data.head(10))
print(train_data.columns)

print("\nTesting Data with Cleaned Sentences:")
print(test_data.head(10))
```

```
print(test_data.columns)

# Tokenize the cleaned sentences
tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_data['cleaned_sentence'])
X_train_seq = tokenizer.texts_to_sequences(train_data['cleaned_sentence'])
X_test_seq = tokenizer.texts_to_sequences(test_data['cleaned_sentence'])

# Pad the sequences
max_sequence_length = max(len(seq) for seq in X_train_seq)
X_train_pad = pad_sequences(X_train_seq, maxlen=max_sequence_length)
X_test_pad = pad_sequences(X_test_seq, maxlen=max_sequence_length)

# Extract the vocabulary size
vocab_size = len(tokenizer.word_index) + 1

# Train a Word2Vec model on the cleaned sentences
sentences = [sentence.split() for sentence in train_data['cleaned_sentence']]
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Create an embedding matrix
embedding_dim = 100
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in tokenizer.word_index.items():
    if word in word2vec_model.wv:
        embedding_matrix[i] = word2vec_model.wv[word]

LSTM Model

# Input layer
word_input = Input(shape=(max_sequence_length,), dtype='int32')

# Embedding layer
word_embedding = Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           weights=[embedding_matrix],
                           input_length=max_sequence_length,
                           trainable=True)(word_input)

# LSTM layers
lstm_out = LSTM(128, return_sequences=True, dropout=0.2, recurrent_dropout=0.2)(word_embedding)
lstm_out = LSTM(64, return_sequences=False, dropout=0.2, recurrent_dropout=0.2)(lstm_out)

# Dense layer
dense_out = Dense(64, activation='relu')(lstm_out)
dense_out = Dropout(0.5)(dense_out)

# Output layer
output = Dense(len(label_encoder.classes_), activation='softmax')(dense_out)

# Compile the model
model = Model(inputs=word_input, outputs=output)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Convert labels to categorical
y_train_cat = to_categorical(train_data['encoded_label'])
y_test_cat = to_categorical(test_data['encoded_label'])

# Train the model
```

```
model.fit(X_train_pad, y_train_cat, epochs=10, batch_size=32, validation_split=0.2)

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test_pad, y_test_cat)
print(f"Test accuracy: {test_accuracy}")

# Predict the labels on the test set
y_pred = model.predict(X_test_pad)
y_pred_classes = np.argmax(y_pred, axis=1)

# Print out the classification report
print(classification_report(test_data['encoded_label'], y_pred_classes,
target_names=label_encoder.classes_))
```