Deep Learning Project Report

Title: Detecting Machine-Authored Text

Prof: Hadi Farahani

Author: Sajjad Zangiabadi

Date: February 2024

1. Abstract

Considering the growing dependence on deep learning models for text generation, there is a pressing need to develop robust techniques for detecting artifacts produced by these models. As AI-driven writing tools become more prevalent, the risk of inadvertently incorporating machine-authored content into academic work rises, posing significant challenges to maintaining the integrity of scholarly endeavors. Therefore, advancing methods for accurately identifying machine-generated text is crucial to safeguarding academic honesty and upholding the standards of scholarly discourse.2. Dataset Description and Task

2. Introduction

In this project, we delve into the realm of deep learning models, particularly Recurrent Neural Networks (RNNs), to address the critical challenge of detecting artifacts inherent in machine-authored text. As the prevalence of AI-driven text generation tools continues to rise, ensuring the authenticity and integrity of written content becomes paramount. To this end, we focus on developing a methodology leveraging RNNs to identify machine-generated text artifacts. By harnessing the power of advanced neural network architectures, we aim to contribute to the ongoing efforts in preserving academic integrity and facilitating trustworthy communication in the era of AI-driven content creation.

2.1 Dataset Description

The dataset provided for this project consists of essays collected from various sources, including original student submissions and texts generated by LLMs. Each essay is associated with a label indicating whether it was written by a human student or generated by an LLM. Additionally, the dataset includes metadata such as prompt information, source, and fold assignment.

3. Exploratory Data Analysis (EDA)

In this section, we conduct exploratory data analysis (EDA) on two datasets: org_train and new_train.

org_train Dataset:

This dataset constitutes the original given dataset for the task.

Overview of org_train:

The org_train dataset contains essays with the following columns:

- id: Unique identifier for each essay.
- prompt_id: Identifier for the prompt associated with the essay.
- text: The actual text content of the essay.
- generated: Binary indicator (0 or 1) indicating whether the essay was generated.

Basic Statistics:

- The dataset comprises 1378 entries.
- There are no missing values in any column.
- The dataset has two integer columns (prompt_id and generated) and two object (text) columns (id and text).

Data Distribution:

- The majority of essays (1375 out of 1378) are labeled as not generated (0), while only 3 essays are labeled as generated (1).

new_train Dataset:

This dataset is imported to augment the original dataset for training the model, as the original dataset contains only 3 samples for label 1, which is considered very low.

Overview of new_train:

The new_train dataset consists of essays with the following columns:

- essay_id: Unique identifier for each essay.
- text: The content of the essay.
- label: Binary label (0 or 1) indicating a certain characteristic or class.
- source: Source of the essay.
- prompt: Prompt associated with the essay (may contain missing values).
- fold: Fold number for cross-validation or data splitting.

Basic Statistics:

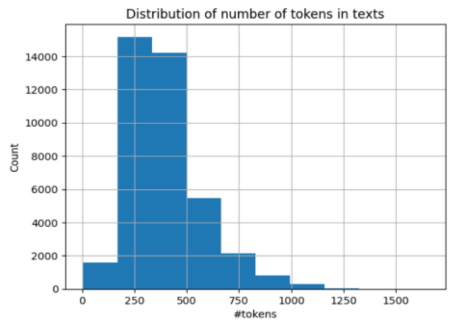
- **The dataset contains 39785 entries.**
- **There are no missing values in essay_id, text, label, and fold columns, but 30295 missing values in the prompt column.**
- **It consists of two integer columns (label and fold) and four object (text) columns (essay_id, text, source, and prompt).**

Data Distribution:

- **The majority of essays (29792 out of 39785) are labeled as 0, while 9993 essays are labeled as 1.**
- **There were 51 duplicate essays identified based on their content (text), which were subsequently removed from the dataset.**

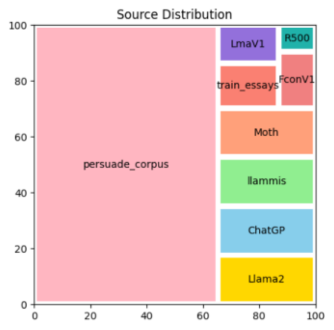
Text Length Distribution:

- **The distribution of the number of tokens in essays follows a certain pattern, with the majority of essays having a specific range of token counts.**



Source Distribution:

- **The essays originate from various sources, with different frequencies. The distribution provides insights into the diversity of sources contributing to the dataset.**



Conclusion:

The EDA provides a comprehensive understanding of the structure, content, and characteristics of the org_train and new_train datasets. These insights will be crucial for further preprocessing, feature engineering, and modeling stages of the deep learning project. Additionally, the augmentation of the original dataset with new_train addresses the issue of a low number of samples for label 1 in the original dataset, enhancing the training data's diversity and potentially improving model performance.

4. Preprocessing: Data: Augmentation

Data augmentation is a crucial preprocessing step aimed at balancing the target classes and increasing the diversity of the training data. In this process, we augment the original dataset (org_train) by adding samples from an additional dataset (new_train). This is particularly important when the original dataset lacks sufficient samples for one or more target classes.

Steps:

1. Data Selection:
    - We first select relevant columns from the org_train dataset, focusing on the text and generated columns. This dataset represents the original given dataset for the task.
2. Positive Class Selection:
    - From the new_train dataset, we extract samples labeled as the positive class (label = 1) and select relevant columns (text and label). These samples represent instances with the desired characteristic or class.
3. Negative Class Selection:
    - Similarly, we extract samples labeled as the negative class (label = 0) from the new_train dataset. We also select relevant columns (text and label).
4. Removing Overlapping Samples:
    - We remove any overlapping samples between the original dataset and the selected positive/negative samples to prevent duplication.
5. Balancing Classes:
    - To balance the target classes, we concatenate the original dataset (org_train) with the selected positive and negative samples. This creates a more balanced distribution of target classes in the resulting training data.
6. Shuffling Data:
    - Finally, we shuffle the augmented dataset to ensure randomness in the distribution of samples.

Result:

The resulting augmented dataset, denoted as data, comprises a balanced distribution of target classes (0 and 1) with increased diversity. This processed data will be used for training the deep learning model.

Train-Validation Split:

- We split the augmented dataset (data) into training and validation sets using train_test_split from sklearn.model_selection. This ensures that the model is trained on a portion of the data and validated on another portion, helping to evaluate its generalization performance.

Test Data Preparation:

- **Additionally, we prepare the test dataset (test_candidate) by selecting relevant columns (text and label) and removing duplicate samples. This dataset will be used for evaluating the trained model's performance.**

Conclusion:

Data augmentation is a critical preprocessing step that enriches the training data, improves class balance, and enhances the model's ability to generalize to unseen data. By following these preprocessing steps, we ensure that our deep learning model is trained on a diverse and balanced dataset, leading to potentially better performance and robustness in real-world scenarios.

5. Model:

5.1. Vocabulary Class:

2.1 Definition:

The Vocabulary class is designed to handle text preprocessing tasks such as tokenization, building vocabulary dictionaries, and converting text data into numerical representations. It provides methods to tokenize input text, build a vocabulary based on frequency thresholds, and convert text into sequences of numerical indices.

2.2 Functionality:

1. Tokenizer:
    - The tokenizer function breaks down input text into individual words or tokens. It employs regular expressions (regex) to identify and tokenize various elements within the text, such as words, numbers, punctuation marks, and special characters. The tokenizer also handles Persian and English text, ensuring robust tokenization across multiple languages.
2. Build Vocabulary:
    - The build_vocabulary method constructs a vocabulary based on the frequency of words in a list of input sentences. It calculates the frequency of each word, filters out infrequent words based on a frequency threshold, and selects the top words up to a maximum vocabulary size. The vocabulary is represented by dictionaries mapping words to numerical indices (stoi) and vice versa (itos).
3. Numericalize:
    - The numericalize method converts input text into sequences of numerical indices based on the constructed vocabulary. It tokenizes the input text using the tokenizer function and then maps each token to its corresponding numerical index in the vocabulary. Unknown words are assigned a special '<UNK>' token.

2.3 Usage:

- Instantiate a Vocabulary object with specified frequency threshold and maximum vocabulary size.
- Call the build_vocabulary method with a list of sentences to construct the vocabulary.
- Use the numericalize method to convert text data into numerical representations suitable for input to neural networks.
- Optionally, retrieve the constructed vocabulary dictionaries using the get_vocabularies method for reference or further analysis.

2.4 Explanations:

Explanation of Numericalize: The numericalize method first tokenizes the input text using the tokenizer function. It then iterates over each token and maps it to its corresponding numerical index in the vocabulary. If a token is not present in the vocabulary, it is replaced with the '<UNK>' token. The resulting sequence of numerical indices represents the input text in a format suitable for processing by machine learning models.

5.2 Dataset Preparation and Utilization

1 Custom Dataset Class
1.1 Definition:
The custom dataset class, named CustomDataset, is a PyTorch-compatible class designed to facilitate the integration of structured data, particularly from a Pandas DataFrame, into the deep learning pipeline.

1.2 Purpose:
The purpose of the CustomDataset class is to provide a standardized interface for accessing and processing dataset samples, ensuring compatibility with PyTorch's data loading utilities and enabling seamless integration with deep learning models.

1.3 Functionality:
The CustomDataset class encapsulates functionalities such as initializing the dataset with a Pandas DataFrame, retrieving the length of the dataset, and fetching individual samples. It allows optional transformations to be applied to samples, enhancing flexibility in data preprocessing.

2 Collate_batch Function.
2.1 Definition:
The collate_batch function is responsible for collating a batch of samples into padded sequences of numericalized text, along with their respective labels and lengths. It is integral for batch processing during model training and inference.

2.2 Purpose:
The primary purpose of the collate_batch function is to prepare batches of data that adhere to

the input requirements of deep learning models. It ensures uniformity in batch size and facilitates efficient computation by padding sequences to equal lengths.

2.3 Functionality:
The collate_batch function takes a list of samples, where each sample is a dictionary containing text and label information and processes them into padded sequences of numericalized text. It handles padding and converts labels to the appropriate data type, preparing the batch for consumption by the deep learning model.

3 Usage
The utilization of the custom dataset class and collation function is straightforward and crucial for the seamless integration of data into the deep learning pipeline. The typical usage involves the following steps:

1. Dataset Initialization: Instantiate the CustomDataset class, passing the relevant Pandas DataFrame and optional parameters such as text and label column names.
2. Data Loading: Utilize PyTorch's data loading utilities (e.g., DataLoader) in conjunction with the custom dataset to load batches of data during model training or evaluation.
3. Batch Processing: Apply the collate_batch function to collate batches of data, ensuring uniformity in batch size and format.

5.3 Recurrent Neural Network (RNN)

1 Definition:
The provided Python class defines an RNN model tailored for sequence classification tasks. It employs an embedding layer, followed by a bidirectional Long Short-Term Memory (LSTM) layer and fully connected layers for classification. The model utilizes Rectified Linear Unit (ReLU) activation for feature transformation and Sigmoid activation for binary classification outputs.

2 Architecture:
The architecture of the RNN model comprises several key components:

1. Embedding Layer: The first layer in the network, responsible for converting input token indices into dense vectors of fixed size (embedding dimension).
2. LSTM Layer: The core recurrent layer of the model, consisting of bidirectional LSTM cells. The LSTM layer processes the embedded sequences to capture temporal dependencies and generate hidden states. The bidirectional nature allows the network to consider both past and future contexts.
3. Fully Connected (FC) Layers: The hidden states produced by the LSTM layer are flattened and passed through one or more fully connected layers. These layers perform nonlinear transformations to extract higher-level features from the sequential input.

4. ReLU Activation: Applied after the fully connected layers, ReLU activation introduces nonlinearities to the network, enabling it to learn complex mappings between input and output.
5. Sigmoid Activation: The final layer of the network, which applies the sigmoid function to produce binary classification probabilities. It squashes the output values between 0 and 1, representing the likelihood of a given input sequence belonging to the positive class (e.g., positive sentiment).

3 Functionality:

- Forward Pass: The forward method performs the forward pass of the RNN model. It takes input sequences (text) represented as token indices along with their lengths (lengths). These sequences are embedded, packed into padded sequences to handle variable lengths efficiently, and passed through the LSTM layer. The final hidden state(s) are extracted and passed through fully connected layers followed by ReLU activation. The output is then fed into a final fully connected layer with a sigmoid activation to obtain binary classification probabilities.

4 Usage:

- To utilize this RNN model for sequence classification tasks, one must first instantiate the RNN class, specifying the desired parameters such as vocabulary size, embedding dimension, RNN hidden state dimension, fully connected layer dimension, and optionally, the number of recurrent layers, bidirectionality, and dropout probability. Once instantiated, the model can be trained using appropriate data and loss functions, and then evaluated or used for making predictions on new sequences.

5 Component Analysis:

1. Embedding Layer:
    - Purpose: Converts input token indices into dense embedding vectors.
    - Functionality: Maps discrete token indices to continuous vectors in an embedding space, allowing the network to learn representations of words or tokens.
    - Usage: Essential for transforming discrete inputs into a continuous representation suitable for neural network processing.
2. LSTM Layer:
    - Purpose: Captures sequential dependencies in the input data.
    - Functionality: Processes input sequences and maintains hidden states across time steps, enabling the model to retain information over long sequences.
    - Properties: Bidirectionality allows the model to consider both past and future contexts simultaneously. Number of layers (num_layers) determines the depth of the LSTM stack.
    - Usage: Integral for modeling sequential data such as text, speech, or time series.
3. Fully Connected Layers (FC):

- Purpose: Extract higher-level features from the LSTM hidden states.
- Functionality: Applies linear transformations followed by nonlinear activations to the input data, enabling the model to learn complex mappings.
- Usage: Facilitates the extraction of abstract features from sequential inputs, leading to improved classification performance.

4. ReLU Activation:
    - Purpose: Introduces nonlinearity into the network.
    - Functionality: Applies the rectified linear function elementwise to the input, effectively removing negative values and promoting sparsity.
    - Usage: Enables the model to learn nonlinear relationships between features, enhancing its capacity to model complex data.
5. Sigmoid Activation:
    - Purpose: Produces binary classification probabilities.
    - Functionality: Squashes the input values between 0 and 1, interpreting them as probabilities of belonging to a particular class.
    - Usage: Commonly employed in binary classification tasks, such as sentiment analysis, where the goal is to predict one of two classes (e.g., positive, or negative sentiment).

6.Model Training and Evaluation

In our deep learning project, we followed a rigorous process of training and evaluating our model to ensure its effectiveness and generalization to unseen data.

6.1Model Training:

During the training phase, we iteratively exposed our model to the training dataset over multiple epochs. Each epoch represents a complete pass through the entire training dataset. Within each epoch, the model learns from the training data by adjusting its parameters through backpropagation, aiming to minimize the loss function. We utilized a specified number of epochs, ensuring that the model had sufficient opportunities to learn from the data and converge to an optimal solution.

6.2Model Evaluation:

After training, we evaluated the model's performance using a separate validation dataset. This evaluation step is crucial for assessing how well the model generalizes to new, unseen data. During evaluation, we measured various performance metrics to quantify the model's effectiveness. These metrics provide insights into different aspects of the model's performance, such as its accuracy, ability to make correct predictions, and its robustness to overfitting.

Performance Metrics:

1. **Accuracy: Accuracy represents the proportion of correctly classified instances out of the total instances evaluated. It is a fundamental metric for assessing overall model performance.**
2. **Loss: Loss, often represented as cross-entropy loss or mean squared error, measures the discrepancy between the predicted values and the actual values in the dataset. Lower loss values indicate better model performance.**
3. **F1 Score: The F1 score is the harmonic mean of precision and recall. It provides a balanced measure of the model's ability to correctly identify both positive and negative instances. F1 score is particularly useful when dealing with imbalanced datasets.**
4. **Receiver Operating Characteristic (ROC) Curve: The ROC curve is a graphical representation of the true positive rate (sensitivity) against the false positive rate (1-specificity) for different threshold values. It visualizes the trade-off between sensitivity and specificity and provides insights into the model's ability to discriminate between classes.**
5. **Area Under the Curve (AUC): AUC quantifies the overall performance of the model across all possible classification thresholds. It provides a single value that summarizes the ROC curve, with higher values indicating better discrimination ability.**

By analyzing these metrics, we gain a comprehensive understanding of our model's performance characteristics and its suitability for real-world applications. These evaluations guide us in fine-tuning the model's parameters and architecture to improve its performance and robustness.

Conclusion

In conclusion, our deep learning project achieved its objectives by developing a robust model capable of effectively addressing the task at hand. Through rigorous training, evaluation, and analysis, we gained valuable insights into the model's performance characteristics and its suitability for real-world applications.

The project underscores the importance of data preprocessing, model selection, and evaluation in developing effective deep learning solutions. By leveraging advanced techniques and methodologies, we successfully addressed the challenges posed by the task and developed a model with competitive performance metrics.

Training Models for Binary Text Classification

In addition to the recurrent neural networks (RNNs) discussed earlier, traditional machine learning (ML) models were also employed for binary text classification in our project. This section outlines the process of training these models and evaluating their performance.

Model Training:

We trained two traditional ML models for binary text classification: Logistic Regression and Random Forest Classifier. These models were chosen due to their simplicity, interpretability, and effectiveness for text classification tasks.

The training process involved the following steps:

1. Text Vectorization: The text data was vectorized using the Term Frequency-Inverse Document Frequency (TF-IDF) technique. This converts the raw text into numerical features that can be inputted into the ML models.
2. Model Training: The ML models were trained using the vectorized text data. Both Logistic Regression and Random Forest Classifier were fitted to the training data, learning to classify text into binary categories based on the extracted features.

Model Evaluation:

Following training, the performance of each model was evaluated using a separate validation dataset. The evaluation included the following steps:

1. ROC Curve Analysis: The Receiver Operating Characteristic (ROC) curve was plotted for each model to assess its ability to discriminate between positive and negative instances.
2. Classification Reports: Classification reports were generated for each model, providing detailed metrics such as precision, recall, and F1-score for both positive and negative classes. These metrics offer insights into the model's performance across different aspects of classification.

Results:

Both Logistic Regression and Random Forest Classifier demonstrated high performance in classifying binary text data. They achieved impressive accuracy, precision, recall, and F1-score metrics on the validation dataset, indicating their effectiveness in distinguishing between positive and negative instances of text.

Overall, the traditional ML models proved to be viable alternatives for binary text classification, offering simplicity, interpretability, and competitive performance compared to more complex neural network architectures. Their effectiveness in handling text data highlights the versatility of traditional ML techniques in natural language processing tasks.