Report: Hate Speech Classifying by Using LSTM

Abstract:

- The focus of this project is to develop a machine learning model to detect hate speech in social media texts. The task is to classify text data into three categories: 0 (not hated), 1 (racism) and 2 (xeno). To achieve this, we leveraged an LSTM-based neural network, which is well suited to processing sequential data such as text. The data comes from various Kaggle datasets, including the Hate Speech Detection Curated Dataset, Racism Data and Xeno Data. These datasets were pre-processed, tokenized, and used to build a comprehensive vocabulary. Our model architecture included an embedding layer, a bidirectional LSTM layer, and fully connected layers with ReLU activation. The model was trained using the Adam optimizer and evaluated for accuracy, precision, recall and F1 score.

Introduction:

- Hate speech on social media platforms has become a growing problem with serious consequences for individuals and communities. It can incite violence and spread discrimination, creating a toxic online environment. Effective detection and mitigation of hate speech is critical to maintaining safe and inclusive digital spaces. Social media companies and policymakers are increasingly relying on automated systems to detect and flag hate speech, but the complexities and nuances of human language present significant challenges.

Data Sources:

- The data for this project was sourced from three different datasets available on Kaggle:
    - **Hate Speech Detection Curated Dataset**:
        - This dataset contains labeled instances of hate speech and non-hate speech. For this project, we only considered entries labeled as `0` (not hated) and ignored entries labeled as `1` (hate speech).
    - **Racism Data**:
        - This dataset specifically focuses on instances of racism in social media texts. Entries in this dataset are labeled as `1` (racism).
    - **Xeno Data**:
        - This dataset contains instances of xenophobia in social media texts, with entries labeled as `2` (xeno).

Vocabulary Building

- The process of building a vocabulary is essential for text-based machine learning tasks, including hate speech classification. A well-constructed vocabulary ensures that the model can effectively process and understand the textual data it encounters. The following steps outline how the vocabulary is built using the provided code:

  1. **Initialization**:
     - An instance of the `Vocabulary` class is created with two parameters: `freq_threshold` and `max_size`. These parameters control the vocabulary's size and the minimum frequency required for a word to be included in the vocabulary.
  2. **Tokenization**:
     - The `TweetTokenizer` from the NLTK library is utilized to tokenize the text data. Tokenization involves splitting each text entry into individual tokens or words.
  3. **Building Vocabulary**:
     - The `build_vocabulary` method is called to construct the vocabulary based on the tokenized text data (`X_train`). This method calculates the frequency of each token in the dataset and retains only those tokens that occur more frequently than the specified `freq_threshold`.
     - After filtering, the vocabulary is capped at a maximum size (`max_size`) to prevent it from becoming too large and unwieldy.
     - The most frequent tokens are selected to be included in the vocabulary, and each token is assigned a unique index for numerical representation.
     - Special tokens such as `<PAD>`, `<SOS>`, `<EOS>`, and `<UNK>` (unknown) are also added to handle padding, start-of-sequence, end-of-sequence, and out-of-vocabulary words, respectively.
  4. **Numericalization**:
     - The `numericalize` method is provided to convert textual data into numerical form using the built vocabulary. Each token in a text entry is replaced with its corresponding index in the vocabulary. Unknown tokens are represented by the index of `<UNK>`.
  5. **Finalization**:
     - Once the vocabulary is constructed and the textual data is numericalized, the vocabulary instance (`vocab`) is ready for use in model training and evaluation.

Custom Dataset Description

- The `TextDataset` class represents a custom dataset designed specifically for text classification tasks, such as hate speech classification. This dataset encapsulates textual data along with their corresponding labels, allowing for seamless integration with PyTorch's data loading utilities. Here's a comprehensive description of the `TextDataset` class and its functionalities:

- Dataset Initialization:

  - The `TextDataset` class is initialized with the following parameters:
    - `texts`: A pandas Series containing textual data (e.g., sentences, phrases).
    - `labels`: A pandas Series containing corresponding labels for the textual data (e.g., class labels indicating hate speech or non-hate speech).
    - `vocab`: An instance of the `Vocabulary` class, which provides methods for tokenizing and numericalizing text data.

        - `max_length`: An optional parameter specifying the maximum length of input sequences. Text sequences longer than this length will be truncated, while shorter sequences will be padded with zeros.
- Dataset Length:
    - The `__len__` method returns the total number of samples in the dataset, which corresponds to the length of the `texts` Series.

- Dataset Indexing:

    - The `__getitem__` method enables indexing into the dataset to retrieve individual samples. Given an index `idx`, it retrieves the text and label at that index.
    - The text data is tokenized and numericalized using the vocabulary (`vocab`). Tokenization involves splitting the text into individual tokens (words), while numericalization replaces each token with its corresponding index in the vocabulary.
    - If the length of the numericalized text sequence is less than the specified `max_length`, it is padded with zeros to ensure uniform input size. If the sequence exceeds `max_length`, it is truncated to the specified length.

Model Development

- The model development phase involves designing and implementing a neural network architecture capable of effectively classifying hate speech in textual data. The provided code defines a text classification model using a Long Short-Term Memory (LSTM) network. Here's a detailed explanation of the model's components and their functionalities:

- Model Architecture:

1. **Embedding Layer**:
    - The `Embedding` layer is the first component of the model, responsible for converting input tokens into dense vector representations called embeddings. It learns to represent words in a continuous vector space where similar words have similar embeddings.
    - The `vocab_size` parameter determines the size of the vocabulary, while the `embedding_dim` parameter specifies the dimensionality of the embedding vectors.
    - The `padding_idx` parameter ensures that the padding token `<PAD>` is ignored during training.
2. **LSTM Layer**:
    - The `LSTM` layer is a type of recurrent neural network (RNN) architecture that is well-suited for sequence modeling tasks like text classification. It processes sequences of input embeddings and captures dependencies over time.
    - The `embedding_dim` parameter specifies the size of the input embeddings, while the `hidden_dim` parameter determines the dimensionality of the hidden state vectors.
    - The `num_layers` parameter controls the depth of the LSTM network, allowing for multiple layers of stacked LSTMs.
    - The `bidirectional=True` parameter indicates that the LSTM is bidirectional, meaning it processes input sequences in both forward and backward directions.
3. **Fully Connected Layers**:
    - Following the LSTM layer, two fully connected (linear) layers (`fc1` and `fc2`) are employed to perform classification based on the LSTM output.
    - The first fully connected layer (`fc1`) reduces the dimensionality of the LSTM output vectors (`hidden_dim * 2`) to improve computational efficiency.

- The `ReLU` activation function is applied after the first fully connected layer to introduce non-linearity and enable the model to learn complex patterns in the data.
- The second fully connected layer (`fc2`) produces the final output logits, which are used to predict the probability distribution over the output classes.

Result: (totally unseen data)

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1000
1	0.98	0.94	0.96	573
2	0.98	0.99	0.98	336
accuracy			0.98	1909
macro avg	0.98	0.97	0.98	1909
weighted avg	0.98	0.98	0.98	1909