- ***ARASH KHAJOOEI***
- ***[ARASH.KHAJOOEI@GMAIL.COM]***

SMS Spam Collection Dataset Analysis

The SMS Spam Collection Dataset is a collection of SMS messages labeled as either spam or ham (non-spam). The objective of this analysis is to perform a thorough exploration and understanding of the dataset by employing various text analysis techniques. The analysis includes preprocessing the text data, performing basic and advanced analyses, and visualizing the results.

Step 1: Load and Explore the Dataset

The first step involves loading the dataset and gaining an initial understanding of its structure. The dataset is loaded using the Pandas library, and the distribution of labels (spam vs. ham) is examined to identify class imbalances.

Step 2: Data Preprocessing

In this step, the raw text data is preprocessed to make it suitable for analysis. The preprocessing steps include:

1. Removing punctuation marks: Punctuation is removed from the text using string manipulation.
2. Converting to lowercase: All text is converted to lowercase to ensure uniformity.
3. Tokenization: The text is split into individual words or tokens using the NLTK library's tokenization function.
4. Stopword Removal: Common stopwords (e.g., "and", "the", "is") are removed from the text using NLTK's stopwords list.
5. Stemming and Lemmatization: Words are reduced to their base form using stemming and lemmatization to handle variations of words.

Step 3: Basic and Advanced Analysis

This step involves both basic and advanced analyses to extract meaningful insights from the preprocessed data.

- ***Basic Analysis:***
    1. The number of words in each message is calculated and stored.
    2. The number of unique words and repeated words in each message is determined.
    3. Basic statistics, such as mean, median, and quartiles, are computed for word counts, unique words, and repeated words.
- ***Advanced Analysis:***
    1. **Sentiment Analysis:** Sentiment intensity scores are computed for each message using the Sentiment Intensity Analyzer from the NLTK library. The distribution of sentiment scores is visualized through a histogram.

2. **Word Cloud Visualization:** A word cloud is generated using the most frequent words in the dataset. The word cloud provides a visual representation of word frequency, with more frequent words displayed larger.
3. **Named Entity Recognition (NER):** Named entities are identified in a sample message using POS tagging and the ne_chunk function from NLTK. NER helps in identifying entities such as names, locations, and organizations.
4. **Visualizing Named Entities:** Named entities are visualized in a sample message using the spaCy library's displacy module. This provides a visual representation of the recognized named entities.

Step 4: Conclusion and Insights

Through this comprehensive analysis, several insights can be gained:

- The dataset's class distribution shows the balance between spam and ham messages.
- Sentiment analysis reveals the overall sentiment of the messages, whether they are positive, negative, or neutral.
- The word cloud provides a visual representation of frequently occurring words, highlighting common themes.
- Named entities and their types can be identified using NER, and this information can be useful for understanding message content and context.

Step 5: Future Considerations

This analysis can be extended and customized based on specific objectives:

- Advanced sentiment analysis techniques could be applied, considering aspects like sarcasm detection.
- Topic modeling techniques could be employed to identify distinct topics within the messages.
- Machine learning models could be trained for spam detection using text classification algorithms.

Custome database to train Naïve bayes

NLTK Version:

1. Import Required Libraries

- Here, you're importing the necessary libraries for data processing and machine learning. pandas is used for data handling, CountVectorizer for converting text data to numerical features, MultinomialNB for the Naive Bayes classifier, train_test_split for splitting data, accuracy_score for evaluating accuracy, and nltk for natural language processing tasks.

2. Define Your Dataset

- In this step, you need to define your dataset. The dataset consists of text sentences and corresponding emotion labels. You can add your own data in the form of (sentence, emotion) pairs.

3. Convert Data to DataFrame

- The dataset is converted into a pandas DataFrame with two columns: "sentence" and "emotion".

4. Split Data into Training and Testing Sets

- The dataset is split into training and testing sets using the train_test_split function from sklearn.model_selection. This is essential to assess the model's performance on unseen data.

5. Preprocess Text Data

- This step involves preprocessing the text data to remove stopwords and perform tokenization. NLTK's stopwords and word_tokenize functions are used to remove common words that don't contribute much to the meaning of the text.

6. Convert Text Data to Numerical Features

- Using CountVectorizer, the text data is converted into numerical features that machine learning algorithms can work with. This involves creating a matrix where each row corresponds to a sentence, and each column corresponds to a unique word in the entire dataset.

7. Train Naive Bayes Classifier

- A Naive Bayes classifier is chosen for its simplicity and effectiveness in text classification tasks. The classifier is trained on the training data, with the text features (X_train) and corresponding emotions (train_data["emotion"]).

8. Predict Emotions on Test Data

- The trained classifier is used to predict emotions for the test data (X_test). The predictions are stored in the predictions variable.

9. Evaluate Accuracy

- Using the ground truth emotion labels from the test data, the accuracy of the model's predictions is evaluated using the accuracy_score function. This gives you an indication of how well the model performs on the test data.

scikit-learn Version:

- The scikit-learn version follows a similar structure to the NLTK version, but it uses scikit-learn's built-in CountVectorizer and stopwords for preprocessing.

- The main difference is in how stopwords are handled. In the NLTK version, NLTK's stopwords are downloaded and used, while in the scikit-learn version, stopwords are handled directly by CountVectorizer using the stop_words parameter.

Overall, both versions aim to preprocess text data, convert it into numerical features, train a Naive Bayes classifier, predict emotions, and evaluate the model's accuracy on test data. I achieved 75% accuracy using both models as it can be seen in the code.

LSTM for sentiment analysis using IMDB comments dataset

Here's a comprehensive step-by-step explanation of the implementation of an LSTM for sentiment analysis using IMDB comments:

Step 1: Loading and Preprocessing the Dataset

- The IMDB comments dataset is loaded from a CSV file. This dataset contains movie reviews along with their corresponding sentiment labels (positive or negative).
- The dataset is read row by row, and each review is transformed to lowercase and split into individual words. The sentiment label is also extracted.

Step 2: Building the Vocabulary

- A vocabulary is built from the words in the reviews. Each word is counted to determine its frequency in the dataset.
- The vocabulary is sorted based on word frequency, and a specified number of the most common words are selected to form the vocabulary.
- Each word in the vocabulary is assigned a unique index for numerical representation.

Step 3: Converting Text to Numerical Data

- The dataset is processed again, and each review's words are replaced by their corresponding vocabulary indices. This converts the reviews to numerical sequences that can be fed into the LSTM.
- The dataset now consists of pairs: a numerical sequence representing a review and its corresponding sentiment label.

Step 4: Loading Pretrained GloVe Embeddings

- Pretrained word embeddings (e.g., GloVe embeddings) are loaded from a file. These embeddings are trained on a large corpus and capture semantic relationships between words.

- Each embedding is associated with a specific word. The embeddings are loaded into a dictionary where words are keys, and their embeddings are values.

Step 5: Creating an Embedding Matrix

- An embedding matrix is created to represent the limited vocabulary used in this analysis. The matrix has dimensions: vocabulary size × embedding dimension.
- For each word in the limited vocabulary, if the word has a corresponding embedding in the pretrained embeddings, its embedding values are copied into the embedding matrix at the index corresponding to that word.
- This matrix now contains pretrained word embeddings for words in the limited vocabulary.

Step 6: Preparing Training and Testing Data

- The numerical data is split into training and testing sets using a specified split ratio (e.g., 80% for training and 20% for testing).
- The training and testing data are now ready for use in training and evaluating the LSTM model.

Step 7: Implementing the LSTM Model

- The LSTM model is implemented. In the provided code, the implementation focuses on creating an LSTM cell, forward pass, and basic training loop.
- The LSTM cell consists of gates (forget, input, output) and a memory cell that updates at each time step to capture information from previous steps.
- The forward pass processes the input sequence (numerical review) through the LSTM cells and outputs a prediction.
- The basic training loop iterates through the training data in minibatches. For each batch, forward and backward passes are performed, and model parameters are updated using gradient descent.

Step 8: Evaluating and Testing the Model

- After training the LSTM, the model's performance is evaluated using the testing data.
- The LSTM predicts sentiment labels for the testing data and compares them to the ground truth labels.
- The accuracy or other suitable metrics are calculated to assess how well the model performs on the testing data.

Challenges with Testing:

- **Resource Requirements:** Training deep learning models like LSTMs requires significant computational resources, including memory (RAM) and processing power (CPU/GPU).
- **Code Complexity:** While the provided code implements a basic LSTM, there are many details and optimizations required for a functional model. These details can be challenging to get right without using established libraries like TensorFlow or PyTorch.

Memory (RAM) Limitations:

- **Embeddings:** Loading and managing pretrained embeddings can be memory-intensive. These embeddings are typically large matrices, and storing them can consume a significant amount of memory.
- **Vocabulary Size:** When working with a large vocabulary, the memory needed to store embeddings and numerical representations of words can quickly add up.
- **Minibatch Training:** The code tries to implement minibatch training to reduce memory consumption. However, managing minibatches while training an LSTM can still demand substantial memory.
- **Backpropagation:** Training an LSTM involves storing intermediate values during the backward pass for gradient computation. This process can require additional memory.
- **Training Data:** Large training datasets, even if limited, can consume memory when loaded into memory for training.
- **Model Parameters:** Storing the weights and biases of the model (LSTM cell) adds to memory usage.

Why RAM is Crucial:

- **Data Movement:** Deep learning involves lots of data movement, especially during matrix multiplications and gradient computations. RAM facilitates quick access to these data, improving training speed.
- **Storage of Parameters:** Model parameters, intermediate computations, and data used in backpropagation all need to be stored in RAM during training.
- **Minibatch Processing:** Loading minibatches requires memory to hold the current batch of data.
- **Parallelism:** Some operations (e.g., matrix multiplications) can be parallelized, and sufficient RAM enables efficient parallel processing.

Consequences of Insufficient RAM:

- **Crashes:** Insufficient RAM can lead to program crashes or out-of-memory errors.
- **Slow Training:** Smaller RAM might result in slower training due to frequent disk swaps or limited parallelism.
- **Unreliable Results:** If the model can't fit into memory, it might not train or generalize well due to truncation or lack of data.

The Long Short-Term Memory (LSTM) architecture implemented in the provided code is a fundamental component of a recurrent neural network (RNN). LSTMs are designed to capture and process sequential data by maintaining a memory of previous time steps, making them particularly useful for tasks like sequence prediction and sentiment analysis.

LSTM Architecture Overview:

- **Input Sequence:** The input to the LSTM is a sequence of word indices, representing a review. Each word index corresponds to a word in the vocabulary and is embedded into a vector using pretrained word embeddings.
- **LSTM Cell:** The core component of the LSTM architecture is the LSTM cell. Each LSTM cell contains several key components that allow it to capture and process sequential information:
    1. **Input Gate (i_t):** Determines how much of the current input should be added to the cell's memory.
    2. **Forget Gate (f_t):** Decides what information from the previous cell state should be forgotten or discarded.
    3. **Output Gate (o_t):** Controls how much of the cell's memory should be exposed as the output.
    4. **Cell State (c_t):** Represents the memory of the cell at a given time step. It can store both short-term and long-term information.
- **Cell State Update:** The cell state is updated based on the input gate, forget gate, and new candidate values. The forget gate decides what information should be removed from the cell state, while the input gate and candidate values control what new information should be added.
- **Hidden State Update:** The hidden state (or output) of the LSTM is determined by the output gate and the updated cell state. It represents the relevant information that the LSTM has learned from the input sequence.
- **Time Steps:** The LSTM cell is iteratively applied to each time step of the input sequence. The hidden state and cell state are updated at each time step based on the input and previous states.
- **Final Prediction:** After processing the entire sequence, the final hidden state is used to make predictions about the sentiment of the input review. The output can be interpreted as the probability of the sentiment being positive or negative.

Advantages of LSTM Architecture:

- **Long-Term Dependencies:** LSTMs are designed to capture long-term dependencies in sequential data, making them suitable for tasks where context over multiple time steps is crucial.
- **Memory Management:** The architecture of LSTM cells allows them to retain and update memory over time, making them well-suited for tasks involving memory retention and processing.
- **Vanishing Gradient Problem:** Unlike traditional RNNs, LSTMs address the vanishing gradient problem by maintaining a separate memory cell and using gates to control the flow of information.

Challenges and Considerations:

- **Model Complexity:** While LSTMs are powerful, they can be complex to implement and tune due to the multiple gates and interactions within the cell.
- **Resource Demands:** LSTMs can be memory-intensive, and training them on large datasets may require significant computational resources.
- **Overfitting:** As with any neural network, LSTMs can overfit if not properly regularized or if the dataset is too small.

To access my code, you can visit the NLP repository which is available in my GitHub profile: https://github.com/arashkhgit/NLP_task/tree/main

ARASH KHAJOOEI

29 AUGUST 2023