Sentiment Analysis on imdb dataset using CNN, RNN and Transformer

Krishnasai Paleti, Raghavendra Jagirdar and Mega Sri Shyam

kxp5619, rvj5301, mpb6512

CMPSC 448 Final Project Report

Github Repository: Codebase

December 3, 2023

Abstract

This article delves into the development of sentiment analysis models using the IMDB dataset. The models aim to classify movie reviews into positive or negative sentiments. This work is significant in the field of natural language processing (NLP), providing insights into consumer opinions and aiding in automated review analysis.

1 Introduction and Background

1.1 Sentiment Analysis

Sentiment analysis, often known as opinion mining, is a natural language processing (NLP) method for identifying the positivity and negativity of data. Businesses frequently do sentiment analysis on text information to track the perception of their brands and products in customer reviews and to better understand their target market. Sentiment analysis is primarily concerned with the polarity of a text, but it also extends beyond polarity to identify particular sentiments, such as urgency and intent. categories can be defined and modified to match sentiment analysis needs based on how one wishes to interpret consumer input and questions.

The IMDB dataset, a large collection of movie reviews, is widely used for benchmarking sentiment analysis models. It offers a balanced set of 50,000 reviews, split equally into training and test sets.

2 Pre-Processing procedure

Different preprocessing procedures are used for each model; CNN, RNN, and Transformer.

2.1 CNN Pre-Processing

The IMDB dataset is loaded from Keras. This dataset contains 50,000 movie reviews from IMDB, split into 25,000 reviews for training and 25,000 for testing. Each review is already pre-processed and encoded as a sequence of word indexes, corresponding to the words' frequency ranking in the dataset.

Keras provides a default word index for the IMDB dataset, which maps words to their corresponding indices. However, in the preprocessing step, this index is adjusted:

The index is set to 3: This adjustment is made to make room for three special tokens: PAD, START, and UNK. Adding Special Tokens:

- PAD (Padding Token): Assigned an index of 0. Used to fill in sequences to make them all the same length.
- START (Start Token): Assigned an index of 1. Indicates the beginning of a review.
- UNK (Unknown Token): Assigned an index of 2. Represents words that are not in the top 80,000 words of the vocabulary.

The vocab size is restricted to 80,000. This parameter is fixed and does not change when the model is trained and tuned.

The reviews in the IMDB dataset vary in length. For a neural network to process these reviews, they need to be of uniform length. This is achieved through sequence padding. The Maximum sequence length is set to 500. Reviews longer than 500 words are truncated, and shorter ones are padded.

A function "decode_review" is defined to transform these indexed sequences back into words. This is useful for human readability and verification of the preprocessing steps. It converts the sequence of indices back into words using the adjusted word index.

The preprocessing steps convert textual movie reviews into numerical sequences suitable for input into a neural network. These steps include tokenization, adjusting the word index to include special tokens, restricting the vocabulary size to the top frequent words, and padding the sequences to a fixed length. These preprocessing steps are critical for the consistent and effective training of the model on the IMDB dataset.

2.2 RNN Pre-Processing

The IMDB dataset is loaded using Keras's.load-data() function. This dataset consists of 50,000 movie reviews (split equally into training and test sets) and is pre-encoded as sequences of word indices. The parameters used for loading the dataset are:

- num_words: This parameter is not set in the initial loading, implying all words are considered. However, it's later used in the Optuna hyperparameter optimization to define the size of the vocabulary.
- skip-top: Set to 0, which means no top frequent words are skipped.
- maxlen: Not set initially, indicating that no maximum review length is enforced at this stage. It's later defined in the hyperparameter tuning.
- start-char: Set to 1, indicating that the start of each sequence will be marked with this character.
- seed: The random seed set to 13 for reproducibility.
- oov_char: Set to 2, representing words that were cut off by the num_words limit or not in the dataset.
- index-from: Set to 3, meaning the actual word indices start from this number, reserving indices for special tokens.

EDA is performed to gain insights into the dataset's characteristics:

Review Length Analysis: The lengths of reviews in both training and test sets are analyzed. Distribution plots are created using Seaborn to visualize the spread and density of review lengths. Additionally, statistical measures like mean, median, and mode are calculated to understand the central tendency and dispersion of review lengths.

Label Distribution: The balance of positive and negative reviews in the test set is examined by counting the occurrences of each label.

Sequence Padding

This step is crucial since it ensures that all sequences (reviews) are of the same length, a requirement for training neural networks:

- pad_sequences: This function is used to standardize the lengths of the reviews. Sequences longer than the specified 'maxlen' are truncated, and shorter ones are padded with zeros.
- The 'maxlen' parameter, which determines the length of each review, is optimized during the hyperparameter tuning process with Optuna.

A function 'decode' is defined to convert review indices back to text. This is more for verification and understanding of the data rather than a direct part of the preprocessing pipeline for model training.

2.3 Transformer Pre-Processing

The dataset is loaded from HuggingFace library. The Transformer model we use in the article is 'distilbert-base-uncased' from the HuggingFace library. Unlike CNN and RNN models, HuggingFace provides an inbuilt method to use DistilBERT tokenizer (AutoTokenizer class). AutoTokenizer is a class that automatically retrieves the appropriate tokenizer for a given model architecture from the Hugging Face model hub.

The distilbert-base-uncased tokenizer is specifically designed for the DistilBERT model and is trained to handle lowercase text (as indicated by 'uncased'). We then create a a preprocessing function to tokenize text and truncate sequences to be no longer than DistilBERT's maximum input length.

To apply the preprocessing function over the entire dataset, use HuggingFace Datasets 'map' function. We can speed up 'map' by setting batched=True to process multiple elements of the dataset at once.

We then create a batch of examples using 'DataCollatorWithPadding'. It's more efficient to dynamically pad the sentences to the longest length in a batch during collation, instead of padding the whole dataset to the maximum length.

3 Model Architecture

3.1 CNN Architecture and Implementation

The architecture comprises several layers, each designed to process and extract meaningful patterns from textual data.

Embedding Layer:
The first layer is an Embedding layer, which transforms the input sequence of word indices into dense vectors of fixed size (here, 16 dimensions). The vocabulary size is 80,000, and the input length for each review is set to a maximum of 500 words.

Convolutional and Pooling Layers
After the embedding layer, the model includes:

- Dropout Layer (Regularization): Introduced to reduce overfitting by randomly setting a fraction (rate) of the input units to 0 at each update during training time.
- Conv1D Layer: A 1D convolutional layer that convolves the embedded word

vectors using filters (default 64) of a specified kernel size (default 3) and strides (default 1). This layer uses 'relu' activation function.

- GlobalMaxPooling1D: This layer downsamples the input representation by taking the maximum value over the time dimension, reducing its dimensionality, and allowing for assumptions about feature independence.

Dense Layers:
Following the convolutional and pooling layers, the network includes a fully connected layer with relu activation function. The number of units (default 256) and kernel initializer (default 'glorot-uniform') can be adjusted. Another Dropout Layer follows the dense layer with the same rate as before. The final layer is a Dense layer with a single unit and a 'sigmoid' activation function, used for binary classification tasks.

3.2 RNN Architecture and Implementation

The model for sentiment analysis on the IMDB dataset is a simple recurrent neural network (SimpleRNN) developed using Keras.

Embedding Layer The first layer in the model is an Embedding layer, which serves to convert word indices into dense vectors of fixed size.

SimpleRNN layer The SimpleRNN layer is a type of recurrent neural network that can process sequences of data. It's particularly suited for tasks like sentiment analysis where the sequence of words and their context are important.

The layer will return only the last output in the output sequence, reducing the dimensionality of the output and making it suitable for passing to a dense layer. The activation function used is "relu" (Rectified Linear Unit).

Dense Layer A Dense layer is a fully connected layer that comes after the RNN layer. Its role is to interpret the features learned by the RNN. This layer contains a single neuron as it's used for binary classification (positive or negative sentiment).

An activation layer is added to apply the sigmoid activation function to the output of the Dense layer. The sigmoid function outputs a value between 0 and 1, representing the probability of a particular class (in this case, the likelihood of a review being positive).

3.3 Transformer Architecture and Implementation [2]

DistilBERT, short for Distilled-BERT, is a smaller, faster, and lighter version of the original BERT (Bidirectional Encoder Representations from Transformers) model, which is a state-of-the-art machine learning model for natural language processing (NLP) tasks. DistilBERT is designed to provide a more efficient alternative to BERT while retaining most of its performance.

DistilBERT is created through a process called knowledge distillation. This process involves training a smaller model (student) to replicate the behavior of a larger, pre-trained model (teacher). In the case of DistilBERT, the teacher is the BERT model. Although smaller in size, DistilBERT retains about 97% of BERT's performance on various benchmarks while being 40% smaller and 60% faster.

DistilBERT's architecture is similar to BERT but with fewer layers. Key components include:

- Transformer Blocks: The core of DistilBERT, like BERT, is based on the Transformer architecture. However, DistilBERT uses fewer transformer blocks (layers) than BERT. For instance, while BERT-base has 12 layers, DistilBERT has 6.
- Self-Attention Mechanism: Each transformer block in DistilBERT uses a self-attention mechanism, allowing the model to weigh the importance of different words in a sentence contextually.
- Feed-Forward Neural Network: Each transformer block also contains a feed-forward neural network for processing the output from the self-attention layers.
- Reduced Layers: The reduction in the number of layers significantly decreases the model size and computational requirements, making DistilBERT more efficient for both training and inference.
- Parameter Reduction: DistilBERT has about 40% fewer parameters than BERT, leading to less memory consumption and improved efficiency.
- Task-Agnostic: Like BERT, DistilBERT is pre-trained on a large corpus in a task-agnostic way, using tasks like masked language modeling (predicting missing words in a sentence).
- Fine-Tuning: For specific tasks, DistilBERT can be fine-tuned in a similar way to BERT, adjusting its weights slightly based on the task-specific data.

Before we start training our model, create a map of the expected ids to their labels with id2label and label2id:

Load DistilBERT with AutoModelForSequenceClassification along with the number of expected labels, and the label mappings

4 Training Details and Hyperparameter Tuning

The initial approach for hyperparameter tuning was to perform exhaustive grid search on the respective hyperparameters of CNN and RNN models. But while running the code on Google colab, the RAM maxes out causing the runtime to disconnect. To circumvent this bottleneck challenge, we have used 'Optuna', a framework to tune the hyperparameters.

4.1 Optuna [3]

Optuna is an open-source hyperparameter optimization framework, designed to automate the process of finding the best hyperparameters for machine learning models. It's particularly well-suited for deep learning models where the selection of optimal hyperparameters can significantly impact model performance.

How Optuna works:

Define the Optimization Objective: The user defines an objective function that takes a set of hyperparameters and returns a numerical score indicating the performance of a model trained with those hyperparameters. This score is usually a validation metric like accuracy, F1-score, etc.

Suggest Hyperparameters: Within the objective function, Optuna's API is used to suggest values for hyperparameters. Optuna supports various types of hyperparameters such as categorical, numerical (both discrete and continuous), and even conditional hyperparameters.

Optimization Algorithm: Optuna uses a Bayesian optimization approach, specifically the Tree-structured Parzen Estimator (TPE) algorithm, to decide which hyperparameters to evaluate next. TPE models the probability of a hyperparameter set given the observed performances and selects new hyperparameters to minimize (or maximize) the objective function.

Trial: Each iteration where the model is trained and evaluated with a particular set of hyperparameters is called a trial. Study: A collection of trials is referred to as a study. Optuna supports both single-objective and multi-objective optimization studies.

Pruning Mechanism: Optuna provides a feature called pruning, which stops a trial if it's unlikely to lead to a better result than the best trial so far. This saves computation time and resources.

4.2 CNN Training Details [1]

The model employs Optuna, a hyperparameter optimization framework, to find the best combination of parameters for the neural network. The objective function defines the hyperparameters to be tuned:

- **Filters**: The number of filters in the Conv1D layer, with options 64, 128, or 256.
- **Kernel Size**: The size of the kernel in the Conv1D layer, with choices 3, 5, or 7.
- **Strides**: The stride length of the convolution operation, with options 1, 2, or 5.

- **Units**: The number of neurons in the Dense layer, either 128 or 512.
- **Kernel Initializer**: The initializer for the kernel weights matrix, chosen from 'zero', 'glorot_uniform', 'glorot_normal', or 'TruncatedNormal'.
- **Rate**: The dropout rate, a float between 0.1 and 0.5.
- **Optimizer**: The optimization algorithm, either 'adam' or 'rmsprop'.
- **Batch Size**: The size of the batch, with options 32, 64, or 128.

The objective function constructs and trains the model with these parameters, using a fixed epoch count of 5.

Early Stopping
The training employs an EarlyStopping callback with val accuracy as the monitored metric. Training halts when this metric stops improving, enhancing efficiency and preventing overfitting.

Model Training
The model is trained using the KerasClassifier, with the best parameters suggested by Optuna. Training is performed on the IMDB dataset with the prepared training and test sets.

Fine-tuning
Fine-tuning of the model is accomplished through the iterative process of Optuna's study. The framework runs numerous trials (80 in this case), each time evaluating different hyperparameter combinations to maximize validation accuracy.

Best Parameters obtained after finetuning
filters: 256, kernel_size: 5, strides: 1, units: 128, kernel_initializer: glorot-uniform, rate: 0.1364690573129436, optimizer: rmsprop, batch_size': 64.

4.3 RNN Training Details [1]

Optuna is employed for hyperparameter optimization. The objective function suggests values for the following hyperparameters:

- **num_words**: The number of most frequent words to consider in the dataset.
- **maxlen**: The maximum length of the review sequences.
- **embedding_dim**: The dimensionality of the embedding layer.
- **rnn_units**: The number of units in the SimpleRNN layer.
- **optimizer_type**: The type of optimizer, either 'RMSprop' or 'Adam'.

Training Procedure:
The model is trained using the suggested hyperparameters. Each trial involves:

- Loading and preprocessing the dataset according to the current trial's hyperparameters
- Building and compiling the model with the current set of hyperparameters. Fitting the model on the training data for 5 epochs with a batch size of 128.
- Evaluating the model on the test set and returning the accuracy as the objective metric.

Optimization Study:
Optuna conducts an optimization study with a specified number of trials (50 in this case) to find the hyperparameters that maximize the accuracy on the test set.

Fine-tuning:
After the completion of the optimization study, the best hyperparameters are extracted. The model is then rebuilt and retrained using these optimal parameters, ensuring the best possible performance on the sentiment analysis task.

Best Parameters obtained after finetuning
num_words: 15000, maxlen: 350, embedding_dim: 16, rnn_units: 16, optimizer_type: 'rmsprop'

4.4 DistilBERT Training Details [2]

The training process is configured using TrainingArguments from the Hugging Face Transformers library. Key hyperparameters include:

- **output-dir**: This is where the model checkpoints and outputs are saved.
- **learning-rate**: Set at 2e-5, determining the step size at each iteration while moving towards a minimum of the loss function.
- **per_device_train_batch_size and per_device_eval_batch_size**: Both set at 16, defining the number of samples processed simultaneously on each device during training and evaluation.
- **num_train_epochs**: The number of training epochs is set to 5, indicating the number of times the learning algorithm will work through the entire training dataset.
- **weight_decay**: Set at 0.01 to reduce overfitting by applying regularization.
- **evaluation_strategy**: Configured to "epoch", meaning the evaluation occurs at the end of each training epoch.
- **save_strategy**: Set to "epoch", indicating model checkpoints are saved at the end of each epoch.
- **load_best_model_at_end**: A boolean indicating that the model checkpoint with the best performance on the validation set will be loaded at the end of training.

A Trainer object is created, encapsulating all aspects of training. It was configured with the model, training arguments, datasets, tokenizer, data collator, and a custom metric computation function.

The training is initiated using the trainer.train() method.

There is no dedicated hyperparameter tuning done on this model.

5 Results and Observations

The models are run on 5 epochs. Once the best hyperparameters are found, the model is refit and the validation set accuracy is determined. There is no test set accuracy as the test data is utilized as validation data, in essence, it is completely in-domain data.

Table 1: Vaidation accuracy on 5 epochs

CNN	RNN	DistilBERT
90.5%	87.9%	**93.2%**

The two best models are DistilBERT and CNN. It is expected that the transformer model will outperform CNN and RNN, especially since their architectures are simple with only a few layers. Further, the Transformer model self-attention mechanism helps the model to understand and learn better in terms of context and label.

CNN outperforming RNN is also in line given the CNN architecture implemented had multiple layers with regularization.

6 Conclusion

The purpose of this experiment was to learn, understand, and implement different deep learning systems to perform sentiment analysis. Hence that purpose is achieved and the accuracy values obtained provide a good ablation across systems. This experiment provides an opportunity to get acquainted with the implementation of deep learning systems in a way that is essentially different from that of classical experiments on numerical data. This experiment also allows one to learn how to work with Natural Language datasets in a simple and pedagogical way.

7 Challenges faced and future work

We faced several situations while working on this project, that essentially put us at a crossroads. The first such instance is while choosing the dataset. The debate was whether we should go with a standard CSV file or whether it is more logical to utilize the dataset loaded in Keras and HuggingFace. We ended up going with the library option as the data has already been cleaned and a minimal amount of pre-processing has been done. The next step is to decide on the tokenization scheme while implementing CNN and RNN systems. We have initially experimented with TF-IDF tokenization and CountVectorizer. However, these approaches gave poor accuracies and on further analysis using the 'decode' custom function, we realized that we need to find a different way. Hence on further research, we decided to experiment with the word index tokenization which is more in line with the usage of the imdb dataset. On experimentation, this approach has produced higher accuracies compared to the previous methods.

We have decided to keep our systems simple and perform an exhaustive hyper-parameter tuning. This is where we encountered the next challenge which is a bottleneck problem due to the RAM on GoogleColab maxing out during exhaustive GridSearch hyperparameter tuning process. On further research, we have decided to run our hyperparameter tuning using the Optuna Framework. This helped us circumvent the RAM capacity bottleneck and reach the best parameters for CNN and RNN.

Future work on this project could be to experiment on complex CNN and RNN systems and to perform hyperparameter tuning on the transformer model. We have not performed tuning on different learning rates and batch sizes on the transformer model due to its dense nature and the time it takes to run the model (approximately 2 hours for 5 epochs without using Optuna).

References

[1] Keras Documentation

[2] HuggingFace Documentation

[3] Optuna Documentation