Sage Plagarism Report

Powered by schoolworkspro.com

Hari Sharan Shrestha
150212
Word Count: 6908

26
Results Found

12.75 %
Match Percentage

Source: 4 Link: https://web.stanford.edu/~jurafsky/slp3/4.pdf	**0.13%**
Submitted to: Softwarica	**3.17%**
Submitted to: Softwarica	**0.88%**
Submitted to: Softwarica	**0.88%**
Submitted to: Softwarica	**0.8%**
Source: Sentiment Analysis | Comprehensive Beginners Guide | Thematic | Thematic Link: https://getthematic.com/sentiment-analysis/	**0.77%**
Source: Traffic sign recognition using CNN / Habr Link: https://habr.com/en/post/706134/	**0.6%**
Submitted to: Softwarica	**0.52%**
Source: Naive Bayes Classifier in Machine Learning - Javatpoint Link: https://www.javatpoint.com/machine-learning-naive-bayes-classifier#:~:text=Naïve%20Bayes%20Classifier%20is%20one,the%20probability%20of%20an%20object	**0.46%**
Submitted to: Softwarica	**0.44%**
Submitted to: Softwarica	**0.42%**
Submitted to: Softwarica	**0.36%**
Source: Plot With Pandas: Python Data Visualization for Beginners – Real Python Link: https://realpython.com/pandas-plot-python/	**0.36%**
Submitted to: Texas	**0.34%**

Submitted to: Softwarica	**0.33%**
Submitted to: Sunway	**0.31%**
Source: How to Use Sentiment Analysis to Manage Your Brand Reputation & Win Customers Link: https://brandmentions.com/blog/sentiment-analysis/	**0.26%**
Submitted to: Softwarica	**0.26%**
Submitted to: Softwarica	**0.25%**
Submitted to: Softwarica	**0.23%**
Submitted to: Softwarica	**0.21%**
Submitted to: Softwarica	**0.2%**
Source: Array in C Link: https://en.wikipedia.org/wiki/Hadamard_product_(matrices)	**0.16%**
Submitted to: Softwarica	**0.16%**
Submitted to: Softwarica	**0.15%**
Source: What is Sentiment Analysis? - Sentiment Analysis Explained - AWS Link: https://aws.amazon.com/what-is/sentiment-analysis/	**0.1%**

REFLECTIVE REPORT

Sentiment Analysis using LSTM STW7088CEM – Artificial Neural

Sentiment Analysis using LSTM STW7088CEM – Artificial Neural

Name: Hari sharan shrestha College ID: 150212 Git Hub Repository Link: https://github.com/softwarica-github/ 150212_hari_sharan_stw7088cem

Contents

Abstract Sentiment analysis, a subfield of natural language processing (NLP), involves determining the sentiment or opinion expressed in text data, such as movie reviews, social media posts, or customer feedback.
In this in-depth exploration, we delve into the world of sentiment analysis using Long Short-Term Memory (LSTM) neural networks.
In this in-depth exploration, we delve into the world of sentiment analysis using Long Short-Term Memory (LSTM) neural networks.
Our primary goal is to equip you with a deep understanding of how to apply neural networks to solve real-world sentiment analysis problems.
Manually analyzing large amounts of textual data is more difficult and time- consuming.
Sentiment analysis employs artificial intelligence (AI) in an automated manner to identify both positive and negative views derived from the text.
Sentiment analysis is frequently employed in gaining knowledge from comments on social media, poll results, and product evaluations to produce data-driven choices.
Systems for sentiment analysis are used to adding up to the unstructured text by saving time and streamlining business procedures hours of processing by hand.
Lately, Deep Learning has (DL) has attracted growing interest in the sector and intellectual community for its exceptional achievement throughout many fields.
Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) The most often used kinds of DL are neural networks (CNNs).
Sentiment analysis of text reviews is performed by with Long-Term Short-Term
Introduction Sentiment analysis is the process of determining the sentiment or opinion expressed in a piece of text, such as a movie review, tweet, or comment.
It categorizes text as positive, negative, or neutral.
LSTM, a type of recurrent neural network (RNN), is well-suited for this task because it can capture sequential information in text data.
Long Short-Term Memory Networks is a deep learning, sequential neural network that allows information to persist.
It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN.
LSTM was designed by Hoch Reiter and Schmidhuber that resolves the problem caused by traditional rnns and machine learning algorithms.
Sentiment analysis is the automated application of higher order cognitive processes to a few provided opinions.
Topics taken from a written record in an extraordinarily current each generation, we produce around 1.5 quintillion bytes of data.
To provide important insights and automate all procedures for their expansion as a business.
Moreover, sentiment analysis is known as "opinion mining."

Not only is sentiment analysis a not just sentiment analysis but also text contextual analysis, which finds and extracts arbitrary data from the source material and assisting a company in understanding the societal attitude of their brand, product, or service while keeping an eye on the internet

Literature Review

AUTHOR TOPIC MEASUREMENT DESCRIPTION

BoPang, Lillian Lie A Sentimental Education: Sentiment Analysis using subjectivity summarizatio n based on Minimum Cut.

Accuracy (Performance)

Accuracy (Performance)

Accuracy (Performance)

This paper proposed that SVM and NB are better technique for improving the performance of a model up to 86.4 %.

Erik.Boiy, Pieter Hens, Koen Dschacht, Marie Francine Moens

Automatic Sentiment Analysis in Online Text

Accuracy (speed and size)

This paper shows the varying level of accuracy when symbolic and machine learning methods were applied to different social network dataset.

Doreen Hii Using Meaning specificity to aid negation handling in sentiment analysis

Accuracy (Performance)

Accuracy (Performance)

Accuracy (Performance)

This paper compared the accuracy of 1-,2-,3-,4-, gram and suggested 4- gram is the best performing window size in Lexicon method

Dr. Sefer Kurnaz, and Mustafa Ahmed Mahmood.

Sentiment Analysis in data of Twitter using Machine learning algorithm

Accuracy (Time) This paper proposes a new technique which offers an accuracy of 98 % when compared with Deep learning method, SVM and Maximum Entropy method.

Xiaomie Zou, Jing Yang, and Jianpei Zhang.

Microblog sentiment analysis using social and topic context.

Accuracy (Performance)

Accuracy (Performance)

Accuracy (Performance)

This paper proposes a new method to identify the polarity of the sentiment and shows the structure similarity has a better accuracy than user direct relations.

Proposed work As a synthetic recurrent memory, long short-term memory (LSTM) neural network (RNN) architecture that is used in the deep learning domain.

Contrary to conventional feed forward neural networks, LSTM is connected to feedback.

LSTM networks are useful for processing, categorizing, and generating forecasts based on statistical data since there could be unexplained gaps between significant occurrences in a extremely high figure.

LSTMs were created to take into account the issues with fading and expanding gradients that might faced while conventional RNNs are being trained.

Associated one benefit of LSTM over RNNs is its insensitivity to gap length.

Hidden Markov models and other techniques for learning sequences in a variety of settings.

Numerous architectures exist for units of LSTM.

An architecture typically consists of a cell (the three "regulators" and memory, a component of the LSTM unit normally forget gate-equipped LSTM
The condensed versions of the forward pass equations for an LSTM units have a forget gate:
Where c0=0 and h0=0 are the starting values as well as the operator.
The element-wise product, or Hadamard product, is shown by o.
The time step is represented by subscript t. The model in which σ is the Tanh is the hyperbolic tangent and sigmoid activation function, Wi, Wc, Wf, Wo, Ui, Xt the input at time t, Weight matrices Uc, Uf, and Uo are used to control the input, and bi, bc, The bias vectors are bo and bf.
Architecture of proposed work used
Raw Text The IMDB movie reviews datasets are what we utilize for model validation and training.
These datasets include tweets classified as good or negative overall.
We only load it if it is saved on your computer as a text file.
The text is then changed to lower case, and all punctuation is eliminated.
All of the strings are together in one very large string.
Individual reviews must now be kept apart and stored in different list components.
Tokenization The process of tokenizing, or dividing a text string into an inventory of tokens, is known as tokenization.
Tokenization The process of tokenizing, or dividing a text string into an inventory of tokens, is known as tokenization.
It is possible to think of tokens as component pieces; for example, a word may be a token in a phrase, and a sentence could be a token in a paragraph.
Tokenization, or breaking a string into its intended components, is essential to all NLP activities.
Tokenization, or breaking a string into its intended components, is essential to all NLP activities.
Tokenization has made it such that there is no one right.
The application determines which algorithm is appropriate.
Since sentiment data is frequently given infrequently and sparsely, I believe tokenization is even more crucial in sentiment analysis than it is in other NLP domains.
For example, a single cluster of punctuation like >:-(might convey an entire message.
Make a dictionary that maps vocabulary to integers.
Build an index mapping dictionary in the majority of NLP activities such that the terms that appear most often are given lower indices.
Using the Collections library's Counter function is one of the most popular ways to accomplish this.
Encrypt the text Using vocabulary from all of our evaluations, we have so far produced an index mapping dictionary and a list of reviews.
All we did was generate an encoding of our reviews by substituting numbers for the words in them; the result is a list of lists.
Every review is a list of floating-point or in Encrypt the Label Since there are just two output labels, this is easy.
Thus, we shall simply designate "positive" as 1 and "negative" as 0.
Using this class, a text corpus can be vectored by converting each text into one of two formats: either a vector with a binary coefficient for each token based on word count or term frequency-inverse document frequency, or a sequence of integers, where each integer represents a token in a dictionary.
Embedding The subject of Natural Language Processing (NLP), which combines computer science, artificial intelligence, and machine learning, is where the phrase "embedding" first

appeared.
And language computation.
An embedded word is a text mining method for figuring up word relationships textual information (Corpus).
The semantic and syntactic interpretations are revealed by the context in which they are employed.
According to the idea of distributional hypothesis, words having meaningful similarities when happening in comparable contexts.
Tally prediction-based embedding's and based embedding's are the two major methods for word embedding.
Insertions express connections via words.
The embedded are thick characters represented as vectors.
Dense vectors are created by the embedding layer from integer indices 128 in length.
Dimension of input: The size of the vocabulary, expressed as the quantity the most used terms.
Softmax There are limits to how well softmax layers can determine multi-class probability.
The more courses there are, the more costly softmax may get.
Candidate sampling may be a better remedy in some circumstances.
A softmax layer can restrict the range of its computations to a certain class set by using candidate sampling.
For instance, just the apples in a picture of a bowl of fruit need to have their probability estimated; other fruit types do not need to be considered.
Furthermore, a softmax layer will not function in scenarios when an item belongs to many classes since it thinks that there is only one member per class.
Alternatively, multiple logistic regressions might be used in that situation.
Qualities the multi classification model uses the softmax function, which yields the likelihood of each class as well as the likelihood that the target class will have a high probability.
The formula calculates the total of the exponential values of all the values in the inputs as well as the exponential (e-power) of the supplied input value.
The output of the SoftMax function is thus the ratio of the exponential of the input value and also the sum of the exponential values.
It is used in the various levels of neural networks and for multi-classification tasks.
Compared to other values, the high value will have a higher likelihood.
It's also possible for a neural network to be trying to determine whether a photo contains a dog.
It need to be able to provide a likelihood that
Algorithm Loading Data Begin by importing the necessary libraries and loading the IMDb movie review dataset, which consists of 50,000 movie reviews labeled as positive or negative sentiments.
The provided code reads a CSV (Comma-Separated Values) file containing movie reviews from the IMDb dataset and then displays the first few rows of the dataset using the pandas library in Python.
Let's break down the code step by step: base_csv = /MDB Dataset.csv': This line defines a variable base_csv that stores the file path of a CSV file named "IMDB Dataset.csv."
The file path appears to be an absolute path pointing to a location of file.
df = pd.read_csv(base_csv): Here, the code uses the pandas library to read the CSV file located at the path specified in the base_csv variable.
The pd.read_csv() function reads the CSV file and creates a DataFrame, a data structure provided by pandas for working with tabular data.

The resulting DataFrame is assigned to the variable df.
df.head(): Finally, the code calls the head() method on the DataFrame df.
This method is used to display the first few rows of the DataFrame, providing a quick overview of the dataset's structure and content.
By default, it shows the first five rows, but you can specify the number of rows to display within the parentheses if desired.
Splitting Data Splitted the dataset into training and test sets, ensuring that the distribution of sentiment labels (positive and negative) is maintained in both sets.
X, y = df['review'].values, df['sentiment'].values: This line extracts two arrays from a DataFrame df.
X is assigned the values from the 'review' column of the DataFrame, which presumably contains the movie reviews as text.
y is assigned the values from the 'sentiment' column of the DataFrame, which presumably contains the corresponding sentiment labels (e.g., 'positive' or 'negative') for each review.
x_train, x_test, y_train, y_test = train_test_split(X, y, stratify=y): This line splits the data into training and testing sets using the train_test_split function from the scikit- learn library.
The stratify parameter is used to ensure that the class distribution (the distribution of 'sentiment' labels) is preserved in the training and testing sets.
The stratify parameter is used to ensure that the class distribution (the distribution of 'sentiment' labels) is preserved in the training and testing sets.
The stratify parameter is used to ensure that the class distribution (the distribution of 'sentiment' labels) is preserved in the training and testing sets.
X and y are the data and labels, respectively, which are split into training and testing data (x_train and x_test) and training and testing labels (y_train and y_test).
print(f'shape of train data is {x_train.shape}'): This line prints the shape of the training data (i.e., the x_train array).
The shape typically indicates the number of samples and features.
In this case, it shows the number of reviews in the training set.
print(f'shape of test data is {x_test.shape}'): This line prints the shape of the testing data (i.e., the x_test array).
Similar to the previous line, it shows the number of reviews in the testing set.
Analyzing Sentiment The provided code generates a bar plot that visualizes the distribution of sentiment labels (positive and negative) in the training data.
Analyzing Sentiment The provided code generates a bar plot that visualizes the distribution of sentiment labels (positive and negative) in the training data.
Analyzing Sentiment The provided code generates a bar plot that visualizes the distribution of sentiment labels (positive and negative) in the training data.
Let's break down the code step by step:
Let's break down the code step by step:
dd = pd.Series(y_train).value_counts(): This line of code computes the count of each unique sentiment label in the training data (y_train) using the value_counts() function.
The result is stored in a pandas Series called dd.
The dd Series will have two entries, one for 'negative' sentiment and one for 'positive' sentiment, with corresponding counts.
sns.barplot(x=np.array(['negative','positive']), y=dd.values): This line uses the Seaborn library (sns) to create a bar plot.
The x argument specifies the labels for the x-axis of the bar plot.

In this case, it's an array of two strings, 'negative' and 'positive,' representing the sentiment categories.
The y argument provides the values for the height of the bars on the y-axis.
These values are extracted from the dd Series using dd.values, representing the counts of each sentiment category.
plt.show(): This line displays the bar plot.
plt.show(): This line displays the bar plot.
The plt.show() function is used to render and show the plot to the user.
The plt.show() function is used to render and show the plot to the user.
Tokenization and Vocabulary Creation The provided code defines two functions that are used for text preprocessing and tokenization.
These functions are typically used in natural language processing (NLP) tasks, such as sentiment analysis, to clean and prepare text data for further analysis.
Let's describe each function:
preprocess_string(s): This function takes a string s as input and performs several text preprocessing steps to clean the text.
Here's what each step does:
re.sub(r"[^\w\s]", '', s): This step removes all non-word characters (everything except numbers and letters) from the string s. It uses regular expressions to match and replace non-alphanumeric characters with an empty string.
re.sub(r"\s+", ' ', s): This step replaces all runs of white spaces with no space.
re.sub(r"\s+", ' ', s): This step replaces all runs of white spaces with no space.
It ensures that multiple consecutive spaces are reduced to a single space.
re.sub(r"\d", '', s): This step replaces digits with no space.
It removes numeric characters from the string.
The function returns the preprocessed string s.
tokenize(x_train, y_train, x_val, y_val): This function takes four input arguments:
x_train: A list of training text data.
y_train: A list of labels corresponding to the training data.
x_val: A list of validation text data.
y_val: A list of labels corresponding to the validation data.
Here's what the function does:
It initializes an empty list word_list to store words from the training data.
It imports a set of English stop words using the stopwords.words('english') function.
Stop words are common words (e.g., "the," "and," "is") that are often removed from text data because they carry little meaningful information.
It processes each sentence in the x_train data: It converts the sentence to lowercase using .lower() and splits it into individual words.
It applies the preprocess_string function to each word to remove non-alphanumeric characters, extra spaces, and digits.
If the processed word is not a stop word and is not an empty string, it is added to the word_list.
It creates a Counter object corpus to count the frequency of each word in the word_list.
It sorts the words in corpus in descending order of frequency, selecting the top 1000 most common words as corpus_.
It creates a one-hot encoding dictionary (onehot_dict) that maps each word in corpus_ to a unique numerical index.

It tokenizes the training and validation data: For each sentence in x_train and x_val, it processes each word using the preprocess_string function and checks if the word is in the onehot_dict.
If the word is in the dictionary, it replaces the word with its corresponding numerical index from onehot_dict.
It encodes the labels: It assigns the value 1 to positive labels and 0 to negative labels in both the training and validation label lists (y_train and y_val).
It encodes the labels: It assigns the value 1 to positive labels and 0 to negative labels in both the training and validation label lists (y_train and y_val).
The function returns the following arrays: final_list_train: Tokenized training data.
encoded_train: Encoded training labels.
final_list_test: Tokenized validation data.
encoded_test: Encoded validation labels.
onehot_dict: The one-hot encoding dictionary.
These functions are important for preparing text data for further processing, such as feeding it into machine learning models, including neural networks for tasks like sentiment analysis.
The provided code snippet calls the tokenize function, which you previously defined, to preprocess and tokenize text data for a sentiment analysis task.
Let's break down the code and its purpose:
x_train, y_train, x_test, y_test, vocab = tokenize(x_train, y_train, x_test, y_test) Here's what each part of the code does:
x_train, y_train, x_test, y_test, vocab = tokenize(x_train, y_train, x_test, y_test) Here's what each part of the code does:
x_train, y_train, x_test, y_test, vocab = tokenize(x_train, y_train, x_test, y_test) Here's what each part of the code does:
x_train: This variable represents the training data, typically containing movie reviews or textual content on which you want to perform sentiment analysis.
It is expected to be a list of text data.
y_train: This variable represents the training labels, which are the corresponding sentiment labels for the training data.
For sentiment analysis, these labels are often binary, indicating whether a review is "positive" or "negative."
x_test: This variable represents the testing data, similar to x_train, containing a different set of movie reviews or text data.
y_test: This variable represents the testing labels, which are the corresponding sentiment labels for the testing data.
vocab: This variable is used to store the vocabulary or one-hot encoding dictionary created during the preprocessing of the text data.
This dictionary maps words to unique numerical
indices, allowing text data to be converted into numerical representations for machine learning models.
The tokenize function is called with these variables as arguments, and it performs the following tasks, which have been described earlier: Tokenizes the text data (both training and testing sets).
Encodes the labels to binary values (e.g., 1 for "positive" and 0 for "negative").
Encodes the labels to binary values (e.g., 1 for "positive" and 0 for "negative").
Creates a one-hot encoding dictionary (vocab) based on the most common words in the training data.

Returns the tokenized data, encoded labels, tokenized testing data, encoded testing labels, and the vocabulary.
Analyzing Review Length The provided code snippet is used to analyze the lengths of movie reviews (or text data) in the training set (x_train).
It calculates the length of each review, creates a histogram to visualize the distribution of review lengths, and then provides a summary of the statistics for these lengths.
It calculates the length of each review, creates a histogram to visualize the distribution of review lengths, and then provides a summary of the statistics for these lengths.
Let's break down the code step by step:
Let's break down the code step by step:
rev_len = [len(i) for i in x_train]:
This line creates a list called rev_len using a list comprehension.
It iterates over each element (i) in the x_train list, which contains tokenized movie reviews.
For each review, it calculates its length using the len(i) function, which counts the number of tokens (words) in the review.
The resulting list rev_len contains the lengths of all the movie reviews in the training set.
pd.Series(rev_len).hist():
This line creates a Pandas Series from the rev_len list.
This line creates a Pandas Series from the rev_len list.
This Series contains the review lengths.
It then calls the .hist() method on the Series to create a histogram.
A histogram is a graphical representation of the distribution of data, in this case, the distribution of review lengths.
A histogram is a graphical representation of the distribution of data, in this case, the distribution of review lengths.
The histogram shows the frequency of review lengths on the x-axis and the number of reviews with those lengths on the y-axis.
plt.show():
plt.show():
This line displays the histogram using plt.show().
It is assumed that you have imported the matplotlib.pyplot library as plt.
pd.Series(rev_len).describe():
This line creates a Pandas Series from the rev_len list.
This line creates a Pandas Series from the rev_len list.
It calls the .describe() method on the Series to generate summary statistics about the review lengths.
The summary typically includes statistics like the mean, standard deviation, minimum, 25th percentile, median, 75th percentile, and maximum review length.
Observations : a) Mean review length = around 69. b) minimum length of reviews is 2. c)There are quite a few reviews that are extremely long, we can manually investigate them to check whether we need to include or exclude them from our analysis.
Padding To ensure that all sequences have the same length, you pad the tokenized sequences.
Sequences longer than a certain length (500 in this case) are truncated, and sequences shorter are padded with zeros.
Batching and Loading Data
Tensor datasets and data loaders for training and validation data.
This enables efficient batch processing for model training.

LSTM Model Defined the SentimentRNN class, which represents the LSTM-based sentiment analysis model.
LSTM Model Defined the SentimentRNN class, which represents the LSTM-based sentiment analysis model.
The model consists of an embedding layer, an LSTM layer, a dropout layer, a fully connected layer, and a sigmoid activation function.
The provided code defines a Python class called SentimentRNN, which is a neural network model for sentiment analysis using recurrent neural networks (RNNs), specifically Long Short-Term Memory (LSTM) cells.
This class is designed to be used for sentiment classification tasks.
Let's break down the code and explain its components: class SentimentRNN(nn.Module): This line defines a new class named SentimentRNN, which is a subclass of nn.Module.
In PyTorch, neural network models are typically defined as classes that inherit from nn.Module. This allows you to leverage PyTorch's functionalities for building and training neural networks.
def __init__(self, no_layers, vocab_size, hidden_dim, embedding_dim, drop_prob=0.5): This is the constructor method of the class, which is called when an instance of the class is created.
It takes several parameters to configure the model: no_layers: The number of LSTM layers in the model.
vocab_size: The size of the vocabulary, which indicates the number of unique words in the input data.
hidden_dim: The dimension of the hidden states in the LSTM cells.
embedding_dim: The dimension of word embeddings for input words.
drop_prob: The probability of dropout, which is a regularization technique to prevent overfitting (default value is 0.5).
Inside the constructor, the following components are defined: self.output_dim and self.hidden_dim: These attributes store the output dimension and hidden dimension of the model.
self.no_layers and self.vocab_size: These attributes store the number of layers and vocabulary size, respectively.
self.embedding: This is an embedding layer created using nn.Embedding.
It converts word indices into dense word embeddings.
self.lstm: This is an LSTM layer created using nn.LSTM, which will process the embedded words.
self.lstm: This is an LSTM layer created using nn.LSTM, which will process the embedded words.
self.dropout: A dropout layer with a specified dropout probability.
self.fc: A fully connected (linear) layer used for the final prediction.
self.sig: A sigmoid activation function to produce binary sentiment predictions.
self.sig: A sigmoid activation function to produce binary sentiment predictions.
self.sig: A sigmoid activation function to produce binary sentiment predictions.
def forward(self, x, hidden): This method defines the forward pass of the model, which specifies how input data is processed to produce output predictions.
It takes two arguments:
x: The input data, which is typically a sequence of word indices.
hidden: The initial hidden state of the LSTM.
Inside the forward method, the following operations are performed:
The input data is embedded using the embedding layer.

The embedded data is passed through the LSTM layer to obtain LSTM outputs.
The LSTM outputs are reshaped to have the same number of features as the hidden dimension.
Dropout is applied to prevent overfitting.
The fully connected layer (self.fc) produces the final prediction scores.
The sigmoid function is applied to obtain the binary sentiment prediction.
The final prediction is returned, along with the updated hidden state.
def init_hidden(self, batch_size): This method initializes the hidden state for the LSTM.
It takes the batch size as an argument and returns an initial hidden state, including the hidden state (h0) and cell state (c0) of the LSTM.
This method is typically used at the beginning of processing each batch of data.
Training This code snippet deals with training and evaluating a sentiment analysis model using PyTorch.
Training This code snippet deals with training and evaluating a sentiment analysis model using PyTorch.
It encompasses the following tasks:
Loss and Optimization Functions: lr = 0.001: Learning rate, a hyperparameter that controls the step size during optimization.
criterion = nn.BCELoss(): Binary Cross-Entropy Loss, a common loss function for binary classification tasks.
optimizer = torch.optim.Adam(model.parameters(), lr=lr): Adam optimizer, used to update the model's parameters during training.
Accuracy Calculation Function: def acc(pred, label): This function takes predicted values (pred) and true labels (label) and calculates the accuracy of the predictions.
It rounds the predicted values to the nearest integer (0 or 1) and compares them to the true labels.
The function returns the number of correct predictions.
Training Loop:
clip = 5: This value is used for gradient clipping, a technique to prevent exploding gradients in recurrent neural networks (RNNs) or LSTMs.
epochs = 5: The number of training epochs, which is the number of times the model goes through the entire training dataset.
valid_loss_min = np.Inf: A variable to keep track of the minimum validation loss.
It is initialized to positive infinity to ensure that the model's initial loss is considered the minimum.
Training Loop: The code runs a loop for a specified number of epochs (5 in this case).
Training Loop: The code runs a loop for a specified number of epochs (5 in this case).
Inside the loop, training and validation losses, as well as training and validation accuracies, are recorded for each epoch.
Inside the loop, training and validation losses, as well as training and validation accuracies, are recorded for each epoch.
Training Phase: For each training epoch, the model is set to training mode using model.train().
The hidden state for the LSTM is initialized.
The hidden state for the LSTM is initialized.
The training data is processed in batches using the train_loader, and for each batch: The inputs and labels are moved to the specified device (e.g., GPU).
The model's gradients are set to zero using model.zero_grad().
The model makes predictions and calculates the loss.

Backpropagation is performed to compute gradients.
Training loss and accuracy are recorded.
Gradient clipping is applied using nn.utils.clip_grad_norm_ to prevent exploding gradients.
Model parameters are updated using the optimizer's step() method.
Validation Phase:
For each validation epoch, the model is set to evaluation mode using model.eval().
The validation data is processed in batches using the valid_loader, and for each batch: The inputs and labels are moved to the specified device.
The model makes predictions and calculates the validation loss.
Validation accuracy is recorded.
After each epoch, the mean training and validation losses, as well as training and validation accuracies, are computed and recorded.
If the validation loss is lower than the previously recorded minimum (valid_loss_min), the model's state dictionary is saved to a file (state_dict.pt), effectively saving the best model.
Finally, a plot is generated to visualize the training and validation accuracy and loss over the epochs.
Inference This code snippet defines a function predict_text(text) and demonstrates how to use this function to predict sentiment labels for two specific text samples from the dataset.
Inference This code snippet defines a function predict_text(text) and demonstrates how to use this function to predict sentiment labels for two specific text samples from the dataset.
Inference This code snippet defines a function predict_text(text) and demonstrates how to use this function to predict sentiment labels for two specific text samples from the dataset.
Here's an explanation of the code:
def predict_text(text): This function takes a text as input and is used to predict the sentiment label for that text.
It follows these steps:
word_seq is a NumPy array that stores the one-hot encoded representation of the input text.
It processes the text by splitting it into words and checking if each word is present in the vocabulary (vocab).
If a word is in the vocabulary, it retrieves the corresponding one-hot encoded value; otherwise, it's skipped.
word_seq is then expanded along a new axis to make it suitable for processing as input to the model.
pad is created using padding_() to ensure that the input data has a consistent shape (in this case, a sequence length of 500 words).
inputs is created as a PyTorch tensor from pad, and batch_size is set to 1.
The hidden state h is initialized using the model's init_hidden() method.
This is necessary before making predictions.
The model (model) is used to make predictions by passing the inputs and hidden state.
The result is stored in output.
The function returns the predicted sentiment label as a float value using output.item().
This value can be interpreted as the model's confidence in predicting a positive sentiment.
A higher value indicates a stronger positive sentiment prediction, while a lower value implies a stronger negative sentiment prediction.
The code then proceeds to demonstrate the use of the predict_text function for two specific text samples from the dataset: index = 30 and index = 32 are selected as sample indices.
The text content at these indices is printed using df['review'][index].

The actual sentiment labels from the dataset are printed using df["sentiment"][index].
The predict_text function is called for these text samples to obtain the predicted sentiment label and the associated probability.
The predict_text function is called for these text samples to obtain the predicted sentiment label and the associated probability.
The predict_text function is called for these text samples to obtain the predicted sentiment label and the associated probability.
The code then determines whether the predicted sentiment is "positive" or "negative" based on whether the predicted probability is greater than 0.5.
It also calculates the probability of the opposite sentiment and prints both the predicted sentiment and the associated probabilities.
Conclusion Sentiment analysis using LSTM is a powerful application of deep learning and NLP.
By following the steps outlined in this guide and the code you provided, you can build a robust sentiment analysis system.
By following the steps outlined in this guide and the code you provided, you can build a robust sentiment analysis system.
The model is capable of classifying text data as positive or negative sentiments, making it a valuable tool for analyzing reviews, comments, and other forms of textual feedback.
The model is capable of classifying text data as positive or negative sentiments, making it a valuable tool for analyzing reviews, comments, and other forms of textual feedback.
This comprehensive guide demonstrates the complete pipeline for sentiment analysis using LSTM, from data preprocessing and model building to training and inference.
This comprehensive guide demonstrates the complete pipeline for sentiment analysis using LSTM, from data preprocessing and model building to training and inference.
It provides a strong foundation for understanding and implementing sentiment analysis in real-world applications.
In this research, I have chosen an LSTM-based technique for text data sentiment categorization.
Global users openly share and express their thoughts on a variety of issues.
Since it is exceedingly difficult to analyze huge volumes of this kind of data manually, there is a legitimate need for computer processing of the data.
Sentiment analysis examines how individuals feel about various goods and services, political issues, social gatherings, and business tactics.
Text documents with reviews (from sites like IMDB) and social network posts (mainly from Facebook and Twitter) are the most promising for sentiment analysis.
More training data improves the performance of deep learning techniques like long short term memory (LSTM), which can classify sentiment with an accuracy of 85%.
References (IJERT), I. J. o. E. R. &.
T., 2023.
Text based Sentiment Analysis using LSTM.
Text based Sentiment Analysis using LSTM.
[Online] Available at: https://www.researchgate.net/publication/341873850_Text_based_Sentiment_Analysis_using_LSTM [Accessed 20 10 2023].
Anon., 2023.
Large Movie Review Dataset.
[Online] Available at: http://ai.stanford.edu/~amaas/data/sentiment/ [Accessed 15 10 2023].
Narkhede, S., 2023.

Understanding Confusion Matrix.
[Online] Available at: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62 [Accessed 10 10 2023].
Sepp Hochreiter, J. S., 2023.
Long Short-Term Memory.
[Online] Available at: https://direct.mit.edu/neco/article-abstract/9/8/1735/6109/Long-Short-Term- Memory?redirectedFrom=fulltext [Accessed 1 11 2023].
Appendix import numpy as np # linear algebra import pandas as pd # data processing, CSV file I/O (e.g.
pd.read_csv) import torch import torch.nn as nn import torch.nn.functional as F from nltk.corpus import stopwords from collections import Counter import string import re import seaborn as sns from tqdm import tqdm import matplotlib.pyplot as plt from torch.utils.data import TensorDataset, DataLoader from sklearn.model_selection import train_test_split
is_cuda = torch.cuda.is_available()
# If we have a GPU available, we'll set our device to GPU.
We'll use this device variable later in our code.
if is_cuda: device = torch.device("cuda") print("GPU is available") else: device = torch.device("cpu") print("GPU not available, CPU used")
base_csv = '../input/IMDB Dataset.csv' df = pd.read_csv(base_csv) df.head()
# SPLITTING TO TRAIN AND TEST DATA X,y = df['review'].values,df['sentiment'].values
x_train,x_test,y_train,y_test = train_test_split(X,y,stratify=y) print(f'shape of train data is {x_train.shape}') print(f'shape of test data is {x_test.shape}')
# ANALYSING SENTIMENT dd = pd.Series(y_train).value_counts()
sns.barplot(x=np.array(['negative','positive']),y=dd.values) plt.show()
# ANALYSING SENTIMENT dd = pd.Series(y_train).value_counts()
sns.barplot(x=np.array(['negative','positive']),y=dd.values) plt.show()
# ANALYSING SENTIMENT dd = pd.Series(y_train).value_counts()
sns.barplot(x=np.array(['negative','positive']),y=dd.values) plt.show()
# TOCKENIZATION def preprocess_string(s): # Remove all non-word characters (everything except numbers and letters) s = re.sub(r"[^\w\s]", '', s)
# Replace all runs of whitespaces with no space s = re.sub(r"\s+", '', s) # replace digits with no space s = re.sub(r"\d", '', s)
# Replace all runs of whitespaces with no space s = re.sub(r"\s+", '', s) # replace digits with no space s = re.sub(r"\d", '', s)
def tockenize(x_train,y_train,x_val,y_val): word_list = []
stop_words = set(stopwords.words('english')) for sent in x_train: for word in sent.lower().split(): word = preprocess_string(word) if word not in stop_words and word != '':
word_list.append(word) corpus = Counter(word_list) # sorting on the basis of most common words corpus_ = sorted(corpus,key=corpus.get,reverse=True)[:1000] # creating a dict onehot_dict = {w:i+1 for i,w in enumerate(corpus_)} # tockenize final_list_train,final_list_test = [], [] for sent in x_train: final_list_train.append([onehot_dict[preprocess_string(word)] for word in sent.lower().split() if preprocess_string(word) in onehot_dict.keys()]) for sent in x_val: final_list_test.append([onehot_dict[preprocess_string(word)] for word in sent.lower().split() if preprocess_string(word) in onehot_dict.keys()]) encoded_train = [1 if label =='positive' else 0 for label in y_train] encoded_test = [1 if label =='positive' else 0 for label in y_val] return np.array(final_list_train), np.array(encoded_train),np.array(final_list_test), np.array(encoded_test),onehot_dict

x_train,y_train,x_test,y_test,vocab = tockenize(x_train,y_train,x_test,y_test)
x_train,y_train,x_test,y_test,vocab = tockenize(x_train,y_train,x_test,y_test)
x_train,y_train,x_test,y_test,vocab = tockenize(x_train,y_train,x_test,y_test)
print(f'Length of vocabulary is {len(vocab)}')
# ANALYSING REVIEW LENGTH rev_len = [len(i) for i in x_train] pd.Series(rev_len).hist()
plt.show() pd.Series(rev_len).describe()
# PADDING def padding_(sentences, seq_len): features = np.zeros((len(sentences),
seq_len),dtype=int) for ii, review in enumerate(sentences): if len(review) != 0: features[ii, -
len(review):] = np.array(review)[:seq_len] return features
#we have very less number of reviews with length > 500.
#So we will consideronly those below it.
x_train_pad = padding_(x_train,500) x_test_pad = padding_(x_test,500)
x_train_pad = padding_(x_train,500) x_test_pad = padding_(x_test,500)
x_train_pad = padding_(x_train,500) x_test_pad = padding_(x_test,500)
# BATCHING AND LOADING AS TENSOR # create Tensor datasets train_data =
TensorDataset(torch.from_numpy(x_train_pad), torch.from_numpy(y_train)) valid_data =
TensorDataset(torch.from_numpy(x_test_pad), torch.from_numpy(y_test))
# dataloaders batch_size = 50
# make sure to SHUFFLE your data train_loader = DataLoader(train_data, shuffle=True,
batch_size=batch_size) valid_loader = DataLoader(valid_data, shuffle=True,
batch_size=batch_size)
# obtain one batch of training data dataiter = iter(train_loader) sample_x, sample_y =
dataiter.next()
print('Sample input size: ', sample_x.size()) # batch_size, seq_length print('Sample input: \n',
sample_x) print('Sample input: \n', sample_y)
#MODEL class SentimentRNN(nn.Module): def
__init__(self,no_layers,vocab_size,hidden_dim,embedding_dim,drop_prob=0.5):
super(SentimentRNN,self).__init__() self.output_dim = output_dim self.hidden_dim = hidden_dim
self.no_layers = no_layers self.vocab_size = vocab_size # embedding and LSTM layers
self.embedding = nn.Embedding(vocab_size, embedding_dim) #lstm self.lstm =
nn.LSTM(input_size=embedding_dim,hidden_size=self.hidden_dim, num_layers=no_layers,
batch_first=True)
# dropout layer self.dropout = nn.Dropout(0.3) # linear and sigmoid layer self.fc =
nn.Linear(self.hidden_dim, output_dim) self.sig = nn.Sigmoid() def forward(self,x,hidden):
batch_size = x.size(0) # embeddings and lstm_out embeds = self.embedding(x) # shape: B x S x
Feature since batch = True #print(embeds.shape) #[50, 500, 1000] lstm_out, hidden =
self.lstm(embeds, hidden) lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim) # dropout
and fully connected layer out = self.dropout(lstm_out) out = self.fc(out) # sigmoid function
sig_out = self.sig(out) # reshape to be batch_size first sig_out = sig_out.view(batch_size, -1)
sig_out = sig_out[:, -1] # get last batch of labels # return last sigmoid output and hidden state
return sig_out, hidden def init_hidden(self, batch_size): ''' Initializes hidden state ''' # Create two
new tensors with sizes n_layers x batch_size x hidden_dim, # initialized to zero, for hidden state
and cell state of LSTM h0 = torch.zeros((self.no_layers,batch_size,self.hidden_dim)).to(device)
c0 = torch.zeros((self.no_layers,batch_size,self.hidden_dim)).to(device) hidden = (h0,c0) return
hidden
no_layers = 2 vocab_size = len(vocab) + 1 #extra 1 for padding embedding_dim = 64 output_dim
= 1 hidden_dim = 256

```
model = SentimentRNN(no_layers,vocab_size,hidden_dim,embedding_dim,drop_prob=0.5)
#moving to gpu model.to(device)
print(model)
#TRAINING # loss and optimization functions lr=0.001 criterion = nn.BCELoss() optimizer =
torch.optim.Adam(model.parameters(), lr=lr) # function to predict accuracy def acc(pred,label):
pred = torch.round(pred.squeeze()) return torch.sum(pred == label.squeeze()).item() clip = 5
epochs = 5 valid_loss_min = np.Inf # train for some number of epochs
epoch_tr_loss,epoch_vl_loss = [],[] epoch_tr_acc,epoch_vl_acc = [],[] for epoch in range(epochs):
train_losses = [] train_acc = 0.0 model.train() # initialize hidden state h =
model.init_hidden(batch_size) for inputs, labels in train_loader: inputs, labels =
inputs.to(device), labels.to(device) # Creating new variables for the hidden state, otherwise #
we'd backprop through the entire training history h = tuple([each.data for each in h])
model.zero_grad() output,h = model(inputs,h) # calculate the loss and perform backprop loss =
criterion(output.squeeze(), labels.float()) loss.backward() train_losses.append(loss.item()) #
calculating accuracy accuracy = acc(output,labels) train_acc += accuracy #`clip_grad_norm`
helps prevent the exploding gradient problem in RNNs / LSTMs.
nn.utils.clip_grad_norm_(model.parameters(), clip) optimizer.step() val_h =
model.init_hidden(batch_size)
val_losses = [] val_acc = 0.0 model.eval() for inputs, labels in valid_loader: val_h =
tuple([each.data for each in val_h])
inputs, labels = inputs.to(device), labels.to(device)
output, val_h = model(inputs, val_h) val_loss = criterion(output.squeeze(), labels.float())
val_losses.append(val_loss.item()) accuracy = acc(output,labels) val_acc += accuracy
epoch_train_loss = np.mean(train_losses) epoch_val_loss = np.mean(val_losses)
epoch_train_acc = train_acc/len(train_loader.dataset) epoch_val_acc =
val_acc/len(valid_loader.dataset) epoch_tr_loss.append(epoch_train_loss)
epoch_vl_loss.append(epoch_val_loss) epoch_tr_acc.append(epoch_train_acc)
epoch_vl_acc.append(epoch_val_acc) print(f'Epoch {epoch+1}') print(f'train_loss :
{epoch_train_loss} val_loss : {epoch_val_loss}') print(f'train_accuracy : {epoch_train_acc*100}
val_accuracy : {epoch_val_acc*100}') if epoch_val_loss <= valid_loss_min:
torch.save(model.state_dict(), '../working/state_dict.pt') print('Validation loss decreased ({:.6f} --
> {:.6f}).
Saving model ...'.format(valid_loss_min,epoch_val_loss)) valid_loss_min = epoch_val_loss
print(25*'==') fig = plt.figure(figsize = (20, 6)) plt.subplot(1, 2, 1) plt.plot(epoch_tr_acc,
label='Train Acc') plt.plot(epoch_vl_acc, label='Validation Acc') plt.title("Accuracy") plt.legend()
plt.grid() plt.subplot(1, 2, 2) plt.plot(epoch_tr_loss, label='Train loss') plt.plot(epoch_vl_loss,
label='Validation loss') plt.title("Loss") plt.legend() plt.grid()
plt.show()
plt.show()
#INFERANCE def predict_text(text): word_seq = np.array([vocab[preprocess_string(word)] for
word in text.split() if preprocess_string(word) in vocab.keys()]) word_seq =
np.expand_dims(word_seq,axis=0) pad = torch.from_numpy(padding_(word_seq,500)) inputs =
pad.to(device) batch_size = 1 h = model.init_hidden(batch_size) h = tuple([each.data for each in
h]) output, h = model(inputs, h) return(output.item())
index = 30 print(df['review'][index]) print('='*70) print(f'Actual sentiment is : {df["sentiment"]
[index]}') print('='*70) pro = predict_text(df['review'][index]) status = "positive" if pro > 0.5 else
```

```
"negative" pro = (1 - pro) if status == "negative" else pro print(f'Predicted sentiment is {status} with a probability of {pro}')
index = 30 print(df['review'][index]) print('='*70) print(f'Actual sentiment is : {df["sentiment"][index]}') print('='*70) pro = predict_text(df['review'][index]) status = "positive" if pro > 0.5 else "negative" pro = (1 - pro) if status == "negative" else pro print(f'Predicted sentiment is {status} with a probability of {pro}')
index = 32 print(df['review'][index]) print('='*70) print(f'Actual sentiment is : {df["sentiment"][index]}') print('='*70) pro = predict_text(df['review'][index]) status = "positive" if pro > 0.5 else "negative" pro = (1 - pro) if status == "negative" else pro print(f'predicted sentiment is {status} with a probability of {pro}')
index = 32 print(df['review'][index]) print('='*70) print(f'Actual sentiment is : {df["sentiment"][index]}') print('='*70) pro = predict_text(df['review'][index]) status = "positive" if pro > 0.5 else "negative" pro = (1 - pro) if status == "negative" else pro print(f'predicted sentiment is {status} with a probability of {pro}')
```