Natural Language Processing in Deep Learning
Code Demo

varshahe- 50600698
salonika-50611349

Introduction

Natural Language Processing (NLP) is a subfield of artificial intelligence that enables computers to understand and process human language. From chatbots to machine translation and sentiment analysis, NLP plays a crucial role in many modern applications. This article provides a structured and comprehensive guide to essential NLP techniques with Python, covering key concepts, libraries and implementations of deep learning capabilities in NLP.

Common NLP Libraries

Several Python libraries simplify NLP tasks, providing powerful tools for text analysis and machine learning. Here are some commonly used libraries:

- **NLTK (Natural Language Toolkit)**: A popular library for NLP that provides tools for text processing, tokenization, stemming, and more.
- **spaCy**: An efficient NLP library optimized for production use, offering robust tokenization, Named Entity Recognition (NER), and deep learning models.
- **Gensim**: Used for topic modeling and word embeddings making it ideal for semantic analysis.
- **Hugging Face Transformers**: Provides pre-trained deep learning models for advanced NLP tasks like text classification, translation and sentiment analysis.
- **TensorFlow/Keras and PyTorch**: Popular deep learning frameworks used for building and training NLP models.

Foundations of NLP: Preprocessing and Feature Engineering

Text/ Data Preprocessing

Tokenization

Tokenization is process of breaking text into smaller units such as words, phrases or symbols called "tokens." It helps convert raw text into a format that can be more easily analyzed and processed by machine learning algorithms.

```
from nltk.tokenize import word_tokenize, sent_tokenize
text = "Natural language processing is fascinating. It enables computers to understand
human language."
word_tokens = word_tokenize(text)
sentence_tokens = sent_tokenize(text)
print("Word Tokens:", word_tokens)
print("Sentence Tokens:", sentence_tokens)
```

Output:

```
Word Tokens: ['Natural', 'language', 'processing', 'is', 'fascinating', '.', 'It', 'enables', 'computers', 'to', 'understand', 'human', 'language', '.']
Sentence Tokens: ['Natural language processing is fascinating.', 'It enables computers to understand human language.']
```

Punctuation and removing characters

Punctuation marks and non-alphanumeric characters are often irrelevant for certain NLP tasks like sentiment analysis or topic modeling thereforethey must be removed to prevent unnecessary noise and to focus solely on meaningful words in the text. This helps to improve the performance of NLP models by reducing the complexity of input data and ensuring that the focus remains on the most important features of text.

```
import pandas as pd
import string
string.punctuation
clean_text = ''.join(char for char in text_lowercase if char not in
string.punctuation)
clean_text
```

```
import re
clean_text = re.sub('[^a-zA-Z0-9]', ' ', clean_text)
clean_text = re.sub(r'\s+', ' ', clean_text)
clean_text
```

```
'natural language processing is fascinating it enables computers to understand human language'
```

Removing stopwords

This is an important step that involves eliminating common words like "the", "is", "in", and "at" which do not carry significant meaning on their own. These words are frequently used in language but don't contribute much to the context of the text in many NLP tasks. By removing stopwords we reduce the dimensionality of text and allow model to focus on more meaningful and contextually relevant words. This step ultimately improves performance of model in tasks like text classification, sentiment analysis and information retrieval.

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

# nltk.download('punkt')
# nltk.download('stopwords') # download stopwords
words = word_tokenize(clean_text) #tokenize
stop_words = set(stopwords.words('english')) #load 'stopwords' in English lang
filteredwords = [word for word in words if word.lower() not in stop_words]
```

```
print(filteredwords)
```

```
['natural', 'language', 'processing', 'fascinating', 'enables', 'computers', 'understand', 'human', 'language']
```

Stemming

Stemming reduces words to their base or root form by removing prefixes or suffixes. The goal is to group different forms of a word and treat them as a single item thereby reducing vocabulary size and handling variations of words more efficiently. However, stemming may sometimes lead to "stem words" that are not actual words as it focuses purely on the root form without consideringcontext.

```
from nltk.stem import PorterStemmer, SnowballStemmer
porter = PorterStemmer()
porter.stem('fighting')
```

Lemmatizating

Unlike Stemming, lemmatization is more advanced and reduces words to their base or root(lemma) form while ensuring that the resulting word is valid dictionary word. Lemmatization considers context and part of speech to ensure the lemma is meaningful.

```
# nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()
lem.lemmatize('fighting')
```

```
lem.lemmatize('stripes', 'v')
```

```
lem.lemmatize('stripes', 'n')
```

Comparison between Stemming and Lemmatizing

```
## comparison between stemming and lemmatization
lem_text = " ".join(lem.lemmatize(word) for word in word_tokens)
stem_text = " ".join(porter.stem(word) for word in word_tokens)
print(f"text after lemmatization:- {lem_text}")
print(f"text after stemming:- {stem_text}")
```

```
text after lemmatization:- Natural language processing is fascinating . It enables computer to understand human language .
text after stemming:- natur languag process is fascin . it enabl comput to understand human languag .
```

Spell correction

```
# ! pip install pyspellchecker
from spellchecker import SpellChecker
```

```
sc = SpellChecker()
misspelledText = "I hav been rying to wok wih thi modelu" #I have been trying to work with this module

words = misspelledText.split()
correctedList = [sc.correction(word) for word in words]
correctedText = ' '.join(correctedList)

print("List of corrected words:",correctedList)
print("Original Text:", misspelledText)
print("Corrected Text:", correctedText)
```

```
List of corrected words: ['I', 'have', 'been', 'trying', 'to', 'wok', 'with', 'the', 'model']
Original Text: I hav been rying to wok wih thi modelu
Corrected Text: I have been trying to wok with the model
```

Complete preprocessing pipeline

```
import re
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
import nltk

# nltk.download('punkt')
# nltk.download('stopwords')
# nltk.download('wordnet')

sample_text = "Barack Obama, the 44th President of the United States, gave a speech in Berlin on July 4, 2009, at the Brandenburg Gate. He spoke about the relationship between the United States and Europe, emphasizing the importance of international cooperation. The event was covered by major news outlets, including BBC, CNN, and The New York Times!"

# Convert to lowercase
sample_text = sample_text.lower()

# Remove punctuation
sample_text = re.sub(f"[{string.punctuation}]", "", sample_text)

# Tokenize words
```

```
tokens = word_tokenize(sample_text)

# Remove stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]

# Stemming & Lemmatization
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

stemmed_words = [stemmer.stem(word) for word in filtered_tokens]
lemmatized_words = [lemmatizer.lemmatize(word) for word in filtered_tokens]

print("Stemmed Words:", stemmed_words)
print("Lemmatized Words:", lemmatized_words)
```

Feature Extraction

Part-of-Speech Tagging(POS tagging)

POS tagging helps in understanding the structure of a sentence and is essential for tasks like named entity recognition (NER), parsing and machine translation, as it provides crucial context about how words interact with each other. In this process each word in a sentence is assigned a specific grammatical category such as a noun, verb, adjective, adverb, etc.

```
#pre-requisite
#nltk.download('averaged_perceptron_tagger')
import nltk
from nltk import pos_tag
pos_tag(lemmatized_words)
# pos_tag(['countries'])
```

```
[('barack', 'NN'),
 ('obama', 'VBZ'),
 ('44th', 'CD'),
 ('president', 'NN'),
 ('united', 'JJ'),
 ('state', 'NN'),
 ('gave', 'VBD'),
 ('speech', 'NN'),
 ('berlin', 'NN'),
 ('july', 'RB'),
 ('4', 'CD'),
 ('2009', 'CD'),
```

Named Entity Recognition (NER)

This process involves identifying and classifying named entities in text into predefined categories such as people, organizations, locations, dates, and more.NER helps in extracting valuable information from unstructured text, enabling tasks like information retrieval, question answering, and summarization by organizing and structuring key entities from the data.

```
#pre-requisites
# ! pip install spacy
#run the following script on terminal to download English lang model from spacy
#python -m spacy download en_core_web_sm

import spacy
# Load pre-trained English language model 'en_core_web_sm' from spaCy
nlp = spacy.load('en_core_web_sm')

extract_ner = nlp(sample_text)
for ent in extract_ner.ents:
   print(ent.text, ent.label_)
```

```
44th ORDINAL
the united states GPE
berlin GPE
july 4 2009 DATE
the brandenburg gate FAC
the united states GPE
europe LOC
bbc cnn ORG
the new york times ORG
```

Bag of Words

Common used technique in NLP for representing text data wherein text is treated as an unordered collection of words, ignoring grammar and word order but keeping track of the frequency of each word.

```
from sklearn.feature_extraction.text import CountVectorizer

# bag of words implementation expects an input as a list insead of string. So we'll
use our example text in list form
textForBow = [sample_text] # convert to list
vectorizer = CountVectorizer()
textVectors = vectorizer.fit_transform(textForBow)
bagOfWords = textVectors.toarray() # array- bag of words

vocab= vectorizer.get_feature_names_out() ## fetch the vocabulary

print("Vocabulary:", vocab)
print("Bag of Words Model:")
print(bagOfWords)
```

```
Vocabulary: ['2009' '44th' 'about' 'and' 'at' 'barack' 'bbc' 'berlin' 'between'
 'brandenburg' 'by' 'cnn' 'cooperation' 'covered' 'emphasizing' 'europe'
 'event' 'gate' 'gave' 'he' 'importance' 'in' 'including' 'international'
 'july' 'major' 'new' 'news' 'obama' 'of' 'on' 'outlets' 'president'
 'relationship' 'speech' 'spoke' 'states' 'the' 'times' 'united' 'was'
 'york']
Bag of Words Model:
[[1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1
  2 8 1 2 1 1]]
```

Word Embeddings using Gensim

Word Embeddings using Gensim is a technique in NLP that represents words as dense vectors of real numbers, where words with similar meanings have similar vector representations.Word2Vec uses a neural network to learn word representations based on their context in large text corpora. It uses either the Continuous Bag of Words (CBOW) or Skip-gram approach to predict words within a context window, with the idea that words used in similar contexts will have similar embeddings.Words with similar meanings will have similar vector representations. algorithms like **Word2Vec**, **FastText**, and **GloVe**

```
import gensim.downloader as api
import numpy as np
from gensim.test.utils import common_texts
from gensim.models import Word2Vec

# Load the pre-trained Word2Vec model
model = api.load("word2vec-google-news-300")

# Function to get word embedding
def getWordEmbeddings(word, model):
   try:
       return model[word]
   except KeyError:
       return None

# calculate cosine similarity between words
from numpy.linalg import norm

def cosine_similarity(vec1, vec2):
   return np.dot(vec1, vec2) / (norm(vec1) * norm(vec2))

word1 = "Obama"
word2 = "United"

embedding1 = getWordEmbeddings(word1, model)
embedding2 = getWordEmbeddings(word2, model)

if embedding1 is not None and embedding2 is not None:
   similarity = cosine_similarity(embedding1, embedding2)
   print(f"Cosine similarity between '{word1}' and '{word2}': {similarity}")
else:
   print("Word(s) not found in pre-trained model.")
```

```
Cosine similarity between 'Obama' and 'United': 0.1680458039045334
```

Natural Language Processing in Deep Learning: Hands on Experience

Through three real-life examples of how deep learning models process, create, and categorize text, this article presents an in-depth overview of natural language processing in deep learning:

The three real-life examples are:

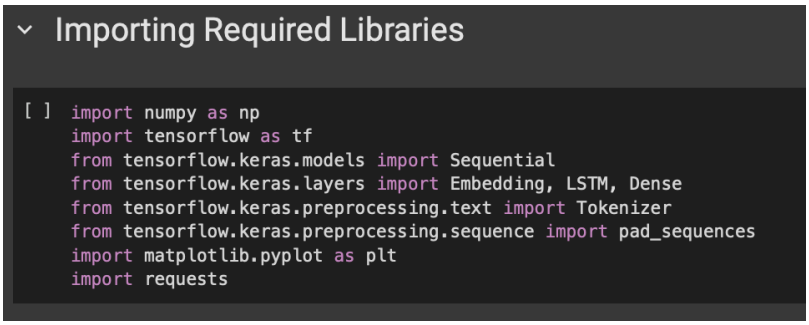
- One straightforward method to conceptualize how LSTMs handle sequential text is Next-Word Prediction with a Small Text Corpus.
- Next-Word Prediction with a Big Dataset: To optimize language learning, scale the model with a real-world dataset.
- Sentiment Analysis of IMDB Movie Reviews: Learn the use of LSTM networks in deep learning for text classification as positive or negative sentiment.

Next-Word Prediction using a Small Text Corpus

Before solving on a larger sample, the text generation and prediction methods are illustrated using a small sample of data. This way, it is kept easy to get an initial idea of how the deep learning algorithm processes and generates text before proceeding to larger data sets.

Importing Required Libraries

Essential libraries for data preprocessing, model training, and evaluation are imported.



Library Functions used in the code:

NumPy: Handles numerical operations and array manipulations.

TensorFlow/Keras: Core deep learning framework used for training models.

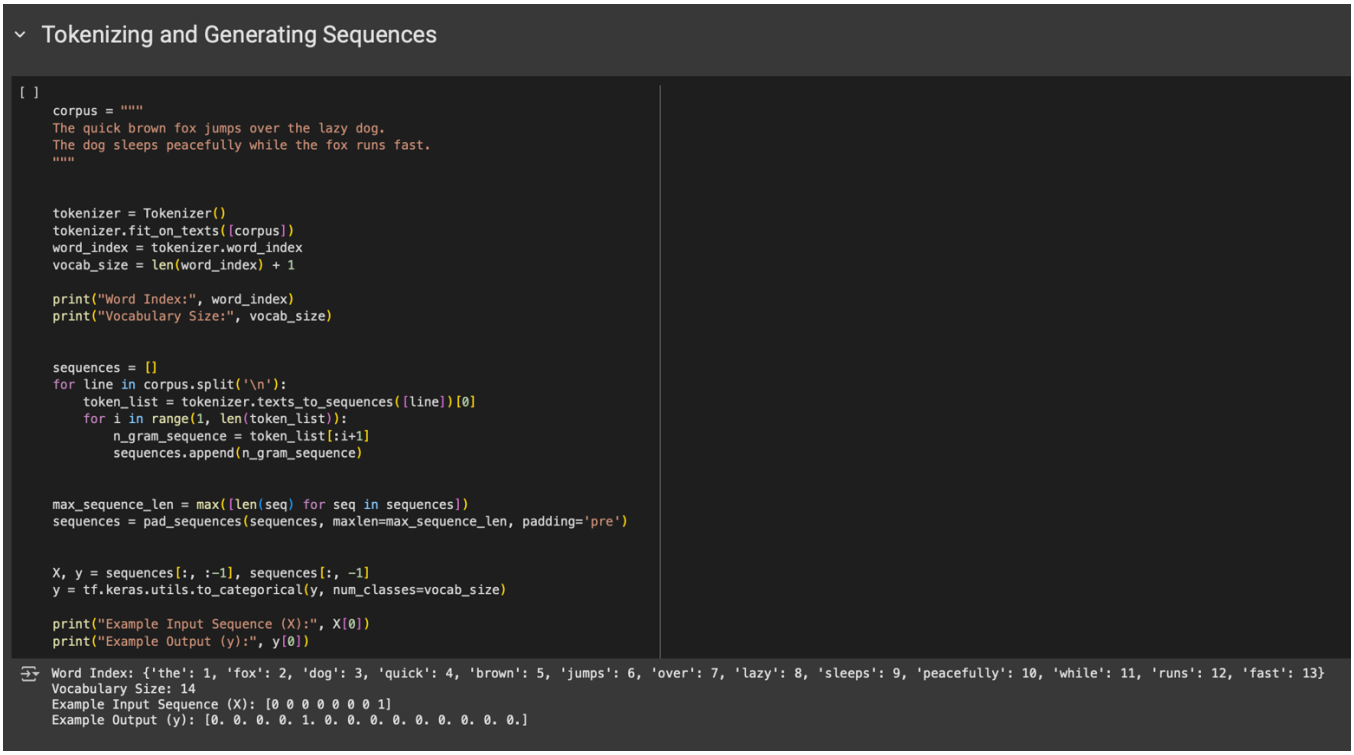
Tokenizer: Converts text into numerical tokens for model processing.

LSTM Layer: Captures long-term dependencies in text sequences.

Matplotlib: Visualizes training performance.

Data Preparation

For pre-training the model, a small corpus of text is defined. The text is tokenized and transformed into a sequence of numbers in the subsequent step. Padding is appended to the sequences formed with an uniform sequence length.



Processing Steps Implemented in the image:

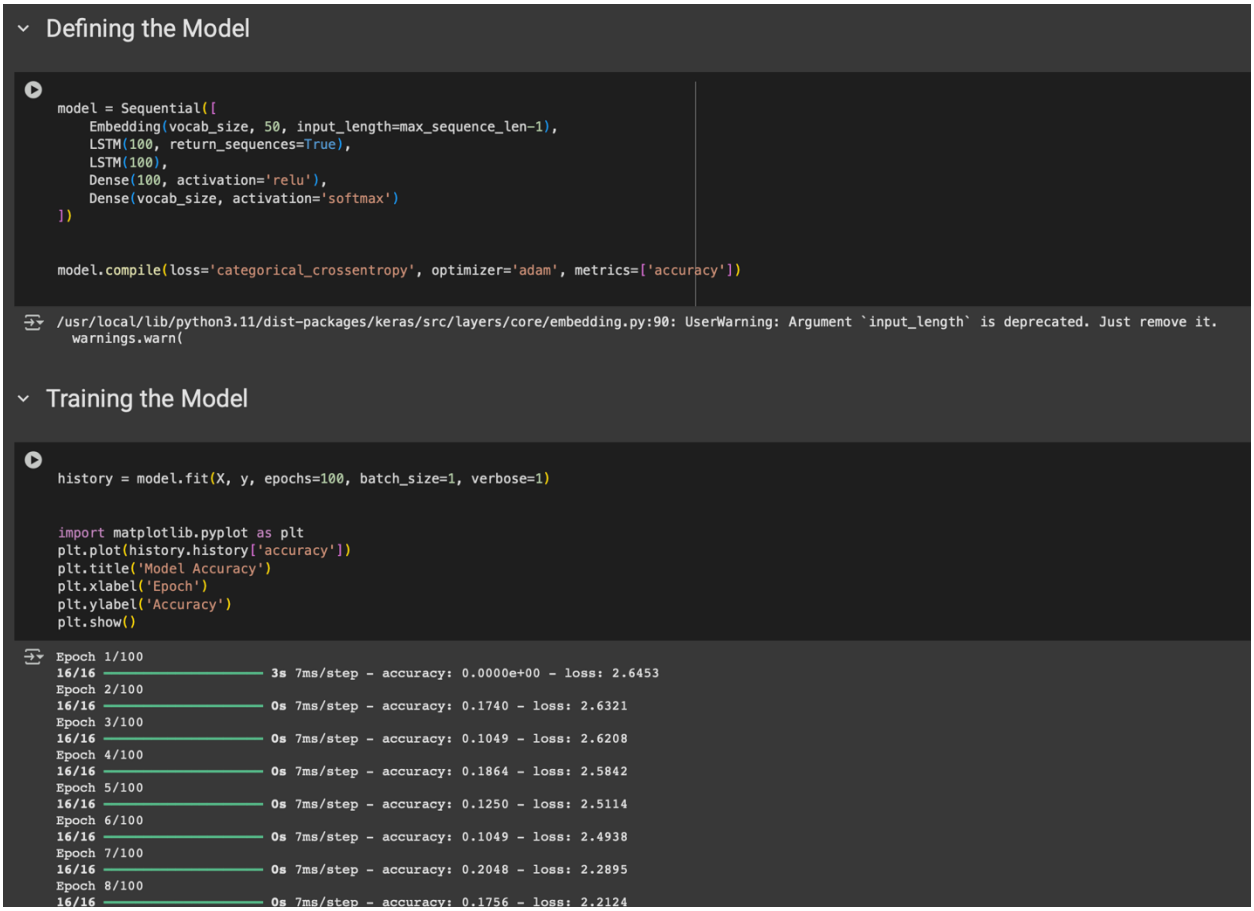
Tokenization assigns each word a unique numerical value.

Sequence Generation enables the model to learn relationships between words.

Padding standardizes sequence lengths for input into the LSTM.

Splitting Data into X (input) and y (output) prepares the model for training.

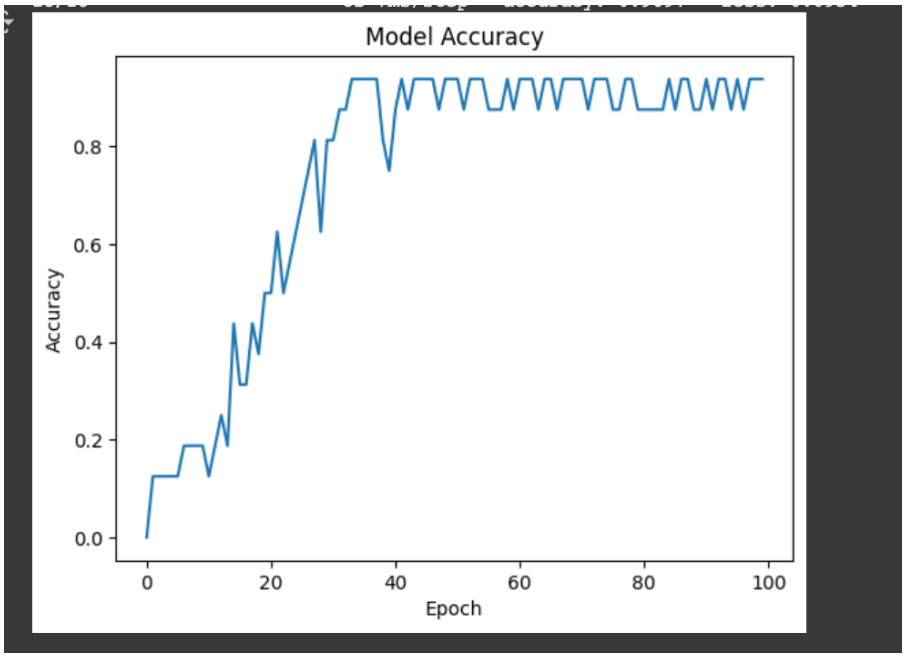
Model Definition and Training



Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) specially designed to learn sequential dependencies and, therefore, optimally adapted to text prediction. The model is trained using categorical cross-entropy loss and Adam optimizer.

LSTM architecture incorporates an embedding layer to carry out word-to-dense vectors transformation, and two LSTM layers to learn sequential patterns. The model incorporates a dense layer of ReLU activation to learn challenging patterns, and the final softmax layer to output the next word. Categorical cross-entropy loss is utilized to compile the model and Adam optimizer.

Plotting the Accuracy of the Model



The chart shows the model precision over 100 epochs, showing how the next-word prediction model learns throughout training. In the beginning, precision is low but gradually increases as the model identifies patterns within the data set. At epoch 40, accuracy reaches almost 90%, showing effective learning. But minor fluctuations in later epochs show some inconsistency in training, probably caused by the data size or model complexity. This plot confirms that the LSTM model learns word dependencies well and is therefore perfectly suited to predict the next word in a sequence.

Making Predictions for the next word

Function to predict the next word

```
[ ]
    def predict_next_word(model, tokenizer, text, max_sequence_len):
        token_list = tokenizer.texts_to_sequences([text])[0]
        token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')
        predicted = model.predict(token_list, verbose=0)
        predicted_word_index = np.argmax(predicted, axis=1)[0]
        for word, index in tokenizer.word_index.items():
            if index == predicted_word_index:
                return word
        return None


    seed_text = "the quick"
    next_word = predict_next_word(model, tokenizer, seed_text, max_sequence_len)
    print(f"Seed text: '{seed_text}' → Predicted next word: '{next_word}'")
```

```
Seed text: 'the quick' → Predicted next word: 'brown'
```

The function takes in a seed text, tokenizes it, pads the sequence to the model's input length, and predicts the next word based on the trained model from the previous step. The argmax function in this step is used

to obtain the index of the most probable next word, which is translated back to its corresponding word form in the tokenizer's vocabulary. Here, the seed text "the quick" is passed through, and the model predicts the next word to be "brown", showing its ability to learn helpful word associations.

Making Predictions for the next sequence

Function to generate a sequence of words

```
def generate_sequence(model, tokenizer, max_sequence_len, seed_text, num_words):
    for _ in range(num_words):
        next_word = predict_next_word(model, tokenizer, seed_text, max_sequence_len)
        if next_word is None:
            break
        seed_text += " " + next_word
    return seed_text

seed_text = "the quick"
generated_text = generate_sequence(model, tokenizer, max_sequence_len, seed_text, 3)
print(f"Seed text: '{seed_text}' → Generated text: '{generated_text}'")
```

```
Seed text: 'the quick' → Generated text: 'the quick brown fox jumps'
```

The technique takes a seed string and continues to predict the next word using the predict_next_word method. It appends each predicted word to the seed string, building an ever longer phrase. The loop continues for a specified number of words or when no next word is validly predicted. Here, from the seed string "the quick", the model is able to produce "the quick brown fox jumps" successfully, demonstrating the ability to construct meaningful multi-word strings from patterns learned.

The use of next-word prediction with a short text provides us with a sense of how sequential text is processed by deep learning models. This is then built upon in the next section by extending the concept to bigger data, increasing performance and accuracy of prediction.

Then, this concept is applied on a large dataset for improved results.

Next-Word Prediction Using a Large Dataset

Even though the small corpus allowed us to understand word sequence learning, it was poor in vocabulary and context richness. In order to get better performance, the model must be trained with a large corpus so that it can learn richer patterns in language.

The process adopted in carrying out the subsequent word prediction with a large dataset is described below.

Fetching and Preprocessing the Dataset and Tokenizing and Creating Sequences

To obtain a meaningful dataset, **Alice in Wonderland** from **Project Gutenberg** is used. The usage of larger dataset increases the vocabulary and lets the model learn from more diverse sentence structure. The project Gutenberg books contain metadata which is removed in the step trimming the header and footer.

Fetching a larger dataset (Alice in Wonderland) for Predicting the Next Word

```
[ ] url = "https://www.gutenberg.org/files/11/11-0.txt"
    response = requests.get(url)
    corpus = response.text
```

Triming the header/footer

```
    start_marker = "*** START OF THE PROJECT GUTENBERG EBOOK"
    end_marker = "*** END OF THE PROJECT GUTENBERG EBOOK"
    start_idx = corpus.find(start_marker) + len(start_marker)
    end_idx = corpus.find(end_marker)
    corpus = corpus[start_idx:end_idx].strip().lower()
```

Tokenizing the text

```
[ ] tokenizer = Tokenizer()
    tokenizer.fit_on_texts([corpus])
    word_index = tokenizer.word_index
    vocab_size = len(word_index) + 1
    print("Vocabulary Size:", vocab_size)
```

```
Vocabulary Size: 3516
```

The text is then tokenized where words are translated into numeric entities. The vocabulary size, 3,516 is determined which will be representing the number of distinct words present in the data set. The step is important to train an LSTM model from a vast corpus so it could acquire the nuances in the language compared to a data set with smaller size.

Creating sequences

```
[ ]
    sequences = []
    for line in corpus.split('\n'):
        token_list = tokenizer.texts_to_sequences([line])[0]
        for i in range(1, len(token_list)):
            n_gram_sequence = token_list[:i+1]
            sequences.append(n_gram_sequence)
```

Pading the sequences

```
[ ] max_sequence_len = max([len(seq) for seq in sequences])
    sequences = pad_sequences(sequences, maxlen=max_sequence_len, padding='pre')
    print("Max Sequence Length:", max_sequence_len)
```

```
Max Sequence Length: 18
```

Spliting into input and output

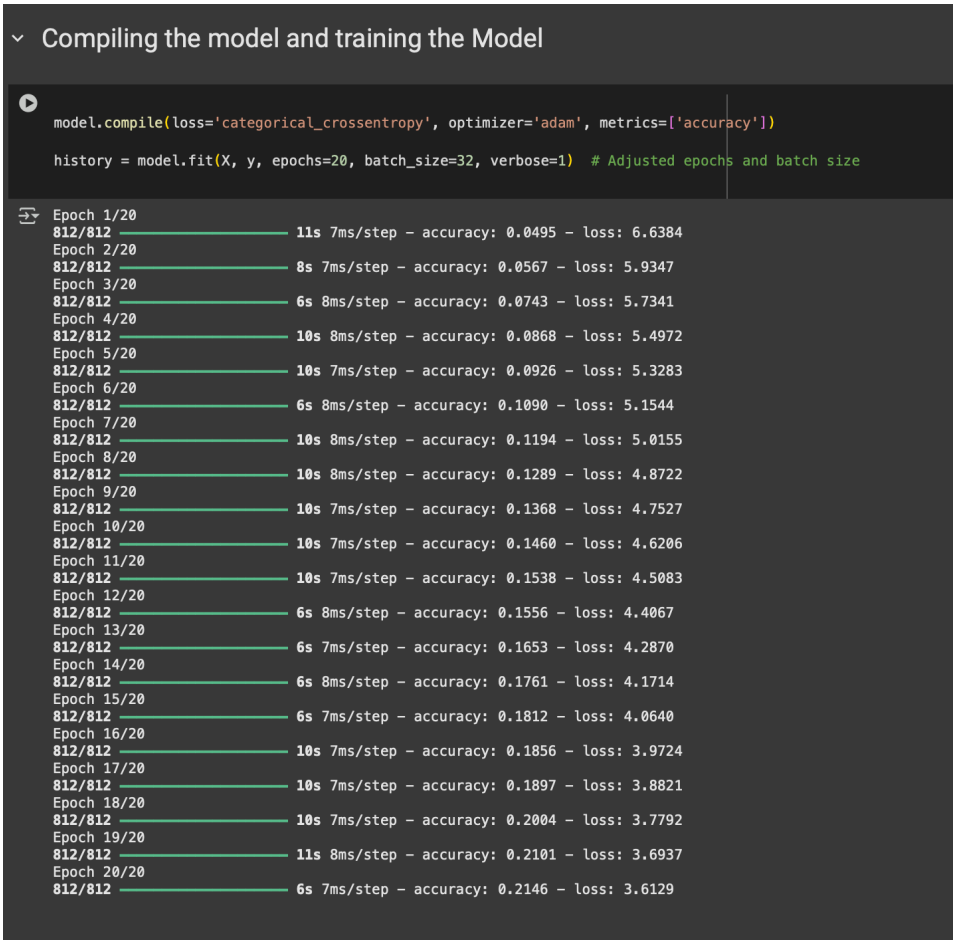
```
[ ] X, y = sequences[:, :-1], sequences[:, -1]
    y = tf.keras.utils.to_categorical(y, num_classes=vocab_size)
```

Building the model

```
    model = Sequential([
        Embedding(vocab_size, 50, input_length=max_sequence_len-1),
        LSTM(100, return_sequences=True),
        LSTM(100),
        Dense(100, activation='relu'),
        Dense(vocab_size, activation='softmax')
    ])
```

First, sequences are created by converting the text into numerical tokens and constructing n-gram sequences. Since sequences are of varying length, padding is done to make them of equal length up to a maximum of 18 words. The dataset is divided into input (X) and output (y), with X being the previous words and y as the next word to be predicted.

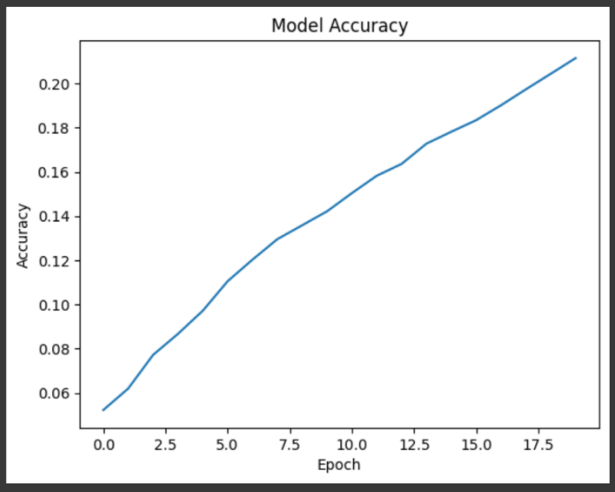
Finally, there is the description of an LSTM-based network containing an embedding layer, two layers of LSTM, and dense prediction layers. This is a model's configuration to learn the long-term dependency of the text as well as boost the prediction of the next word in a sequence.



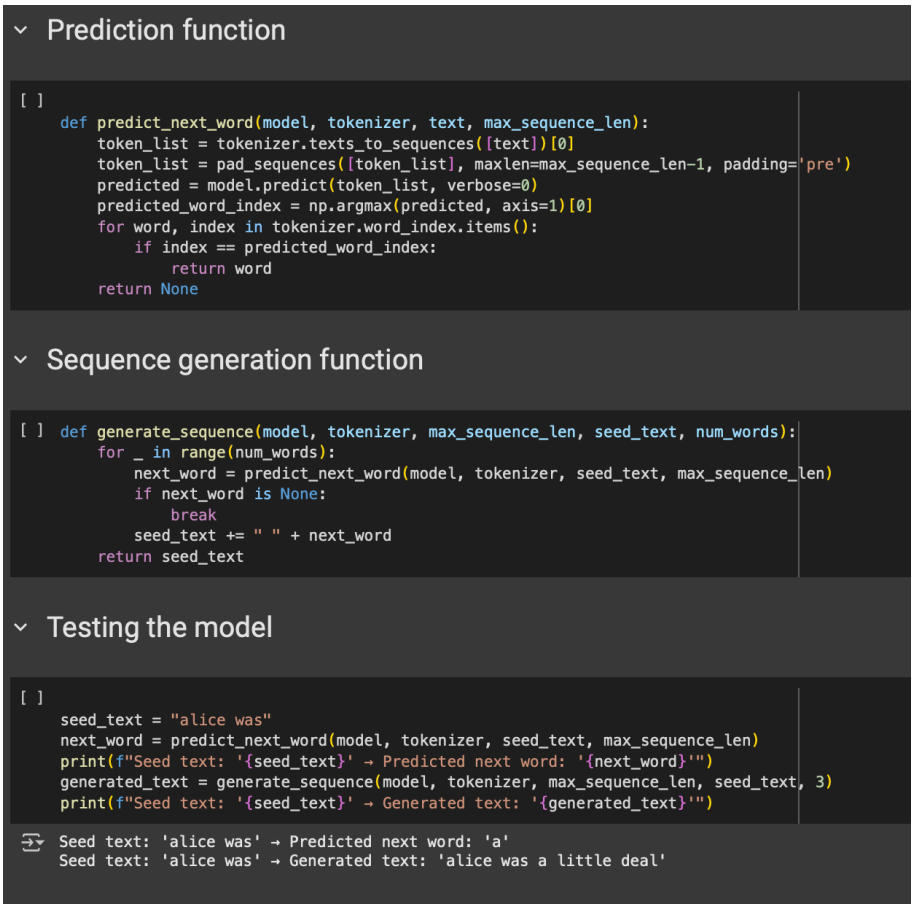
The model is trained with categorical cross-entropy loss function, Adam optimizer, and accuracy as the metric. The model is trained for 20 epochs with batch size of 32 with rising accuracy and declining loss. Initially, the model possesses low accuracy of 0.0495, but training advances, accuracy increases consistently to 0.2146 after the 20th epoch. That implies that the model is learning patterns from the data set and increasingly better able to predict.

Plot accuracy

```
plt.plot(history.history['accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```



The figure above shows the training accuracy curve for the next-word prediction model. The x-axis represents the number of epochs, while the y-axis displays the precision. The graph's steady increase in accuracy over time shows that the model is effectively learning from the dataset.



The predict_next_word function uses the trained LSTM model to make predictions for the next word by likelihood after tokenization of the seed text and padding of the sequence. The idea is expanded in the generate_sequence function, where the model iteratively predicts and appends words to produce an extended sequence.

In the test phase, the model is given the input "Alice was," forecasts the next word to be "a," and produces the brief, coherent sentence "Alice was a little deal." This demonstrates how the LSTM model generates informed text predictions by identifying patterns in language in the sample.

The next word and the next sequences are generated for the multiple test cases.

```
Testing Next Word Prediction:
Seed text: 'alice was' → Predicted next word: 'a'
Seed text: 'alice was' → Generated text: 'alice was a little deal'

Seed text: 'the white rabbit' → Predicted next word: 'blew'
Seed text: 'the white rabbit' → Generated text: 'the white rabbit blew little little'

Seed text: 'down the' → Predicted next word: 'rabbit'
 up'

Seed text: 'she fell' → Predicted next word: 'up'
Seed text: 'she fell' → Generated text: 'she fell up to say'

Seed text: 'what a' → Predicted next word: 'little'
Seed text: 'what a' → Generated text: 'what a little deal to'

Seed text: 'the mad hatter' → Predicted next word: 'deeply'
Seed text: 'the mad hatter' → Generated text: 'the mad hatter deeply and butter'

Seed text: 'into the' → Predicted next word: 'rabbit'
 up'

Seed text: 'very curious' → Predicted next word: 'to'
Seed text: 'very curious' → Generated text: 'very curious to speak on'

Seed text: 'the queen of' → Predicted next word: 'the'
Seed text: 'the queen of' → Generated text: 'the queen of the other side'

Seed text: 'it was a' → Predicted next word: 'little'
Seed text: 'it was a' → Generated text: 'it was a little deal to'

Seed text: 'suddenly she' → Predicted next word: 'was'
Seed text: 'suddenly she' → Generated text: 'suddenly she was a little'

Seed text: 'the little door' → Predicted next word: 'cheered'
Seed text: 'the little door' → Generated text: 'the little door cheered and went'

Seed text: 'how very' → Predicted next word: 'business'
Seed text: 'how very' → Generated text: 'how very business ” the'

Seed text: 'through the' → Predicted next word: 'rabbit'
Seed text: 'through the' → Generated text: 'through the rabbit was the'

Seed text: 'in a moment' → Predicted next word: 'to'
Seed text: 'in a moment' → Generated text: 'in a moment to the other'
```

Transferring from a limited dataset to an increased dataset produced drastic improvements in the quality of predictions, ability to learn, and model correctness. Training on the larger Alice in Wonderland dataset resulted in more contextualization of the model and also increased improvement in coherent sequence generation. In natural language processing (NLP), larger datasets play a crucial role to drive a model to acquiring the capability to learn the patterns of the language and provide correct word predictions.

Sentiment analysis utilizing IMDB movie reviews

Sentiment analysis utilizing IMDB movie reviews is then the main focus, with the goal of categorizing reviews as either positive or negative.

Required libraries are imported for the purpose.

Sentiment Analysis Using the IMDB Dataset

```
[ ] import numpy as np
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Bidirectional
    from tensorflow.keras.preprocessing.sequence import pad_sequences
    from tensorflow.keras.datasets import imdb
    from tensorflow.keras.callbacks import EarlyStopping
    import matplotlib.pyplot as plt
    import re
```

Loading the IMDB dataset

```
[ ]
    max_words = 10000
    max_len = 200


    (X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=max_words)
    X_train = pad_sequences(X_train, maxlen=max_len, padding='pre')
    X_test = pad_sequences(X_test, maxlen=max_len, padding='pre')
```

Building the model

```
[ ]
    model = Sequential([
        Embedding(max_words, 100, input_length=max_len),
        Bidirectional(LSTM(128)),
        Dropout(0.3),
        Dense(64, activation='relu'),
        Dropout(0.3),
        Dense(1, activation='sigmoid')
    ])
```

The data is preprocessed with vocabulary size limiting to 10,000 words and sequence padding for maintaining constant input length. Bidirectional LSTM model, augmented with dropout layers to avoid overfitting and sigmoid activation function to make binary classification possible, is used to increase context consciousness. The design allows for efficient identification of sentiment patterns in textual content.

Compiling the model with early stopping

```
[ ]
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
```

Training the model

```
[ ] history = model.fit(X_train, y_train, epochs=15, batch_size=32, validation_split=0.2,
                        callbacks=[early_stopping], verbose=1)
```

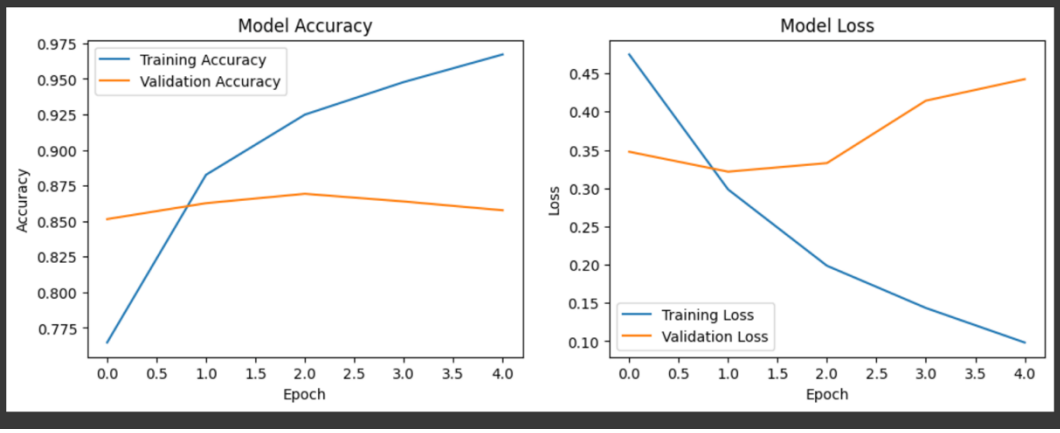
```
Epoch 1/15
625/625 ━━━━━━━━━━━━━━━━━━━━ 16s 19ms/step - accuracy: 0.6716 - loss: 0.5652 - val_accuracy: 0.8514 - val_loss: 0.3476
Epoch 2/15
625/625 ━━━━━━━━━━━━━━━━━━━━ 18s 18ms/step - accuracy: 0.8869 - loss: 0.2881 - val_accuracy: 0.8626 - val_loss: 0.3216
Epoch 3/15
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 18ms/step - accuracy: 0.9263 - loss: 0.1967 - val_accuracy: 0.8692 - val_loss: 0.3328
Epoch 4/15
625/625 ━━━━━━━━━━━━━━━━━━━━ 22s 20ms/step - accuracy: 0.9496 - loss: 0.1400 - val_accuracy: 0.8638 - val_loss: 0.4145
Epoch 5/15
625/625 ━━━━━━━━━━━━━━━━━━━━ 20s 18ms/step - accuracy: 0.9701 - loss: 0.0903 - val_accuracy: 0.8576 - val_loss: 0.4426
```

The model is built with an Adam optimizer, cross-entropy binary loss, and accuracy as a metric. Overfitting avoidance and efficient training are achieved using early pausing. The optimal weights are reintroduced after observation of validation loss for three epochs. The model is trained over 15 epochs of batch size 32 and 20% of the train set is utilized for validation. Training accuracy never goes down and validation accuracy remains the same, indicating an appropriate-generalized model for sentiment classification.

Ploting the accuracy and loss

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Training performance of the model is revealed through accuracy and loss curves. The model has trained appropriately but appears to be drifting towards overfitting as reflected by training accuracy increasing at every moment when validation accuracy stabilizes. As evidenced in the right graph, while validation loss is on the rise, potentially signifying overfitting, the training loss reduces, reflecting very good learning. These trends suggest that in a bid to preserve model generalization, regularization methods or early stopping is required.

Evaluating on test set

```
[ ]
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_acc:.4f}")
```

```
Test Accuracy: 0.8620
```

Testing on a few IMDB samples for comparison

```
print("\nTesting on IMDB Test Samples:")
for i in range(3):
    pred = model.predict(X_test[i:i+1], verbose=0)[0][0]
    sentiment = "Positive" if pred >= 0.5 else "Negative"
    print(f"Sample {i+1}: True Label: {y_test[i]}, Predicted: {sentiment} (Score: {pred:.4f})")


def preprocess_text(text, word_index, max_words, max_len):

    word_index = {word: idx + 3 for word, idx in word_index.items()}
    word_index["<PAD>"] = 0
    word_index["<START>"] = 1
    word_index["<UNK>"] = 2

    reverse_word_index = {idx: word for word, idx in word_index.items()}


    text = re.sub(r'[^\w\s]', '', text.lower())
    tokens = text.split()
    encoded = [min(word_index.get(word, 2), max_words - 1) for word in tokens]
    padded = pad_sequences([encoded], maxlen=max_len, padding='pre')


    decoded = [reverse_word_index.get(idx, "<UNK>") for idx in padded[0][-10:]]
    print(f"Original: '{text}' → Encoded (last 10): {padded[0][-10:]} → Decoded: {decoded}")
    return padded
```

```
Testing on IMDB Test Samples:
Sample 1: True Label: 0, Predicted: Negative (Score: 0.2464)
Sample 2: True Label: 1, Predicted: Positive (Score: 0.9922)
Sample 3: True Label: 1, Predicted: Positive (Score: 0.8596)
```

According to the evaluation results, the model indeed performed accurate sentiment classification with 86.2% accuracy on the IMDB test set. The model's effectiveness is also attested by the sample predictions, which accurately tag reviews as positive or negative depending on the probability score. The difficulty introduced by subtle expression of sentiment in text is seen in the finding that one sample was spuriously detected.

Prediction function

```
[ ]
    def predict_sentiment(model, text, word_index, max_words, max_len):
        processed_text = preprocess_text(text, word_index, max_words, max_len)
        prediction = model.predict(processed_text, verbose=0)[0][0]
        sentiment = "Positive" if prediction >= 0.5 else "Negative"
        return sentiment, prediction
```

Testing on custom reviews

```
    word_index = imdb.get_word_index()
    custom_reviews = [
        "This movie was absolutely fantastic and thrilling",
        "Terrible acting and a boring plot",
        "I loved every minute of this film",
        "The worst movie I have ever seen",

        "A breathtaking adventure that kept me hooked",
        "Poorly written and utterly forgettable",
        "The cinematography was gorgeous and the story touching",
        "A total disaster with no redeeming qualities",
        "Such a delightful experience from beginning to end",
        "Confusing plot and terrible performances"
    ]
    print("\nCustom Predictions:")
    for review in custom_reviews:
        sentiment, score = predict_sentiment(model, review, word_index, max_words, max_len)
        print(f"Review: '{review}'")
        print(f"Predicted Sentiment: {sentiment} (Score: {score:.4f})\n")
```

The sentiment prediction function, which converts the incoming text, structures it according to the learned model, and determines if the sentiment is positive or negative, can be seen in the figure above. The function tokenizes and cleanses the input before feeding it into the LSTM model. If the required score is achieved, the sentiment is marked as positive.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1641221/1641221 ━━━━━━━━━━━━━━━━━━━━ 1s 1us/step

Custom Predictions:
Original: 'this movie was absolutely fantastic and thrilling' → Encoded (last 10): [  0   0   0  14  20  16 427 777   5 3017] → Decoded: ['<PAD>', '<PAD>', '<PAD>', 'this', 'movie', 'was', 'absolutely', 'fan
Review: 'This movie was absolutely fantastic and thrilling'
Predicted Sentiment: Positive (Score: 0.8113)

Original: 'terrible acting and a boring plot' → Encoded (last 10): [  0   0   0   0 394 116   5   6 357 114] → Decoded: ['<PAD>', '<PAD>', '<PAD>', '<PAD>', 'terrible', 'acting', 'and', 'a', 'boring', 'plot']
Review: 'Terrible acting and a boring plot'
Predicted Sentiment: Negative (Score: 0.1145)

Original: 'i loved every minute of this film' → Encoded (last 10): [  0   0   0  13 447 175 786   7  14  22] → Decoded: ['<PAD>', '<PAD>', '<PAD>', 'i', 'loved', 'every', 'minute', 'of', 'this', 'film']
Review: 'I loved every minute of this film'
Predicted Sentiment: Positive (Score: 0.8744)

Original: 'the worst movie i have ever seen' → Encoded (last 10): [  0   0   0   4 249  20  13  28 126 110] → Decoded: ['<PAD>', '<PAD>', '<PAD>', 'the', 'worst', 'movie', 'i', 'have', 'ever', 'seen']
Review: 'The worst movie I have ever seen'
Predicted Sentiment: Negative (Score: 0.1443)

Original: 'a breathtaking adventure that kept me hooked' → Encoded (last 10): [   0   0   0   6 2877 1154  15  828   72 3305] → Decoded: ['<PAD>', '<PAD>', '<PAD>', 'a', 'breathtaking', 'adventure', 'that', 'kept
Review: 'A breathtaking adventure that kept me hooked'
Predicted Sentiment: Positive (Score: 0.9868)

Original: 'poorly written and utterly forgettable' → Encoded (last 10): [   0   0   0   0   0 862 398   5 1254 2441] → Decoded: ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', 'poorly', 'written', 'and', 'utterl
Review: 'Poorly written and utterly forgettable'
Predicted Sentiment: Negative (Score: 0.0500)

Original: 'the cinematography was gorgeous and the story touching' → Encoded (last 10): [   0   0   4 627  16 1490   5   4  65 1301] → Decoded: ['<PAD>', '<PAD>', 'the', 'cinematography', 'was', 'gorgeous', 'a
Review: 'The cinematography was gorgeous and the story touching'
Predicted Sentiment: Positive (Score: 0.9641)

Original: 'a total disaster with no redeeming qualities' → Encoded (last 10): [   0   0   0   6 964 1690  19  57 1653 2432] → Decoded: ['<PAD>', '<PAD>', '<PAD>', 'a', 'total', 'disaster', 'with', 'no', 'redeem
Review: 'A total disaster with no redeeming qualities'
Predicted Sentiment: Negative (Score: 0.1209)

Original: 'such a delightful experience from beginning to end' → Encoded (last 10): [   0   0 141   6 1917 585  39 454   8 130] → Decoded: ['<PAD>', '<PAD>', 'such', 'a', 'delightful', 'experience', 'from', '
Review: 'Such a delightful experience from beginning to end'
Predicted Sentiment: Positive (Score: 0.9844)

Original: 'confusing plot and terrible performances' → Encoded (last 10): [   0   0   0   0   0 1499 114   5 394 354] → Decoded: ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', 'confusing', 'plot', 'and', 'terr
Review: 'Confusing plot and terrible performances'
Predicted Sentiment: Negative (Score: 0.3153)
```

This shows the testing stage of the custom movie reviews, in which the sentiment analysis model that has been trained tokenizes, processes, and analyzes a list of sample reviews. Each review is given a confidence score and an emotion label. positive reviews are given high scores that are almost equal to 1,

while negative reviews are given low scores. The findings show that the model can make use of textual cues to distinguish between negative and positive movie review.

Transformers and Pre-trained models

Unlike earlier architectures like RNNs or LSTMs, which processed sequences word-by-word, the Transformer model uses self-attention mechanisms to consider the entire input sequence at once. This allows for more efficient parallelization and improved performance on tasks involving long-range dependencies.

```
#pip install transformers
from transformers import pipeline

# Load sentiment analysis pipeline
classifier = pipeline('sentiment-
analysis',model='distilbert/distilbert-base-uncased-finetuned-sst-2-
english')
# Sample data
texts = ["I love this product", "This is terrible", "I am so happy",
"I hate this"]

# Get predictions
for text in texts:
   result = classifier(text)
   print(f"Text: {text} => Sentiment: {result[0]['label']}")
```

```
Device set to use mps:0
Text: I love this product => Sentiment: POSITIVE
Text: This is terrible => Sentiment: NEGATIVE
Text: I am so happy => Sentiment: POSITIVE
Text: I hate this => Sentiment: NEGATIVE
```

T5 model(Text-to-text transfer transformer)

It is a specific implementation of the Transformer architecture designed to handle a variety of NLP tasks using a unified text-to-text framework. T5 can be applied to a wide range of NLP tasks like translation, summarization, question answering and sentiment analysis.

```
from transformers import T5ForConditionalGeneration, T5Tokenizer
# Load tokenizer & pre-trained T5 model
model_name = "t5-small"
model = T5ForConditionalGeneration.from_pretrained(model_name)
tokenizer = T5Tokenizer.from_pretrained(model_name)

# Sample text
text = "Translate English to Spanish: I love this product. I can't
believe it is so easy to use it."

# Tokenize input & generate prediction
inputs = tokenizer.encode(text, return_tensors="pt")
outputs = model.generate(inputs, max_length=50)

# Decode & print prediction
translation = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(f"Translation: {translation}")
```

```
Translation: Ich liebe dieses Produkt, das ich nicht glaube, dass es so einfach zu benutzen ist.
```

Conclusion

Natural Language Processing is an ever-evolving field with applications across industries. This guide covered foundational techniques such as tokenization, stemming, lemmatization, and deep learning-based NLP models.

References

https://www.geeksforgeeks.org/what-is-sentiment-analysis/
https://www.analyticsvidhya.com/blog/2022/07/sentiment-analysis-using-python/
https://www.geeksforgeeks.org/next-word-prediction-with-deep-learning-in-nlp/
https://www.analyticsvidhya.com/blog/2023/07/next-word-prediction-with-bidirectional-lstm/
https://www.analyticsvidhya.com/blog/2021/08/predict-the-next-word-of-your-text-using-long-short-term-memory-lstm/
https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

https://github.com/aswintechguy/Data-Science-Concepts/blob/main/NLP/Natural%20Language%20Processing(NLP)%20Concepts%20-%20Hackers%20Realm.ipynb
https://huggingface.co/docs/transformers/en/index
https://huggingface.co/google-t5/t5-small
https://github.com/google-research/text-to-text-transfer-transformer
https://www.geeksforgeeks.org/text-preprocessing-for-nlp-tasks/
https://www.geeksforgeeks.org/feature-extraction-techniques-nlp/
https://www.geeksforgeeks.org/word-embeddings-in-nlp/