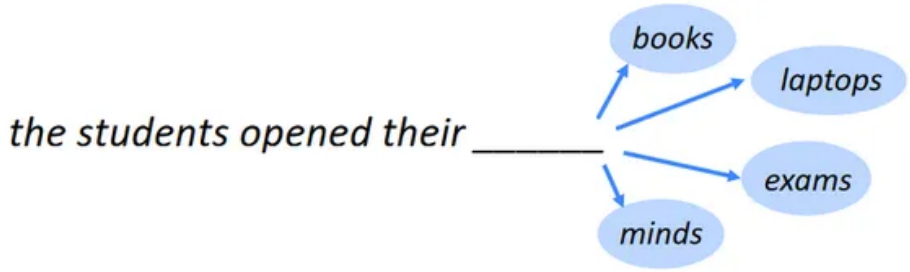
Mastering Next Word Prediction with Bi-LSTM: A Comprehensive Guide

29 Nov, 2023 • 10 min read

Introduction

Identifying the following word is the task of next-word prediction, also known as language modeling. One of the NLP's benchmark tasks is language modeling. In its most basic form, it entails picking the word that follows a string of words based on them that is most likely to occur. In many different fields, language modeling has a wide variety of applications.



Learning Objective

- Recognize the underlying ideas and principles behind the numerous models used in statistical analysis, machine learning, and data science.
- Learn how to create predictive models, including regression, classification, clustering, etc., to generate precise predictions and types based on data.
- Understand the principles of overfitting and underfitting, and learn how to evaluate model performance using measures like accuracy, precision, recall, etc.
- Learn how to preprocess data and identify pertinent characteristics for modeling.
- Learn how to tweak hyperparameters and optimize models using grid search and cross-validation.

This article was published as a part of the Data Science Blogathon.

Table of contents

Applications of Language Modeling

Here are some notable applications of language modeling:

Mobile Keyboard Text Recommendation

A function on smartphone keyboards called mobile keyboard text recommendation, or predictive text or auto-suggestions, suggests words or phrases as you write. It seeks to make typing faster and less error-prone and to offer more precise and contextually appropriate recommendations.

Also Read: Building a Content-Based Recommendation System

Google Search Auto-Completion

Every time we use a search engine like Google to look for anything, we receive many ideas, and as we keep adding phrases, the recommendations grow better and more relevant to our current search. How will it happen, then?

Natural language processing (NLP) technology makes it feasible. Here, we'll employ natural language processing (NLP) to create a prediction model utilizing a bidirectional LSTM (Long short-term memory) model to foretell the sentence's remaining words.

Learn More: What is LSTM? Introduction to Long Short-Term Memory

Import Necessary Libraries and Packages

Importing the necessary libraries and packages to construct a next-word prediction model using a bidirectional LSTM would be best. A sample of the libraries you'll generally require is shown below:

```
import pandas as pd
import os
import numpy as np

import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding, LSTM, Dense, Bidirectional
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
```

Dataset Information

Understanding the features and attributes of the dataset you're dealing with requires knowledge. The following seven publications' medium articles, selected at random and published in 2019, are included in this dataset:

- Towards Data Science
- UX Collective
- The Startup
- The Writing Cooperative
- Data Driven Investor
- Better Humans
- Better Marketing

Dataset Link: https://www.kaggle.com/code/ysthehurricane/next-word-prediction-bi-lstm-tutorial-easy-way/input

```
medium_data = pd.read_csv('../input/medium-articles-dataset/medium_data.csv')
medium_data.head()
```

	id	url	title	subtitle	image	claps	responses	reading_time	publication	da
0	1	https://towardsdatascience.com/a-beginners-gui...	A Beginner's Guide to Word Embedding with Gens...	NaN	1.png	850	8	8	Towards Data Science	2(
1	2	https://towardsdatascience.com/hands-on-graph-...	Hands-on Graph Neural Networks with PyTorch & ...	NaN	2.png	1100	11	9	Towards Data Science	2(
2	3	https://towardsdatascience.com/how-to-use-ggpl...	How to Use ggplot2 in Python	A Grammar of Graphics for Python	3.png	767	1	5	Towards Data Science	2(
3	4	https://towardsdatascience.com/databricks-how-...	Databricks: How to Save Files in CSV on Your L...	When I work on Python projects dealing...	4.jpeg	354	0	4	Towards Data Science	2(
4	5	https://towardsdatascience.com/a-step-by-step-...	A Step-by-Step Implementation of Gradient Desc...	One example of building neural...	5.jpeg	211	3	4	Towards Data Science	2(

Here, we have ten different fields and 6508 records but we will only use the title field for predicting the next word.

```
print("Number of records: ", medium_data.shape[0])
print("Number of fields: ", medium_data.shape[1])
```

```
Number of records:  6508
Number of fields:  10
```

By looking through and comprehending the dataset information, you may choose the preprocessing procedures, model, and evaluation metrics for your next word prediction challenge.

Display Titles of Various Articles and Preprocess Them

Let's have a look at a few sample titles to illustrate the preparation of article titles:

```
medium_data['title']
```

```
0       A Beginner's Guide to Word Embedding with Gens...
1       Hands-on Graph Neural Networks with PyTorch & ...
2                           How to Use ggplot2 in Python
3       Databricks: How to Save Files in CSV on Your L...
4       A Step-by-Step Implementation of Gradient Desc...
                              ...
6503    "We" vs "I" - How Should You Talk About Yourse...
6504                    How Donald Trump Markets Himself
6505        Content and Marketing Beyond Mass Consumption
6506    5 Questions All Copywriters Should Ask Clients...
6507                How To Write a Good Business Blog Post
Name: title, Length: 6508, dtype: object
```

Removing Unwanted Characters and Words in Titles

Preprocessing text data for prediction tasks sometimes includes removing undesirable letters and phrases from titles. Unwanted letters and words might contaminate the data with noise and add needless complexity, thereby lowering the model's performance and accuracy.

1. **Unwanted Characters:**
    1. **Punctuation:** You should remove exclamation points, question marks, commas, and other punctuation. Typically, you can safely discard them because they usually don't help with the prediction assignment
    2. **Special Characters:** Remove non-alphanumeric symbols, such as dollar signs, @ symbols, hashtags, and other special characters, that are unnecessary for the prediction job.
    3. **HTML Tags:** If the titles have HTML markups or tags, remove them using the proper tools or libraries to extract the text.
2. **Unwanted Words:**
    1. **Stop Words:** Remove common stop words such as "a," "an," "the," "is," "in," and other frequently occurring words that do not carry significant meaning or predictive power.
    2. **Irrelevant Words:** Identify and remove specific words that are not relevant to the prediction task or domain. For example, if you are predicting movie genres, words like "movie" or "film" may not provide helpful information.

```
medium_data['title'] = medium_data['title'].apply(lambda x: x.replace(u'\xa0',u' '))
medium_data['title'] = medium_data['title'].apply(lambda x: x.replace('\u200a',' '))
```

Tokenization

Tokenization divides the text into tokens, words, subwords, or characters and then assigns a unique ID or index to each token, creating a word index or Vocabulary.

The tokenization process involves the following steps:

Text preprocessing: Preprocess the text by eliminating punctuation, changing it to lowercase, and taking care of any particular task- or domain-specific needs.

Tokenization: Dividing the preprocessed text into separate tokens by predetermined rules or methods. Regular expressions, separating by whitespace, and employing specialized tokenizers are all common tokenization techniques.

Increasing Vocabulary You can make a dictionary, also called a word index, by assigning each token a unique ID or index. In this process, each ticket is mapped to the relevant index value.

```
tokenizer = Tokenizer(oov_token='<oov>') # For those words which are not found in word_index
tokenizer.fit_on_texts(medium_data['title'])
total_words = len(tokenizer.word_index) + 1

print("Total number of words: ", total_words)
print("Word: ID")
print("------------")
print("<oov>: ", tokenizer.word_index['<oov>'])
print("Strong: ", tokenizer.word_index['strong'])
print("And: ", tokenizer.word_index['and'])
print("Consumption: ", tokenizer.word_index['consumption'])
```

By transforming text into a vocabulary or word index, you can create a lookup table representing the text as a collection of numerical indexes. Each unique word in the text receives a corresponding index value, allowing for further processing or modeling operations that require numerical input.

```
Total number of words:  8238
Word: ID
------------
<oov>:  1
Strong:  4
And:  8
Consumption:  8237
```

Titles Text into Sequences and Make N_gram Model.

These stages can be used to build an n-gram model for accurate prediction based on title sequences:

1. **Convert Titles to Sequences:** Use a tokenizer to turn each title into a string of tokens or manually separate each slip into its constituent words. Assign each word in the lexicon a distinct number index.
2. **Generate n-grams:** From the sequences, make n-grams. A continuous run of n-title tokens is called an n-gram.
3. **Count the Frequency:** Determine the frequency at which each n-gram appears in the dataset.
4. **Build the n-gram Model:** Create the n-gram model using the n-gram frequencies. The model keeps track of each token probability given the previous n-1 tokens. This can be displayed as a lookup table or a dictionary.
5. **Predict the Next Word:** The expected next token in an n-1-token sequence may be identified using the n-gram model. To do this, it is necessary to find the probability in the algorithm and select a token with the greatest likelihood.

Learn More: What Are N-grams and How to Implement Them in Python?

You can use these stages to build an n-gram model that utilizes the titles' sequences to predict the next word or token. Based on the training data, this method can produce accurate predictions since it captures the statistical relationships and trends in the language usage of the titles.

Example of N-gram Model

Sentence: I am a Data Scientist

Word index: {'I': 1, 'am': 2, 'a': 3, 'Data': 4, 'scientist': 5}

Text Sequence: [1,2,3,4,5]

N_gram model for this sentence:
[1,2,3,4,5] = ['I', 'am', 'a', 'Data', 'Scientist']
[1,2,3,4] = ['I', 'am', 'a', 'Data']
[1,2,3] = ['I', 'am', 'a']
[1,2] = ['I', 'am']

```
input_sequences = []
for line in medium_data['title']:
    token_list = tokenizer.texts_to_sequences([line])[0]
    #print(token_list)

    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

# print(input_sequences)
print("Total input sequences: ", len(input_sequences))
```

```
Total input sequences:  48461
```

Make All Titles the Same Length by Using Padding

You may use padding to ensure that each title is the same size by following these steps:

- Find the longest title in your dataset by comparing all the other titles.
- Repeat this process for each title, comparing each one's length to the overall limit.
- When a title is too short, it should be extended using a specific padding token or character.
- For each title in your dataset, carry out the padding procedure again.

Padding will ensure that all titles are the same length and will provide consistency for post-processing or model training.

```
# pad sequences
max_sequence_len = max([len(x) for x in input_sequences])
input_sequences = np.array(pad_sequences(input_sequences, maxlen=max_sequence_len, padding='pre
input_sequences[1]
```

```
array([  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   5, 676,
        68], dtype=int32)
```

Prepare Features and Labels

In the given scenario, if we consider the last element of each input sequence as the label, we can perform one-hot encoding on the titles to represent them as vectors corresponding to the total number of unique words.

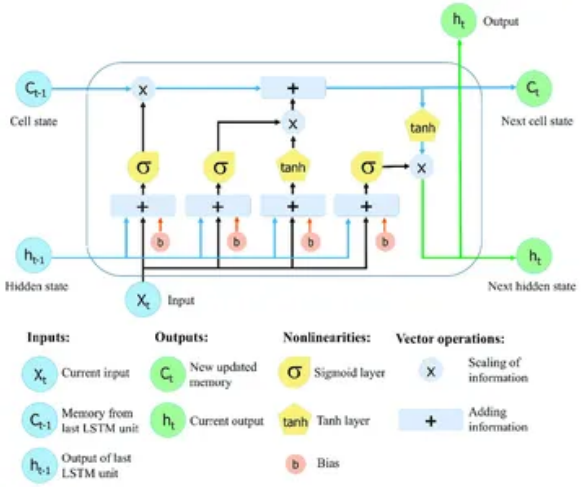
```
# create features and label
xs, labels = input_sequences[:,:-1],input_sequences[:,-1]
ys = tf.keras.utils.to_categorical(labels, num_classes=total_words)


print(xs[5])
print(labels[5])
print(ys[5][14])
```

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   5 676  68   2 452 1518]
14
1.0
```

The Architecture of Bidirectional LSTM Neural Network

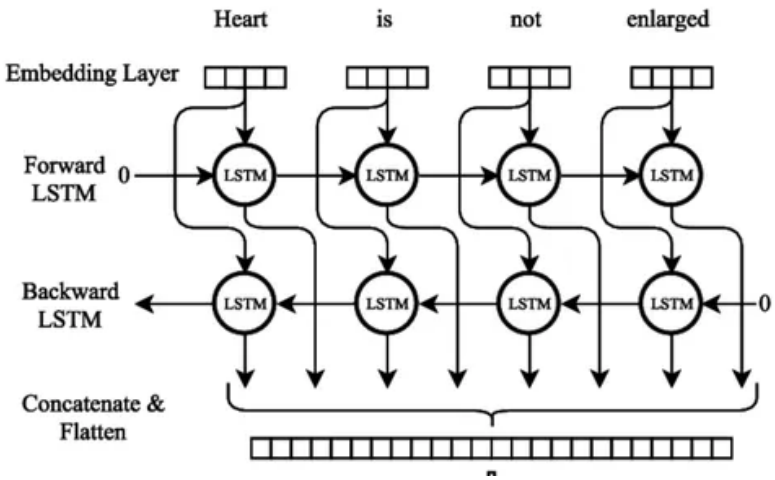
Recurrent neural networks (RNNs) with Long Short-Term Memory (LSTM) can collect and hold information across extensive sequences. LSTM networks use specialized memory cells and gating techniques to overcome the constraints of regular RNNs, which frequently struggle with the vanishing gradient problem and have trouble maintaining long-term dependence.



The critical feature of LSTM networks is the cell state, which serves as a memory unit that can store information over time. The cell state is protected and controlled by three main gates: the forget gate, the input gate, and the output gate. These gates regulate the flow of information into, out of, and within the LSTM cell, allowing the network to remember or forget information at different time steps selectively.

Learn More: Long Short Term Memory | Architecture Of LSTM

Bidirectional LSTM



Bi-LSTM Neural Network Model training

Numerous crucial procedures must be followed while training a bidirectional LSTM (Bi-LSTM) neural network model. The first step is compiling a training dataset with input and output sequences corresponding to them, indicating the next word. The text data must be preprocessed by being divided into separate lines, removing the punctuation, and changing the case to lowercase.

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_len-1))
model.add(Bidirectional(LSTM(150)))
model.add(Dense(total_words, activation='softmax'))
adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
history = model.fit(xs, ys, epochs=50, verbose=1)
#print model.summary()
print(model)
```

By calling the fit() method, the model is trained. The training data consists of the input sequences (xs) and matching output sequences (ys). The model proceeds through 50 iterations, going through the whole training set. During the training process, the training progress is shown (verbose=1).

```
Epoch 39/50
1515/1515 [==============================] - 24s 16ms/step - loss: 2.0000 - accuracy: 0.5354
Epoch 40/50
1515/1515 [==============================] - 24s 16ms/step - loss: 2.0599 - accuracy: 0.5301
Epoch 41/50
1515/1515 [==============================] - 25s 16ms/step - loss: 2.0144 - accuracy: 0.5322
Epoch 42/50
1515/1515 [==============================] - 24s 16ms/step - loss: 1.9902 - accuracy: 0.5398
Epoch 43/50
1515/1515 [==============================] - 25s 16ms/step - loss: 1.9940 - accuracy: 0.5391
Epoch 44/50
1515/1515 [==============================] - 24s 16ms/step - loss: 1.9925 - accuracy: 0.5427
Epoch 45/50
1515/1515 [==============================] - 25s 16ms/step - loss: 2.0268 - accuracy: 0.5319
Epoch 46/50
1515/1515 [==============================] - 24s 16ms/step - loss: 2.0190 - accuracy: 0.5366
Epoch 47/50
1515/1515 [==============================] - 25s 16ms/step - loss: 2.0222 - accuracy: 0.5357
Epoch 48/50
1515/1515 [==============================] - 24s 16ms/step - loss: 1.9856 - accuracy: 0.5407
Epoch 49/50
1515/1515 [==============================] - 25s 16ms/step - loss: 1.9503 - accuracy: 0.5443
Epoch 50/50
1515/1515 [==============================] - 24s 16ms/step - loss: 1.9424 - accuracy: 0.5500
<tensorflow.python.keras.engine.sequential.Sequential object at 0x7f75d2b2a910>
```

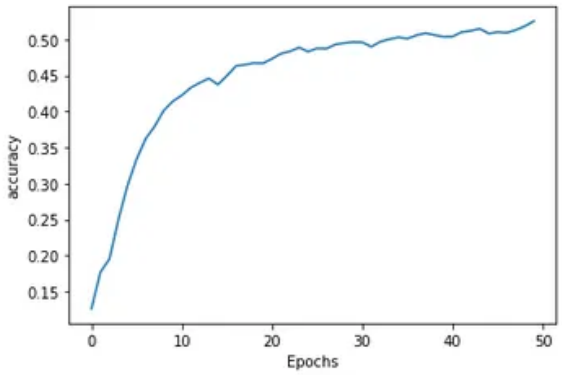
Plotting Model Accuracy and Loss

Plotting a model's accuracy and loss throughout training offers insightful information about how well it performs and how training is going. The mistake or disparity between the anticipated and actual values is called loss. Whereas the percentage of accurate predictions generated by the model is known as accuracy.

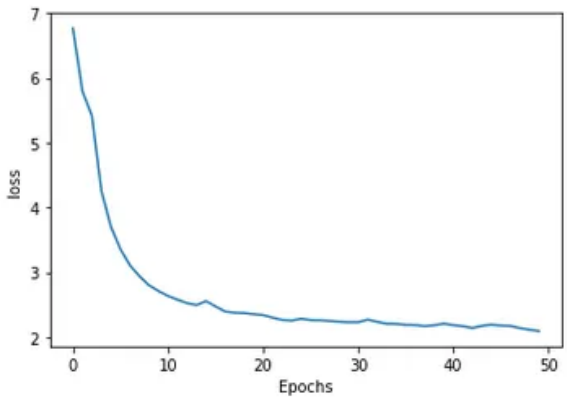
```
import matplotlib.pyplot as plt


def plot_graphs(history, string):
    plt.plot(history.history[string])
    plt.xlabel("Epochs")
    plt.ylabel(string)
    plt.show()

plot_graphs(history, 'accuracy')
```




```
plot_graphs(history, 'loss')
```



Predicting the Next Word of the Title

A fascinating challenge in natural language processing is guessing the following word in a title. Models can propose the most likely talk by looking for patterns and correlations in text data. This predictive power makes applications like text suggestion systems and autocomplete possible. Sophisticated approaches like RNNs and transformer-based architectures increase accuracy and capture contextual relationships.

```
seed_text = "implementation of"
next_words = 2

for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')
    predicted = model.predict_classes(token_list, verbose=0)
```

```
    output_word = ""
    for word, index in tokenizer.word_index.items():
        if index == predicted:
            output_word = word
            break
    seed_text += " " + output_word
print(seed_text)
```

```
/opt/conda/lib/python3.7/site-packages/tensorflow/python/keras/engine/sequential.py:450: UserWar
ning: `model.predict_classes()` is deprecated and will be removed after 2021-01-01. Please use i
nstead:* `np.argmax(model.predict(x), axis=-1)`,   if your model does multi-class classification
(e.g. if it uses a `softmax` last-layer activation).* `(model.predict(x) > 0.5).astype("int32")
`,   if your model does binary classification   (e.g. if it uses a `sigmoid` last-layer activati
on).
  warnings.warn('`model.predict_classes()` is deprecated and '

implementation of rnn lstm
```

Conclusion

In conclusion, training a model to predict the subsequent word in a string of words is the exciting natural language processing challenge known as next-word prediction using a Bidirectional LSTM. Here's the conclusion summarized in bullet points:

- The potent deep learning architecture BI-LSTM for sequential data processing may capture long-range relationships and phrase context.
- To prepare raw text data for BI-LSTM training, data preparation is essential. This includes tokenization, vocabulary generation, and text vectorization.
- Creating a loss function, building the model using an optimizer, fitting it to preprocessed data, and assessing its performance on validation sets are the steps in training the BI-LSTM model.
- BI-LSTM next word prediction takes a combination of theoretical knowledge and hands-on experimentation to master.
- Auto-completion, language creation, and text suggestion algorithms are examples of next-word prediction model applications.

Applications for next-word prediction include chatbots, machine translation, and text completion. You can create more precise and context-aware next-word prediction models with more research and improvement.