Comprehensive Guide to Text Summarization using Deep Learning in Python

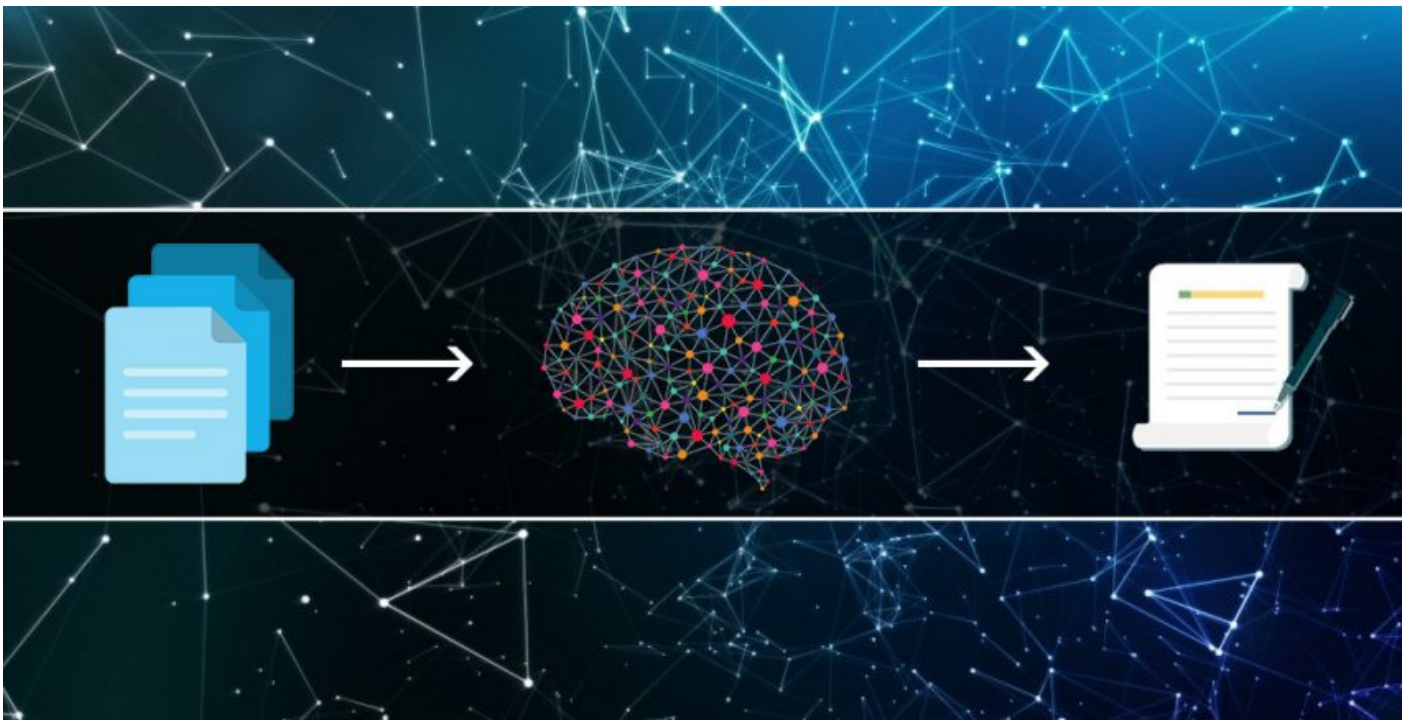
ADVANCED DEEP LEARNING NLP PROJECT PYTHON SEQUENCE MODELING SUPERVISED TEXT UNSTRUCTURED DATA

Introduction

"I don't want a full report, just give me a summary of the results". I have often found myself in this situation – both in college as well as my professional life. We prepare a comprehensive report and the teacher/supervisor only has time to read the summary.

Sounds familiar? Well, I decided to do something about it. Manually converting the report to a summarized version is too time taking, right? Could I lean on Natural Language Processing (NLP) techniques to help me out?

This is where the awesome concept of Text Summarization using Deep Learning really helped me out. It solves the one issue which kept bothering me before – **now our model can understand the context of the entire text**. It's a dream come true for all of us who need to come up with a quick summary of a document!



And the results we achieve using text summarization in deep learning? Remarkable. So in this article, we will walk through a step-by-step process for building a **Text Summarizer using Deep Learning** by covering all the concepts required to build it. And then we will implement our first text summarization model in Python!

Note: This article requires a basic understanding of a few deep learning concepts. I recommend going through the below articles.

- *A Must-Read Introduction to Sequence Modelling (with use cases)*

- *Must-Read Tutorial to Learn Sequence Modeling (deeplearning.ai Course #5)*
- *Essentials of Deep Learning: Introduction to Long Short Term Memory*

Table of Contents

I've kept the 'how does the attention mechanism work?' section at the bottom of this article. It's a math-heavy section and is not mandatory to understand how the Python code works. However, I encourage you to go through it because it will give you a solid idea of this awesome NLP concept.

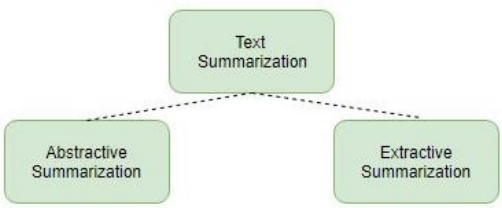
What is Text Summarization in NLP?

Let's first understand what text summarization is before we look at how it works. Here is a succinct definition to get us started:

> "Automatic text summarization is the task of producing a concise and fluent summary while preserving key information content and overall meaning"
>
> -Text Summarization Techniques: A Brief Survey, 2017

There are broadly two different approaches that are used for text summarization:

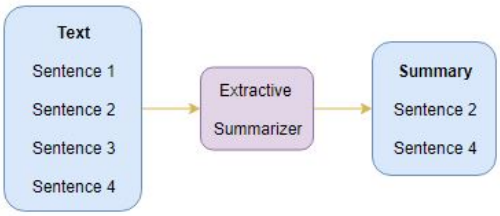
- Extractive Summarization
- Abstractive Summarization



Let's look at these two types in a bit more detail.

Extractive Summarization

The name gives away what this approach does. **We identify the important sentences or phrases from the original text and extract only those from the text.** Those extracted sentences would be our summary. The below diagram illustrates extractive summarization:

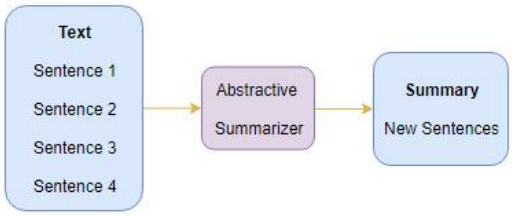


I recommend going through the below article for building an extractive text summarizer using the TextRank algorithm:

- An Introduction to Text Summarization using the TextRank Algorithm (with Python implementation)

Abstractive Summarization

This is a very interesting approach. Here, we generate new sentences from the original text. This is in contrast to the extractive approach we saw earlier where we used only the sentences that were present. The sentences generated through abstractive summarization might not be present in the original text:



You might have guessed it – we are going to build an Abstractive Text Summarizer using Deep Learning in this article! Let's first understand the concepts necessary for building a Text Summarizer model before diving into the implementation part.

Exciting times ahead!

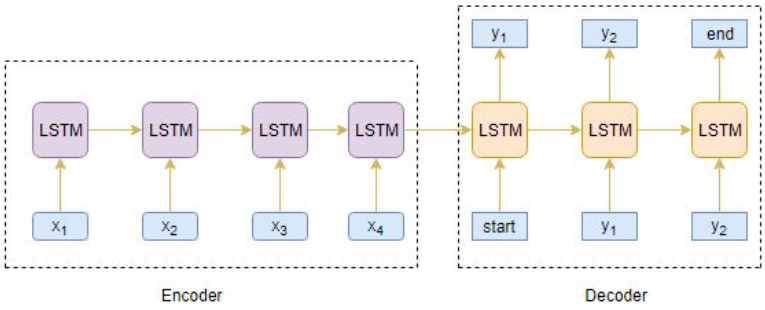
Introduction to Sequence-to-Sequence (Seq2Seq) Modeling

We can build a Seq2Seq model on any problem which involves sequential information. This includes Sentiment classification, Neural Machine Translation, and Named Entity Recognition – some very common applications of sequential information.

In the case of Neural Machine Translation, the input is a text in one language and the output is also a text in another language:

In the Named Entity Recognition, the input is a sequence of words and the output is a sequence of tags for every word in the input sequence:

Our objective is to build a text summarizer where the input is a long sequence of words (in a text body), and the output is a short summary (which is a sequence as well). So, **we can model this as a Many-to-Many Seq2Seq problem.** Below is a typical Seq2Seq model architecture:



There are two major components of a Seq2Seq model:

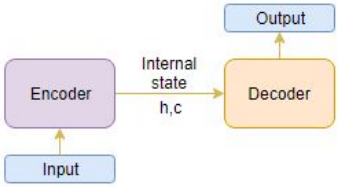
- Encoder
- Decoder

Let's understand these two in detail. These are essential to understand how text summarization works underneath the code. You can also check out this tutorial to understand sequence-to-sequence modeling in more detail.

Understanding the Encoder-Decoder Architecture

> The Encoder-Decoder architecture is mainly used to solve the sequence-to-sequence (Seq2Seq) problems where the input and output sequences are of different lengths.

Let's understand this from the perspective of text summarization. The input is a long sequence of words and the output will be a short version of the input sequence.



Generally, variants of Recurrent Neural Networks (RNNs), i.e. Gated Recurrent Neural Network (GRU) or Long Short Term Memory (LSTM), are preferred as the encoder and decoder components. This is because they are capable of capturing long term dependencies by overcoming the problem of vanishing gradient.

We can set up the Encoder-Decoder in 2 phases:

- Training phase
- Inference phase

Let's understand these concepts through the lens of an LSTM model.

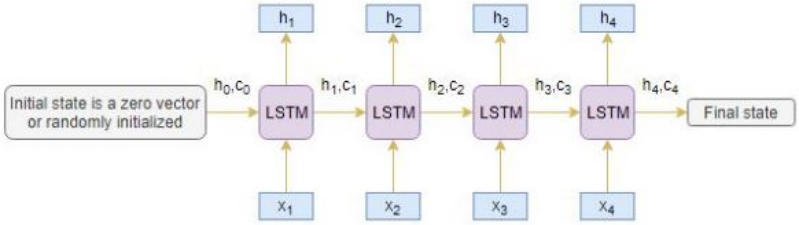
Training phase

In the training phase, we will first set up the encoder and decoder. We will then train the model to predict the target sequence offset by one timestep. Let us see in detail on how to set up the encoder and decoder.

Encoder

An Encoder Long Short Term Memory model (LSTM) reads the entire input sequence wherein, at each timestep, one word is fed into the encoder. It then processes the information at every timestep and captures the contextual information present in the input sequence.

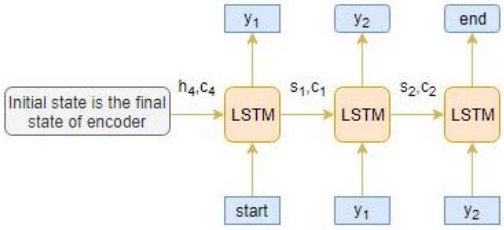
I've put together the below diagram which illustrates this process:



The hidden state (h_i) and cell state (c_i) of the last time step are used to initialize the decoder. Remember, this is because the encoder and decoder are two different sets of the LSTM architecture.

Decoder

The decoder is also an LSTM network which reads the entire target sequence word-by-word and predicts the same sequence offset by one timestep. **The decoder is trained to predict the next word in the sequence given the previous word.**

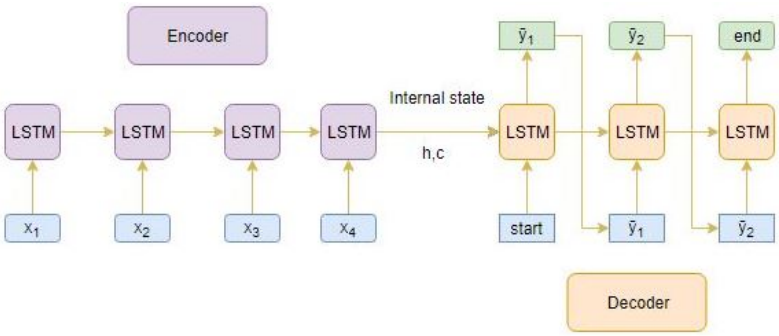


<start> and <**end>** are the special tokens which are added to the target sequence before feeding it into the decoder. The target sequence is unknown while decoding the test sequence. So, we start predicting the target sequence by passing the first word into the decoder which would be always the <**start>** token. And the <**end>** token signals the end of the sentence.

Pretty intuitive so far.

Inference Phase

After training, the model is tested on new source sequences for which the target sequence is unknown. So, we need to set up the inference architecture to decode a test sequence:



How does the inference process work?

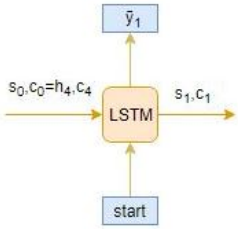
Here are the steps to decode the test sequence:

1. Encode the entire input sequence and initialize the decoder with internal states of the encoder
2. Pass <**start>** token as an input to the decoder
3. Run the decoder for one timestep with the internal states
4. The output will be the probability for the next word. The word with the maximum probability will be selected
5. Pass the sampled word as an input to the decoder in the next timestep and update the internal states with the current time step
6. Repeat steps 3 – 5 until we generate <**end>** token or hit the maximum length of the target sequence

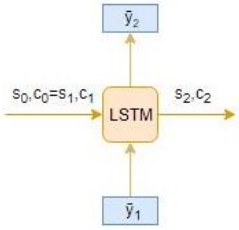
Let's take an example where the test sequence is given by $[x_1, x_2, x_3, x_4]$. How will the inference process work for this test sequence? I want you to think about it before you look at my thoughts below.

1. Encode the test sequence into internal state vectors
2. Observe how the decoder predicts the target sequence at each timestep:

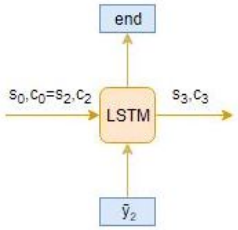
Timestep: t=1



Timestep: t=2



And, Timestep: t=3



Limitations of the Encoder – Decoder Architecture

As useful as this encoder-decoder architecture is, there are certain limitations that come with it.

- The encoder converts the entire input sequence into a fixed length vector and then the decoder predicts the output sequence. **This works only for short sequences** since the decoder is looking at the entire input sequence for the prediction
- Here comes the problem with long sequences. **It is difficult for the encoder to memorize long sequences into a fixed length vector**

> "A potential issue with this encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences. The performance of a basic encoder-decoder deteriorates rapidly as the length of an input sentence increases."
>
> -Neural Machine Translation by Jointly Learning to Align and Translate

So how do we overcome this problem of long sequences? This is where the concept of **attention mechanism** comes into the picture. It aims to predict a word by looking at a few specific parts of the sequence only, rather than the entire sequence. It really is as awesome as it sounds!

The Intuition behind the Attention Mechanism

> How much attention do we need to pay to every word in the input sequence for generating a word at timestep **t**? That's the key intuition behind this attention mechanism concept.

Let's consider a simple example to understand how Attention Mechanism works:

- **Source sequence:** "Which sport do you like the most?
- **Target sequence:** "I love cricket"

The first word **'I'** in the target sequence is connected to the fourth word **'you'** in the source sequence, right? Similarly, the second-word **'love'** in the target sequence is associated with the fifth word **'like'** in the source sequence.

So, instead of looking at all the words in the source sequence, we can increase the importance of specific parts of the source sequence that result in the target sequence. This is the basic idea behind the attention mechanism.

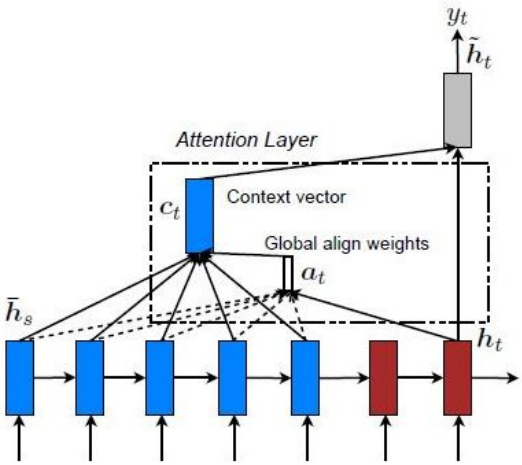
There are 2 different classes of attention mechanism depending on the way the attended context vector is derived:

- Global Attention
- Local Attention

Let's briefly touch on these classes.

Global Attention

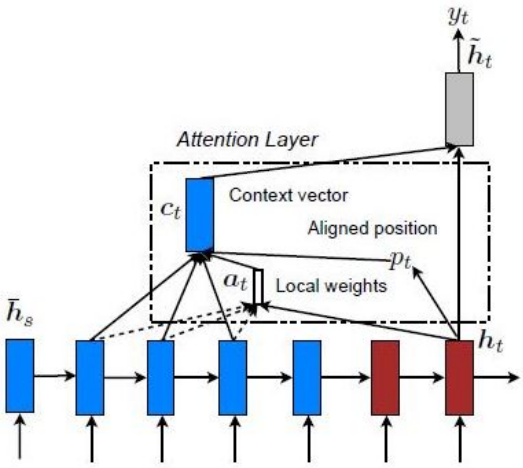
Here, the attention is placed on all the source positions. In other words, **all the hidden states of the encoder are considered for deriving the attended context vector:**



Source: Effective Approaches to Attention-based Neural Machine Translation – 2015

Local Attention

Here, the attention is placed on only a few source positions. **Only a few hidden states of the encoder are considered for deriving the attended context vector:**



Source: Effective Approaches to Attention-based Neural Machine Translation – 2015

We will be using the Global Attention mechanism in this article.

Understanding the Problem Statement

Customer reviews can often be long and descriptive. Analyzing these reviews manually, as you can imagine, is really time-consuming. This is where the brilliance of Natural Language Processing can be applied to generate a summary for long reviews.

We will be working on a really cool dataset. **Our objective here is to generate a summary for the Amazon Fine Food reviews using the abstraction-based approach we learned about above.**

You can download the dataset from here.

Implementing Text Summarization in Python using Keras

It's time to fire up our Jupyter notebooks! Let's dive into the implementation details right away.

Custom Attention Layer

Keras does not officially support attention layer. So, we can either implement our own attention layer or use a third-party implementation. We will go with the latter option for this article. You can download the attention layer from **here** and copy it in a different file called **attention.py**.

Let's import it into our environment:

```
from attention import AttentionLayer
```

view raw

attention.py hosted with ❤ by **GitHub**

Import the Libraries

```
import numpy as np
import pandas as pd
import re
from bs4 import BeautifulSoup
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from nltk.corpus import stopwords
from tensorflow.keras.layers import Input, LSTM, Embedding, Dense, Concatenate, TimeDistributed, Bidirectional
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping
import warnings
pd.set_option("display.max_colwidth", 200)
warnings.filterwarnings("ignore")
```

view raw

libraries.py hosted with ❤ by **GitHub**

Read the dataset

This dataset consists of reviews of fine foods from Amazon. The data spans a period of more than 10 years, including all ~500,000 reviews up to October 2012. These reviews include product and user information, ratings, plain text review, and summary. It also includes reviews from all other Amazon categories.

We'll take a sample of 100,000 reviews to reduce the training time of our model. Feel free to use the entire dataset for training your model if your machine has that kind of computational power.

```
data=pd.read_csv("../input/amazon-fine-food-reviews/Reviews.csv",nrows=100000)
```

view raw

readfile.py hosted with ❤ by **GitHub**

Drop Duplicates and NA values

```
data.drop_duplicates(subset=['Text'],inplace=True)  #dropping duplicates
data.dropna(axis=0,inplace=True)   #dropping na
```

view raw

duplicates.py hosted with ❤ by GitHub

Preprocessing

Performing basic preprocessing steps is very important before we get to the model building part. Using messy and uncleaned text data is a potentially disastrous move. So in this step, we will drop all the unwanted symbols, characters, etc. from the text that do not affect the objective of our problem.

Here is the dictionary that we will use for expanding the contractions:

```
contraction_mapping = {"ain't": "is not", "aren't": "are not","can't": "cannot", "'cause": "because", "could've": "could have",

                       "didn't": "did not", "doesn't": "does not", "don't": "do not", "hadn't": "had not", "hasn't": "has no

                       "he'd": "he would","he'll": "he will", "he's": "he is", "how'd": "how did", "how'd'y": "how do you",

                       "I'd": "I would", "I'd've": "I would have", "I'll": "I will", "I'll've": "I will have","I'm": "I am",

                       "i'd've": "i would have", "i'll": "i will",  "i'll've": "i will have","i'm": "i am", "i've": "i have"

                       "it'd've": "it would have", "it'll": "it will", "it'll've": "it will have","it's": "it is", "let's":

                       "mayn't": "may not", "might've": "might have","mightn't": "might not","mightn't've": "might not have"

                       "mustn't": "must not", "mustn't've": "must not have", "needn't": "need not", "needn't've": "need not

                       "oughtn't": "ought not", "oughtn't've": "ought not have", "shan't": "shall not", "sha'n't": "shall no

                       "she'd": "she would", "she'd've": "she would have", "she'll": "she will", "she'll've": "she will have

                       "should've": "should have", "shouldn't": "should not", "shouldn't've": "should not have", "so've": "s

                       "this's": "this is","that'd": "that would", "that'd've": "that would have", "that's": "that is", "the

                       "there'd've": "there would have", "there's": "there is", "here's": "here is","they'd": "they would",

                       "they'll": "they will", "they'll've": "they will have", "they're": "they are", "they've": "they have"

                       "wasn't": "was not", "we'd": "we would", "we'd've": "we would have", "we'll": "we will", "we'll've":

                       "we've": "we have", "weren't": "were not", "what'll": "what will", "what'll've": "what will have", "w

                       "what's": "what is", "what've": "what have", "when's": "when is", "when've": "when have", "where'd":

                       "where've": "where have", "who'll": "who will", "who'll've": "who will have", "who's": "who is", "who

                       "why's": "why is", "why've": "why have", "will've": "will have", "won't": "will not", "won't've": "wi

                       "would've": "would have", "wouldn't": "would not", "wouldn't've": "would not have", "y'all": "you all

                       "y'all'd": "you all would","y'all'd've": "you all would have","y'all're": "you all are","y'all've": "

                       "you'd": "you would", "you'd've": "you would have", "you'll": "you will", "you'll've": "you will have

                       "you're": "you are", "you've": "you have"}
```

view raw

contraction.py hosted with ❤ by GitHub

We need to define two different functions for preprocessing the reviews and generating the summary since the preprocessing steps involved in text and summary differ slightly.

a) Text Cleaning

Let's look at the first 10 reviews in our dataset to get an idea of the text preprocessing steps:

```
data['Text'][:10]
```

Output:

```
0    I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product look
s more like a stew than a processed meat and it smells better. My Labr...
1           Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually small sized unsalted. Not sure if this wa
s an error or if the vendor intended to represent the product as "Jumbo".
2    This is a confection that has been around a few centuries.  It is a light, pillowy citrus gelatin with nuts - in this case Fil
berts. And it is cut into tiny squares and then liberally coated with ...
3    If you are looking for the secret ingredient in Robitussin I believe I have found it.  I got this in addition to the Root Beer
Extract I ordered (which was good) and made some cherry soda.  The fl...
4                                                  Great taffy at a great price.  There was a wide assortment of yummy
taffy.  Delivery was very quick.  If your a taffy lover, this is a deal.
5    I got a wild hair for taffy and ordered this five pound bag. The taffy was all very enjoyable with many flavors: watermelon, r
oot beer, melon, peppermint, grape, etc. My only complaint is there wa...
6    This saltwater taffy had great flavors and was very soft and chewy.  Each candy was individually wrapped well.  None of the ca
ndies were stuck together, which did happen in the expensive version, ...
7                                                  This taffy is so good.  It is very soft and chewy.  The flavors are
amazing.  I would definitely recommend you buying it.  Very satisfying!!
8                                                Right now I'm mostly just sprouting this so my cats can ea
t the grass. They love it. I rotate it around with Wheatgrass and Rye too
9                                                 This is a very healthy dog food. Good for their digestion. Also
good for small puppies. My dog eats her required amount at every feeding.
Name: Text, dtype: object
```

We will perform the below preprocessing tasks for our data:

- Convert everything to lowercase
- Remove HTML tags
- Contraction mapping
- Remove ('s)
- Remove any text inside the parenthesis ( )
- Eliminate punctuations and special characters
- Remove stopwords
- Remove short words

Let's define the function:

```
stop_words = set(stopwords.words('english'))
def text_cleaner(text):
    newString = text.lower()
    newString = BeautifulSoup(newString, "lxml").text
    newString = re.sub(r'\([^)]*\)', '', newString)
    newString = re.sub('"','', newString)
    newString = ' '.join([contraction_mapping[t] if t in contraction_mapping else t for t in newString.split(" ")])
    newString = re.sub(r"'s\b","",newString)
    newString = re.sub("[^a-zA-Z]", " ", newString)
    tokens = [w for w in newString.split() if not w in stop_words]
    long_words=[]
    for i in tokens:
        if len(i)>=3:                  #removing short word
            long_words.append(i)
    return (" ".join(long_words)).strip()
```

```
cleaned_text = []
for t in data['Text']:
    cleaned_text.append(text_cleaner(t))
```

view raw

textcleaning.py hosted with ❤ by GitHub

b) Summary Cleaning

And now we'll look at the first 10 rows of the reviews to an idea of the preprocessing steps for the summary column:

```
data['Summary'][:10]
```

view raw

readsummary.py hosted with ❤ by GitHub

Output:

```
0                           Good Quality Dog Food
1                               Not as Advertised
2                         "Delight" says it all
3                                Cough Medicine
4                                   Great taffy
5                                   Nice Taffy
6   Great!  Just as good as the expensive brands!
7                         Wonderful, tasty taffy
8                                   Yay Barley
9                               Healthy Dog Food
Name: Summary, dtype: object
```

Define the function for this task:

```
def summary_cleaner(text):
    newString = re.sub('"','', text)
    newString = ' '.join([contraction_mapping[t] if t in contraction_mapping else t for t in newString.split(" ")])
    newString = re.sub(r"'s\b","",newString)
    newString = re.sub("[^a-zA-Z]", " ", newString)
    newString = newString.lower()
    tokens=newString.split()
    newString=''
    for i in tokens:
        if len(i)>1:
            newString=newString+i+' '
    return newString

#Call the above function
cleaned_summary = []
for t in data['Summary']:
    cleaned_summary.append(summary_cleaner(t))

data['cleaned_text']=cleaned_text
data['cleaned_summary']=cleaned_summary
data['cleaned_summary'].replace('', np.nan, inplace=True)
data.dropna(axis=0,inplace=True)
```

view raw

summarycleaning.py hosted with ❤ by GitHub

Remember to add the **START** and **END** special tokens at the beginning and end of the summary:

```
data['cleaned_summary'] = data['cleaned_summary'].apply(lambda x : '_START_ '+ x + ' _END_')
```

view raw

append.py hosted with ❤ by GitHub

Now, let's take a look at the top 5 reviews and their summary:

```
for i in range(5):
    print("Review:",data['cleaned_text'][i])
```

```
    print("Summary:",data['cleaned_summary'][i])
    print("\n")
```

view raw

display.py hosted with ❤ by GitHub

Output:

```
Review: bought several vitality canned dog food products found good quality product looks like stew processed meat smells better la
brador finicky appreciates product better
Summary: _START_ good quality dog food  _END_


Review: product arrived labeled jumbo salted peanuts peanuts actually small sized unsalted sure error vendor intended represent pro
duct jumbo
Summary: _START_ not as advertised  _END_


Review: confection around centuries light pillowy citrus gelatin nuts case filberts cut tiny squares liberally coated powdered suga
r tiny mouthful heaven chewy flavorful highly recommend yummy treat familiar story lewis lion witch wardrobe treat seduces edmund s
elling brother sisters witch
Summary: _START_ delight says it all  _END_


Review: looking secret ingredient robitussin believe found got addition root beer extract ordered made cherry soda flavor medicinal
Summary: _START_ cough medicine  _END_


Review: great taffy great price wide assortment yummy taffy delivery quick taffy lover deal
Summary: _START_ great taffy  _END_
```

Understanding the distribution of the sequences

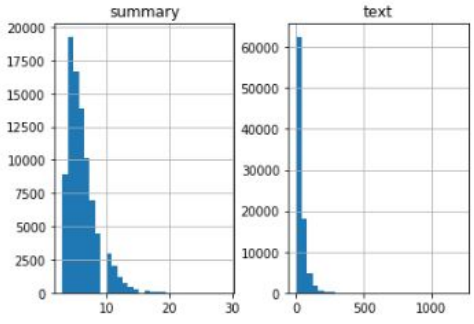
Here, we will analyze the length of the reviews and the summary to get an overall idea about the distribution of length of the text. This will help us fix the maximum length of the sequence:

```
import matplotlib.pyplot as plt
text_word_count = []
summary_word_count = []

# populate the lists with sentence lengths
for i in data['cleaned_text']:
      text_word_count.append(len(i.split()))

for i in data['cleaned_summary']:
      summary_word_count.append(len(i.split()))

length_df = pd.DataFrame({'text':text_word_count, 'summary':summary_word_count})
length_df.hist(bins = 30)
plt.show()
```

view raw

distribution.py hosted with ❤ by GitHub

Output:



Interesting. We can fix the maximum length of the reviews to 80 since that seems to be the majority review length. Similarly, we can set the maximum summary length to 10:

```
max_len_text=80
max_len_summary=10
```

view raw

maxlen.py hosted with ❤ by GitHub

We are getting closer to the model building part. Before that, we need to split our dataset into a training and validation set. We'll use 90% of the dataset as the training data and evaluate the performance on the remaining 10% (holdout set):

```
from sklearn.model_selection import train_test_split
x_tr,x_val,y_tr,y_val=train_test_split(data['cleaned_text'],data['cleaned_summary'],test_size=0.1,random_state=0,shuffle=True)
```

view raw

split.py hosted with ❤ by GitHub

Preparing the Tokenizer

A tokenizer builds the vocabulary and converts a word sequence to an integer sequence. Go ahead and build tokenizers for text and summary:

a) Text Tokenizer

```
#prepare a tokenizer for reviews on training data
x_tokenizer = Tokenizer()
x_tokenizer.fit_on_texts(list(x_tr))

#convert text sequences into integer sequences
x_tr    =   x_tokenizer.texts_to_sequences(x_tr)
x_val   =   x_tokenizer.texts_to_sequences(x_val)

#padding zero upto maximum length
x_tr    =   pad_sequences(x_tr,  maxlen=max_len_text, padding='post')
x_val   =   pad_sequences(x_val, maxlen=max_len_text, padding='post')

x_voc_size   =  len(x_tokenizer.word_index) +1
```

view raw

texttokenizer.py hosted with ❤ by GitHub

b) Summary Tokenizer

```
#preparing a tokenizer for summary on training data
y_tokenizer = Tokenizer()
y_tokenizer.fit_on_texts(list(y_tr))

#convert summary sequences into integer sequences
y_tr    =   y_tokenizer.texts_to_sequences(y_tr)
y_val   =   y_tokenizer.texts_to_sequences(y_val)

#padding zero upto maximum length
y_tr    =   pad_sequences(y_tr, maxlen=max_len_summary, padding='post')
y_val   =   pad_sequences(y_val, maxlen=max_len_summary, padding='post')

y_voc_size  =   len(y_tokenizer.word_index) +1
```

view raw

summarytokenizer.py hosted with ❤ by GitHub

Model building

We are finally at the model building part. But before we do that, we need to familiarize ourselves with a few terms which are required prior to building the model.

- **Return Sequences = True:** When the return sequences parameter is set to **True**, LSTM produces the hidden state and cell state for every timestep
- **Return State = True:** When return state = **True**, LSTM produces the hidden state and cell state of the last timestep only
- **Initial State:** This is used to initialize the internal states of the LSTM for the first timestep
- **Stacked LSTM:** Stacked LSTM has multiple layers of LSTM stacked on top of each other. This leads to a better representation of the sequence. I encourage you to experiment with the multiple layers of the LSTM stacked on top of each other (it's a great way to learn this)

Here, we are building a 3 stacked LSTM for the encoder:

```
from keras import backend as K
K.clear_session()
latent_dim = 500

# Encoder
encoder_inputs = Input(shape=(max_len_text,))
enc_emb = Embedding(x_voc_size, latent_dim,trainable=True)(encoder_inputs)

#LSTM 1
encoder_lstm1 = LSTM(latent_dim,return_sequences=True,return_state=True)
encoder_output1, state_h1, state_c1 = encoder_lstm1(enc_emb)

#LSTM 2
encoder_lstm2 = LSTM(latent_dim,return_sequences=True,return_state=True)
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)

#LSTM 3
encoder_lstm3=LSTM(latent_dim, return_state=True, return_sequences=True)
encoder_outputs, state_h, state_c= encoder_lstm3(encoder_output2)

# Set up the decoder.
decoder_inputs = Input(shape=(None,))
dec_emb_layer = Embedding(y_voc_size, latent_dim,trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

#LSTM using encoder_states as initial state
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs,decoder_fwd_state, decoder_back_state = decoder_lstm(dec_emb,initial_state=[state_h, state_c])

#Attention Layer
Attention layer attn_layer = AttentionLayer(name='attention_layer')
attn_out, attn_states = attn_layer([encoder_outputs, decoder_outputs])

# Concat attention output and decoder LSTM output
decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([decoder_outputs, attn_out])

#Dense layer
decoder_dense = TimeDistributed(Dense(y_voc_size, activation='softmax'))
decoder_outputs = decoder_dense(decoder_concat_input)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.summary()
```

view raw

model.py hosted with ❤ by **GitHub**

Output:

```
==================================================================================================
input_1 (InputLayer)            (None, 80)           0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 80, 500)      25785500    input_1[0][0]
__________________________________________________________________________________________________
lstm (LSTM)                     [(None, 80, 500), (N 2002000     embedding[0][0]
__________________________________________________________________________________________________
input_2 (InputLayer)            (None, None)         0
__________________________________________________________________________________________________
lstm_1 (LSTM)                   [(None, 80, 500), (N 2002000     lstm[0][0]
__________________________________________________________________________________________________
embedding_1 (Embedding)         (None, None, 500)    7048000     input_2[0][0]
__________________________________________________________________________________________________
lstm_2 (LSTM)                   [(None, 80, 500), (N 2002000     lstm_1[0][0]
__________________________________________________________________________________________________
lstm_3 (LSTM)                   [(None, None, 500),  2002000     embedding_1[0][0]
                                                                 lstm_2[0][1]
                                                                 lstm_2[0][2]
__________________________________________________________________________________________________
attention_layer (AttentionLayer [(None, None, 500),  500500      lstm_2[0][0]
                                                                 lstm_3[0][0]
__________________________________________________________________________________________________
concat_layer (Concatenate)      (None, None, 1000)   0           lstm_3[0][0]
                                                                 attention_layer[0][0]
__________________________________________________________________________________________________
time_distributed (TimeDistribut (None, None, 14096)  14110096    concat_layer[0][0]
==================================================================================================
Total params: 55,452,096
Trainable params: 55,452,096
Non-trainable params: 0
```

> I am using **sparse categorical cross-entropy** as the loss function since it converts the integer sequence to a one-hot vector on the fly. This overcomes any memory issues.

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy')
```

view raw

metrics.py hosted with ❤ by **GitHub**

Remember the concept of early stopping? It is used to stop training the neural network at the right time by monitoring a user-specified metric. Here, I am monitoring the validation loss (val_loss). Our model will stop training once the validation loss increases:

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```

view raw

callback.py hosted with ❤ by **GitHub**

We'll train the model on a batch size of 512 and validate it on the holdout set (which is 10% of our dataset):

```
history=model.fit([x_tr,y_tr[:,:-1]], y_tr.reshape(y_tr.shape[0],y_tr.shape[1], 1)[:,1:] ,epochs=50,callbacks=[es],batch_size=512
```

view raw

fit.py hosted with ❤ by **GitHub**

Understanding the Diagnostic plot

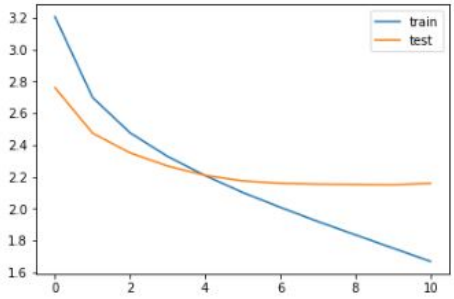
Now, we will plot a few diagnostic plots to understand the behavior of the model over time:

```
from matplotlib import pyplot
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend() pyplot.show()
```

view raw

plot.py hosted with ❤ by **GitHub**

Output:



We can infer that there is a slight increase in the validation loss after epoch 10. So, we will stop training the model after this epoch.

Next, let's build the dictionary to convert the index to word for target and source vocabulary:

```
reverse_target_word_index=y_tokenizer.index_word
reverse_source_word_index=x_tokenizer.index_word
target_word_index=y_tokenizer.word_index
```

view raw

reversedict.py hosted with ❤ by GitHub

Inference

Set up the inference for the encoder and decoder:

```
# encoder inference
encoder_model = Model(inputs=encoder_inputs,outputs=[encoder_outputs, state_h, state_c])

# decoder inference
# Below tensors will hold the states of the previous time step
decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_hidden_state_input = Input(shape=(max_len_text,latent_dim))

# Get the embeddings of the decoder sequence
dec_emb2= dec_emb_layer(decoder_inputs)

# To predict the next word in the sequence, set the initial states to the states from the previous time step
decoder_outputs2, state_h2, state_c2 = decoder_lstm(dec_emb2, initial_state=[decoder_state_input_h, decoder_state_input_c])

#attention inference
attn_out_inf, attn_states_inf = attn_layer([decoder_hidden_state_input, decoder_outputs2])
decoder_inf_concat = Concatenate(axis=-1, name='concat')([decoder_outputs2, attn_out_inf])

# A dense softmax layer to generate prob dist. over the target vocabulary
decoder_outputs2 = decoder_dense(decoder_inf_concat)

# Final decoder model
decoder_model = Model(
[decoder_inputs] + [decoder_hidden_state_input,decoder_state_input_h, decoder_state_input_c],
[decoder_outputs2] + [state_h2, state_c2])
```

view raw

inferencesetup.py hosted with ❤ by GitHub

We are defining a function below which is the implementation of the inference process (which we covered in the above section):

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    e_out, e_h, e_c = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1,1))

    # Chose the 'start' word as the first word of the target sequence
    target_seq[0, 0] = target_word_index['start']

    stop_condition = False
```

```
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + [e_out, e_h, e_c])

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_token = reverse_target_word_index[sampled_token_index]

        if(sampled_token!='end'):
            decoded_sentence += ' '+sampled_token

            # Exit condition: either hit max length or find stop word.
            if (sampled_token == 'end' or len(decoded_sentence.split()) >= (max_len_summary-1)):
                stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1,1))
        target_seq[0, 0] = sampled_token_index

        # Update internal states
        e_h, e_c = h, c

    return decoded_sentence
```

view raw

inferenceprocess.py hosted with ❤ by GitHub

Let us define the functions to convert an integer sequence to a word sequence for summary as well as the reviews:

```
def seq2summary(input_seq):
    newString=''
    for i in input_seq:
      if((i!=0 and i!=target_word_index['start']) and i!=target_word_index['end']):
        newString=newString+reverse_target_word_index[i]+' '
    return newString

def seq2text(input_seq):
    newString=''
    for i in input_seq:
      if(i!=0):
        newString=newString+reverse_source_word_index[i]+' '
    return newString
```

view raw

int2text.py hosted with ❤ by GitHub

```
for i in range(len(x_val)):
  print("Review:",seq2text(x_val[i]))
  print("Original summary:",seq2summary(y_val[i]))
  print("Predicted summary:",decode_sequence(x_val[i].reshape(1,max_len_text)))
  print("\n")
```

view raw

predictions.py hosted with ❤ by GitHub

Here are a few summaries generated by the model:

```
Review: used eating flaxseed brownie hodgson mill brownies super easy make taste great since like dark chocolate usually add littl
e cocoa
Original summary: delicious brownie
Predicted summary:  best brownie mix
```

```
Review: favorite coffee keurig coffeemaker convenient get amazon cheaper running around stores trying find lowest price
Original summary: great coffee
Predicted summary:  great coffee
```

```
Review: mallomars pure chocolate cookies delicious tasty chocolate inside equally tasty cream filling inside pour ice cold glass mi
lk sit back try eat whole box one sitting brian fairbanks
Original summary: delicious
Predicted summary:  best chocolate have ever tasted
```

```
Review: organic usually prefer whatever blech cannot stand taste ended giving away going try another bag mention calories either be
ars calories take haribo please
Original summary: taste terrible
Predicted summary:  not that great
```

```
Review: package six boxes forty eight bags per box listed area large tea bags suitable making gallon time tea fact small single use
bags box web page says family size bags nothing family sized single use bags bad advertisement buy read misleading ads carefully ho
pe company business
Original summary: misleading advertisement
Predicted summary:  not as advertised
```

```
Review: red wine tart unpleasant way comes cans two servings per since carbonated either drink whole extended period save hope flat
share drink fairly quickly like normal soda get lot caffeine sugar pretty short time drinks like come smaller cans good perk right
point give jitters like drinks tend drank full two servings make heart anything drink several cups coffee day occasionally drink en
ergy drinks like well despite caffeine intake caffeinated soda like diet coke still keep night notably drink keep
Original summary: not bad has some ups and downs
Predicted summary:  not as good as it is
```

This is really cool stuff. Even though the actual summary and the summary generated by our model do not match in terms of words, both of them are conveying the same meaning. Our model is able to generate a legible summary based on the context present in the text.

This is how we can perform text summarization using deep learning concepts in Python.

How can we Improve the Model's Performance Even Further?

Your learning doesn't stop here! There's a lot more you can do to play around and experiment with the model:

- I recommend you to **increase the training dataset size** and build the model. The generalization capability of a deep learning model enhances with an increase in the training dataset size
- Try **implementing Bi-Directional LSTM** which is capable of capturing the context from both the directions and results in a better context vector
- Use the **beam search strategy** for decoding the test sequence instead of using the greedy approach (argmax)
- **Evaluate** the performance of your model based on the **BLEU score**
- **Implement pointer-generator networks** and coverage mechanisms

How does the Attention Mechanism Work?

Now, let's talk about the inner workings of the attention mechanism. As I mentioned at the start of the article, this is a math-heavy section so consider this as optional learning. I still highly recommend reading through this to truly grasp how attention mechanism works.

- The encoder outputs the hidden state ($\mathbf{h_j}$) for every time step **j** in the source sequence
- Similarly, the decoder outputs the hidden state ($\mathbf{s_i}$) for every time step **i** in the target sequence
- We compute a score known as an **alignment score ($\mathbf{e_{ij}}$)** based on which the source word is aligned with the target word using a score function. The alignment score is computed from the source hidden state $\mathbf{h_j}$ and target hidden state $\mathbf{s_i}$ using the score function. This is given by:

$$\mathbf{e_{ij} = score\ (s_i, h_j)}$$

where $\mathbf{e_{ij}}$ denotes the alignment score for the target timestep **i** and source time step **j**.

There are different types of attention mechanisms depending on the type of score function used. I've mentioned a few popular attention mechanisms below:

Name	Score function
Dot-Product	$score(s_i,h_j) = s_i^T h_j$
Additive	$score(s_i,h_j) = v_a^T tanh(w_a[s_i;h_j])$ where w_a and v_a are the trainable weight matrices
General	$score(s_i,h_j) = s_i^T w_a hj$ where w_a is a trainable weight matrix

- We normalize the alignment scores using softmax function to retrieve the attention weights ($\mathbf{a_{ij}}$):

$$a_{ij} = e^{eij} / \sum_{k=1}^{Tx} e^{eik}$$

- We compute the linear sum of products of the attention weights $\mathbf{a_{ij}}$ and hidden states of the encoder $\mathbf{h_j}$ to produce the attended context vector ($\mathbf{C_i}$):

$$C_i = \sum_{j=1}^{Tx} a_{ij}\ h_j$$

- The attended context vector and the target hidden state of the decoder at timestep **i** are concatenated to produce an attended hidden vector $\mathbf{S_i}$

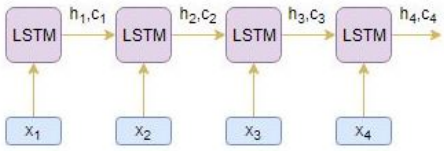
$$S_i = concatenate([s_i; C_i])$$

- The attended hidden vector $\mathbf{S_i}$ is then fed into the dense layer to produce $\mathbf{y_i}$

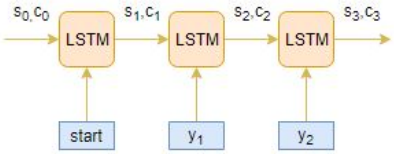
$$y_i = dense(S_i)$$

Let's understand the above attention mechanism steps with the help of an example. Consider the source sequence to be $[x_1, x_2, x_3, x_4]$ and target sequence to be $[y_1, y_2]$.

- The encoder reads the entire source sequence and outputs the hidden state for every timestep, say $\mathbf{h_1, h_2, h_3, h_4}$



- The decoder reads the entire target sequence offset by one timestep and outputs the hidden state for every timestep, say $\mathbf{s_1}$, $\mathbf{s_2}$, $\mathbf{s_3}$



Target timestep i=1

- Alignment scores $\mathbf{e_{1j}}$ are computed from the source hidden state $\mathbf{h_i}$ and target hidden state $\mathbf{s_1}$ using the score function:

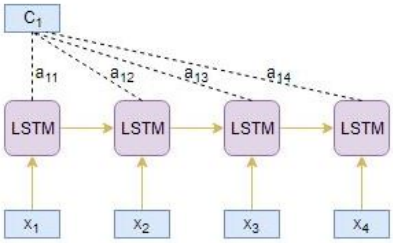
$$e_{11}= score(s_1, h_1)\ e_{12}= score(s_1, h_2)\ e_{13}= score(s_1, h_3)\ e_{14}= score(s_1, h_4)$$

- Normalizing the alignment scores $\mathbf{e_{1j}}$ using softmax produces attention weights $\mathbf{a_{1j}}$:

$$a_{11}= \frac{exp(e_{11})}{(exp(e_{11})+exp(e_{12})+exp(e_{13})+exp(e_{14}))}\ a_{12}= \frac{exp(e_{12})}{(exp(e_{11})+exp(e_{12})+exp(e_{13})+exp(e_{14}))}\ a_{13}= \frac{exp(e_{13})}{(exp(e_{11})+exp(e_{12})+exp(e_{13})+exp(e_{14}))}\ a_{14}= \frac{exp(e_{14})}{(exp(e_{11})+exp(e_{12})+exp(e_{13})+exp(e_{14}))}$$

- Attended context vector $\mathbf{C_1}$ is derived by the linear sum of products of encoder hidden states $\mathbf{h_j}$ and alignment scores $\mathbf{a_{1j}}$:

$$C_1= h_1 * a_{11} + h_2 * a_{12} + h_3 * a_{13} + h_4 * a_{14}$$



- Attended context vector $\mathbf{C_1}$ and target hidden state $\mathbf{s_1}$ are concatenated to produce an attended hidden vector $\mathbf{S_1}$

$$S_1= concatenate([s_1; C_1])$$

- Attentional hidden vector $\mathbf{S_1}$ is then fed into the dense layer to produce $\mathbf{y_1}$

$y_1 = \text{dense}(S_1)$

Target timestep i=2

- Alignment scores $\mathbf{e_{2j}}$ are computed from the source hidden state $\mathbf{h_i}$ and target hidden state $\mathbf{s_2}$ using the score function given by

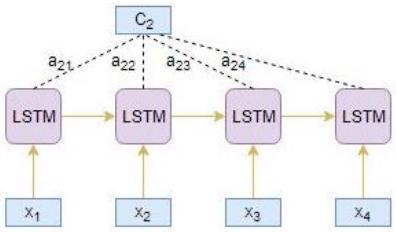
$e_{21} = \text{score}(s_2, h_1)$ $e_{22} = \text{score}(s_2, h_2)$ $e_{23} = \text{score}(s_2, h_3)$ $e_{24} = \text{score}(s_2, h_4)$

- Normalizing the alignment scores $\mathbf{e_{2j}}$ using softmax produces attention weights $\mathbf{a_{2j}}$:

$a_{21} = \exp(e_{21})/(\exp(e_{21})+\exp(e_{22})+\exp(e_{23})+\exp(e_{24}))$ $a_{22} = \exp(e_{22})/(\exp(e_{21})+\exp(e_{22})+\exp(e_{23})+\exp(e_{24}))$ $a_{23} = \exp(e_{23})/(\exp(e_{21})+\exp(e_{22})+\exp(e_{23})+\exp(e_{24}))$ $a_{24} = \exp(e_{24})/(\exp(e_{21})+\exp(e_{22})+\exp(e_{23})+\exp(e_{24}))$

- Attended context vector $\mathbf{C_2}$ is derived by the linear sum of products of encoder hidden states $\mathbf{h_i}$ and alignment scores a_{2j}:

$C_2 = h_1 * a_{21} + h_2 * a_{22} + h_3 * a_{23} + h_4 * a_{24}$



- Attended context vector $\mathbf{C_2}$ and target hidden state $\mathbf{s_2}$ are concatenated to produce an attended hidden vector $\mathbf{S_2}$

$S_2 = \text{concatenate}([s_2; C_2])$

- Attended hidden vector $\mathbf{S_2}$ is then fed into the dense layer to produce $\mathbf{y_2}$

$y_2 = \text{dense}(S_2)$

We can perform similar steps for target timestep i=3 to produce $\mathbf{y_3}$.

I know this was a heavy dosage of math and theory but understanding this will now help you to grasp the underlying idea behind attention mechanism. This has spawned so many recent developments in NLP and now you are ready to make your own mark!

Code

Find the entire notebook here.

End Notes

Take a deep breath – we've covered a lot of ground in this article. And congratulations on building your first text summarization model using deep learning! We have seen how to build our own text summarizer using Seq2Seq modeling in Python.

If you have any feedback on this article or any doubts/queries, kindly share them in the comments section below and I will get back to you. And make sure you experiment with the model we built here and share your results with the community!

You can also take the below courses to learn or brush up your NLP skills:

- Natural Language Processing (NLP) using Python
- Introduction to Natural Language Processing (NLP)

Article Url - https://www.analyticsvidhya.com/blog/2019/06/comprehensive-guide-text-summarization-using-deep-learning-python/

Aravind Pai

Aravind is a sports fanatic. His passion lies in developing data-driven products for the sports domain. He strongly believes that analytics in sports can be a game-changer