

**A Project Report**

**on**

# **Heading Generator using Machine Learning**

**Submitted in partial fulfilment of the  
requirement for the award of the degree of**

**Bachelor of Technology in CSE**



(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

**Under The Supervision of  
Dr. T. Ganesh Kumar**

**Associate Professor**

**Department of Computer Science and Engineering**

**Submitted By**

**Rajat Upadhyay  
18SCSE1010176**

**Harshit Bhati  
19SCSE1010883**

**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING  
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
GALGOTIAS UNIVERSITY, GREATER NOIDA  
INDIA MAY 2022**



**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING  
GALGOTIAS UNIVERSITY, GREATER NOIDA**

**CANDIDATE'S DECLARATION**

We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled "Heading Generator using Machine Learning" in partial fulfilment of the requirements for the award of the Bachelor of Computer Science and Engineering submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of Jan,2022 to May, 2022, under the supervision of Dr. T. Ganesh Kumar Associate Professor, Department of Computer Science and Engineering of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the thesis/project/dissertation has not been submitted by us for the award of any other degree of this or any other places.

Rajat Upadhyay

Harshit Bhati

18SCSE1010176

19SCSE1010883

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Dr. T. Ganesh Kumar

Associate Professor

**CERTIFICATE**

The Final Thesis/Project/ Dissertation Viva-Voce examination of Rajat Upadhyay and Harshit Bhati has been held on \_\_\_\_\_ and his/her work is recommended for the award of Bachelor of Computer Science and Engineering

**Signature of Examiner(s)**

**Signature of Supervisor(s)**

**Signature of Project Coordinator**

**Signature of Dean**

Date:

Place: Greater Noida

## **Abstract**

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy. Supervised learning is the type of machine learning in which machines are trained using well "labelled" training data, and on basis of that data, machines predict the output. We will be focusing on the following model in particular: Heading Generator using Machine Learning using the YouTube trending videos dataset and the Python programming language to train a model of text generation language using machine learning, which will be used for the task of Heading generator for youtube videos or even for your blogs. Heading generator is a natural language processing task and is a central issue for several machine learning, including text synthesis, speech to text, and conversational systems. To build a model for the task of Heading generator or a text generator, the model should be able to learn the probability of a word occurring, using words that have already appeared in the sequence as context. Headline or short summary generation is an important problem in Text Summarization and has several practical applications. We present a discriminative learning framework and a rich feature set for the headline generation task. Secondly, we present a novel Bleu measure based scheme for evaluation of headline generation models, which does not require human produced references. We achieve this by building a test corpus using the Google news service. We propose two stacked log-linear models for both headline word selection (Content Selection) and for ordering words into a grammatical and coherent headline.

**Tools used – Keras ,Tensor Flow ,Python ,Machine learning libraries**

## List of Tables

**Table  
No.**

**Table Name**

**Page  
Number**

## List of Figures

**Figure  
No.**

**Table Name**

**Page  
Number**

## **Acronyms**

<b>B.Tech.</b>	<b>Bachelor of Technology</b>
<b>M.Tech.</b>	<b>Master of Technology</b>
<b>BCA</b>	<b>Bachelor of Computer Applications</b>
<b>MCA</b>	<b>Master of Computer Applications</b>
<b>B.Sc. (CS)</b>	<b>Bachelor of Science in Computer Science</b>
<b>M.Sc. (CS)</b>	<b>Master of Science in Computer Science</b>
<b>SCSE</b>	<b>School of Computing Science and Engineering</b>

## Table of Contents

<b>Title</b>	<b>Page No.</b>
<b>Abstract</b>	<b>I</b>
<b>List of Table</b>	<b>II</b>
<b>List of Figures</b>	<b>III</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
<b>1.1 Introduction</b>	<b>2</b>
<b>1.2 Formulation of Problem</b>	<b>3</b>
<b>1.2.1 Tool and Technology Used</b>	
<b>Chapter 2 Literature Survey/Project Design</b>	<b>5</b>

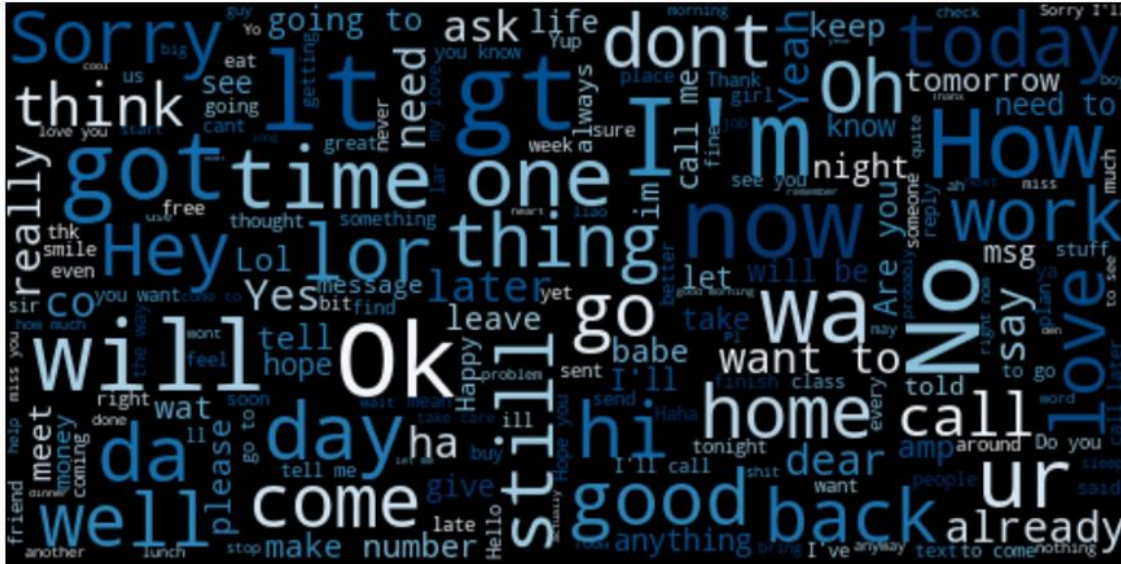


## **CHAPTER-1**

### **Introduction**

The Heading generator is a function of Natural Language Processing and is a subject between several Machine Learning, including text compilation, text speaking, and discussion programs.

To create a Heading-generating work model or a text generator, the model must be trained to learn whether a word may occur, using words that already appear in sequence as context. Text summarization has become a driving application of any information or content management system. The explosive growth of worldwideweb which has mostly unstructured information and online information services has resulted in an information overload problem. Corporations struggle to manage the immense amount of textual information they produce on a day to day basis. It is not surprising that vast amounts of effort and budget have been devoted both in industry and research towards building automated text summarization systems. Within text summarization the focus has almost universally been on extractive techniques i.e. selecting text spans - either complete sentences or paragraphs from the input text. A major pitfall of the extractive summarization techniques is that they cannot generate effective headline styled summaries less than a single .A special application of text summarization is generating very short summarizes from input text, or headlines from news articles and documents, and is the focus of this work. Headline or head- line styled summaries are distinctively different than abstracts of documents. Headlines are terse and convey the singular, most important theme of the input text while abstracts use relatively more words and reflect many important points in the input text.



## Natural Language Processing

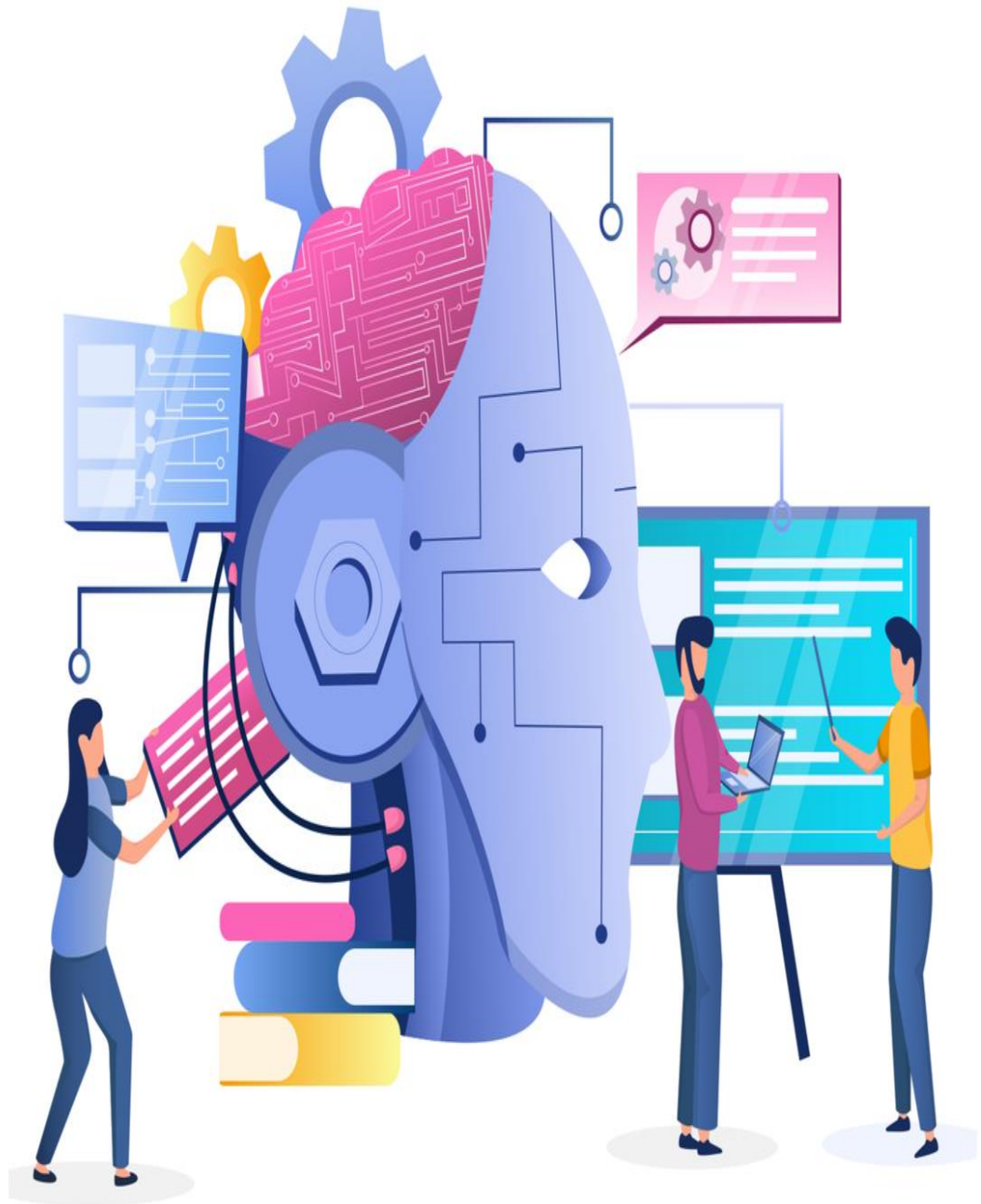
NLP is a branch of data science that consists of systematic processes for analyzing, understanding, and deriving information from the text data in a smart and efficient manner. By utilizing NLP and its components, one can organize the massive chunks of text data, perform numerous automated tasks and solve a wide range of problems such as – automatic summarization, machine translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation etc.

Before moving further, I would like to explain some terms that are used in the article:

- Tokenization – process of converting a text into tokens
- Tokens – words or entities present in the text
- Text object – a sentence or a phrase or a word or an article

**Natural Language Processing (NLP)** is often used for textual segregation activities such as spam detection and emotional analysis, text production, language translation, and text classification. Text data can be viewed in alphabetical order, word order, or sentence sequence. In general, text data is considered a sequence of words in most problems. Here we will enter, a process using simple sample data. However, the steps discussed here apply to any NLP activities. In particular, we will use **TensorFlow2, Keras** to obtain text processing which includes:

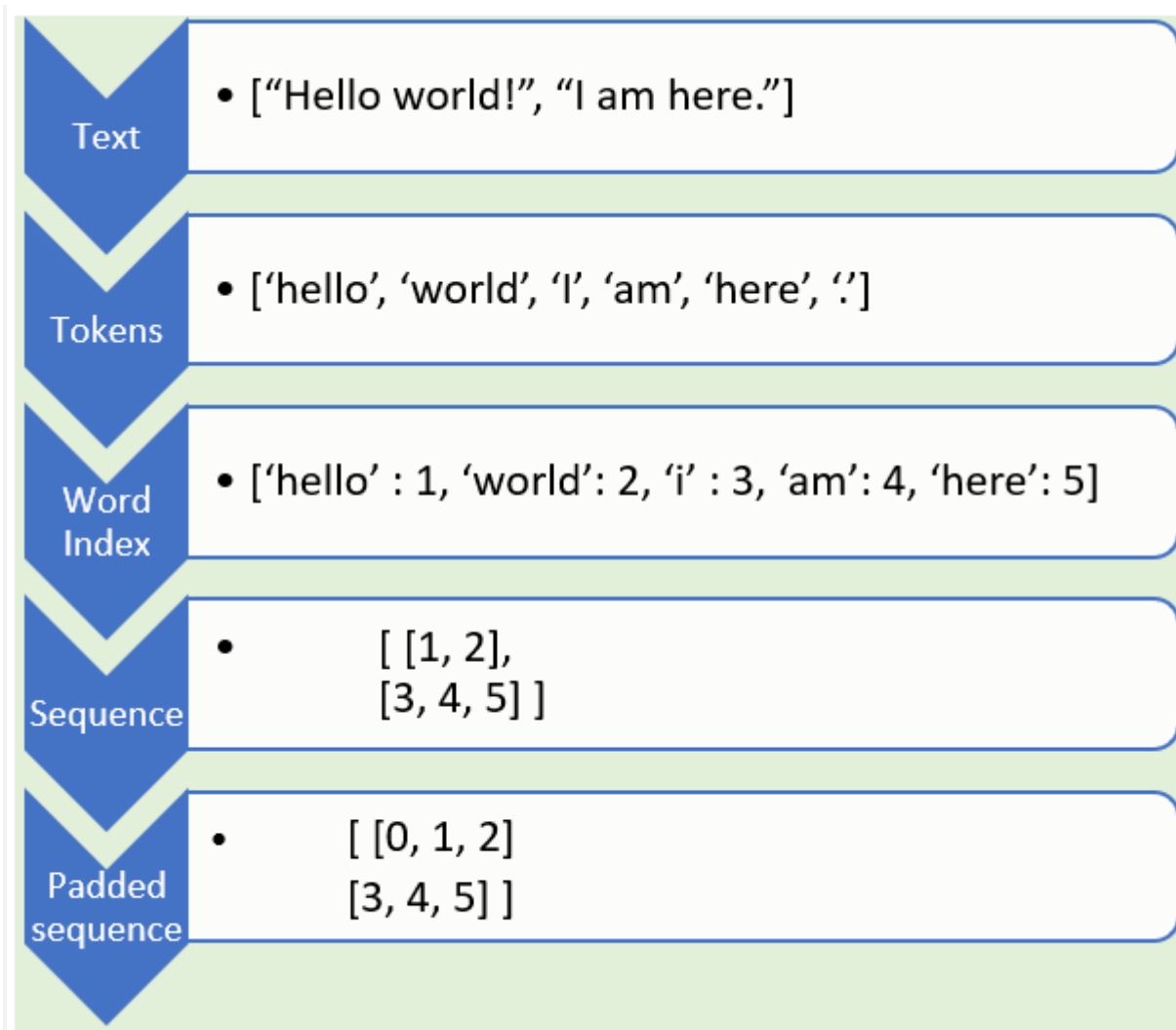
- Tokenization
- Sequence
- Padding



Natural Language Processing (NLP) is commonly used in text classification tasks such as spam detection and sentiment analysis, text generation, language translations and document classification. Text data can be considered either in sequence of character, sequence of words or sequence of sentences. Most commonly, text data are considered as sequence of words for most problems. In this article we will delve into, pre-processing using simple example text data. However, the steps discussed here are applicable to any NLP tasks. Particularly, we'll use TensorFlow2 Keras for text pre-processing which include:

- Tokenization
- Sequencing
- Padding

The figure below depicts the process of text pre-processing along with example outputs.



Step by step text pre-processing example starting from raw sentence to padded sequence

First, let's import the required libraries.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

*Tokenizer* is an API available in TensorFlow Keras which is used to tokenize sentences. We have defined our text data as sentences (each separated by a comma) and with an array of strings. There are 4 sentences including 1 with a maximum length of 5. Our text data also includes punctuations as shown below.

```

sentences = ["I want to go out.",
             " I like to play.",
             " No eating - ",
             "No play!",
             ]
sentences["I want to go out.", ' I like to play.', ' No eating - ', 'No play!']

```

## Tokenization

As deep learning models do not understand text, we need to convert text into numerical representation. For this purpose, a first step is Tokenization. The *Tokenizer* API from TensorFlow Keras splits sentences into words and encodes these into integers. Below are hyperparameters used within *Tokenizer* API:

- `num_words`: Limits maximum number of most popular words to keep while training.
- `filters`: If not provided, by default filters out all punctuation terms (!"#\$%&()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n).
- `lower=1`. This is a default setting which converts all words to lower case
- `oov_tok` : When its used, out of vocabulary token will be added to word index in the corpus which is used to build the model. This is used to replace out of vocabulary words (words that are not in our corpus) during *text\_to\_sequence* calls (see below).
- `word_index`: Convert all words to integer index. Full list of words are available as key value property: key = word and value = token for that word

Let's use the *Tokenizer* and print out word index. We have used `num_words= 100` which is a lot for this data as there are only 9 distinct words and `<OOV>` string for out of vocabulary token.

```

tokenizer = Tokenizer(num_words=100, lower= 1, oov_token="<OOV>")
tokenizer.fit_on_texts(sentences)

```

```
word_index = tokenizer.word_index
print(word_index)
{'<OOV>': 1, 'i': 2, 'to': 3, 'play': 4, 'no': 5, 'want': 6, 'go': 7, 'out': 8, 'like': 9, 'eating': 10}
```

As seen above, each word in our sentences has been converted to numerical tokens. For instance, “i” has a value of 2. The tokenizer also ignored the exclamation mark after the word. For example, there is only one token for the word “play” or “play!” i.e. 4.

## Sequencing

Next, let’s represent each sentence by sequences of numbers using *texts\_to\_sequences* from tokenizer object. Below, we printed out raw sentences, word index and sequences.

```
sequences = tokenizer.texts_to_sequences(sentences)
print(sentences)
print(word_index)
print(sequences)
['I want to go out', 'I like to play', 'No eating - ', 'No play!']
{'<OOV>': 1, 'i': 2, 'to': 3, 'play': 4, 'no': 5, 'want': 6, 'go': 7, 'out': 8, 'like': 9, 'eating': 10}
[[2, 6, 3, 7, 8], [2, 9, 3, 4], [5, 10], [5, 4]]
```

As shown above, texts are represented by sequences. For example,

- “I want to go out” — -> [2, 6, 3, 7, 8]
- “I like to play” — -> [2, 9, 3, 4]
- “No eating” — -> [5, 10]
- “No play!” — -> [5, 4]

## Padding

In any raw text data, naturally there will be sentences of different lengths. However, all neural networks require to have inputs with the same size. For this purpose, padding is done. Below, let’s

use `pad_sequences` for padding. `pad_sequences` uses arguments such as `sequences`, `padding`, `maxlen`, `truncating`, `value` and `dtype`.

- `sequences`: list of sequences that we created earlier
- `padding` = 'pre' or 'post' (default pre). By using pre, we'll pad (add 0) before each sequence and post will pad after each sequence.
- `maxlen` = maximum length of all sequences. If not provided, by default it will use the maximum length of the longest sentence.
- `truncating` = 'pre' or 'post' (default 'pre'). If a sequence length is larger than the provided `maxlen` value then, these values will be truncated to the `maxlen`. 'pre' option will truncate at the beginning whereas 'post' will truncate at the end of the sequences.
- `value`: padding value (default is 0)
- `dtype`: output sequence type (default is int32)

Let's focus on important arguments used in `pad_sequences` : `padding`, `maxlen` and `truncating`.

### **pre and post padding**

Use of 'pre' or 'post' padding depends upon the analysis. In some cases, padding at the beginning is appropriate while not in others. For instance, if we use Recurrent Neural Network (RNN) for spam detection, then padding at the beginning would be appropriate as RNN can not learn long distance patterns. Padding at the beginning allows us to keep the sequences in the end hence RNN can make use of these sequences for prediction of next. However, in any case padding should be done after careful consideration and business knowledge.



Below, the outputs for ‘pre’ followed by ‘post’ padding are shown with default maxlen value of maximum length of sequence.

```
# pre padding
pre_pad = pad_sequences(sequences, padding='pre')
print("\nword_index = ", word_index)
print("\nsequences = ", sequences)
print("\npadded_seq = ")
print(pre_pad)
word_index = {'<OOV>': 1, 'i': 2, 'to': 3, 'play': 4, 'no': 5, 'want': 6, 'go': 7, 'out': 8, 'like': 9, 'eating': 10}

sequences = [[2, 6, 3, 7, 8], [2, 9, 3, 4], [5, 10], [5, 4]]

padded_seq =
[[ 2 6 3 7 8]
 [ 0 2 9 3 4] <----- 0 Padded at the beginning
 [ 0 0 0 5 10]
 [ 0 0 0 5 4]]
```

In our example above, the sequence with maximum length is [ 2, 6, 3, 7, 8] which corresponds to “I want to go out”. When padding = ‘pre’ is used, padded value of 0 is added at the beginning of all other sequences. Because other sequences have shorter sequence than [ 2, 6, 3, 7, 8], padding actually made all other sequences to be of same size with this sequence.

Whereas, when padding = ‘post’ is used , padded value i.e. 0 is added at the end of the sequences.

```
# post padding
post_pad = pad_sequences(sequences, padding='post')
print("\nword_index = ", word_index)
print("\nsequences = ", sequences)
print("\npadded_seq = ")
print(post_pad)
word_index = {'<OOV>': 1, 'i': 2, 'to': 3, 'play': 4, 'no': 5, 'want': 6, 'go': 7, 'out': 8, 'like': 9, 'eating': 10}

sequences = [[2, 6, 3, 7, 8], [2, 9, 3, 4], [5, 10], [5, 4]]

padded_seq =
[[ 2 6 3 7 8]
 [ 2 9 3 4 0] <----- 0 Padded at the end
 [ 5 10 0 0 0]
 [ 5 4 0 0 0]]
```

## pre and post padding with maxlen and truncating option

We can use both padding and truncating argument together if needed. Below we have shown two-scenarios, 1) pre padding with pre truncation and 2) pre padding with post truncation

The truncating with 'pre' option allows us to truncate the sequence at the beginning. Whereas, truncating with 'post' will truncate the sequence at the end.

Let's look at the example of pre padding with pre truncation.

```
# pre padding, maxlen and pre truncation
prepad_maxlen_pretrunc = pad_sequences(sequences, padding = 'pre', maxlen =4, truncating = 'pre')
print(prepad_maxlen_pretrunc)[[ 6 3 7 8]<-----Truncated from length 5 to 4, at the beginning
[ 2 9 3 4]
[ 0 0 5 10]<----- Padded at the beginning
[ 0 0 5 4]]
```

By use of maxlen =4, we are truncating the length of padded sequences to 4. As shown, above, the use of maxlen=4 impacted the first sequence [2, 6, 3, 7, 8]. This sequence had length of 5 and is truncated to 4. The truncation happened at the beginning as we used truncating = 'pre' option.

Let's look at the truncation = 'post' option.

```
# pre padding, maxlen and post truncation
prepad_maxlen_posttrunc = pad_sequences(sequences, padding = 'pre', maxlen =4, truncating = 'post')
print(prepad_maxlen_posttrunc)[[ 2 6 3 7]<-----Truncated from length 5 to 4, at the end
[ 2 9 3 4]
[ 0 0 5 10]<----- Padded at the beginning
[ 0 0 5 4]]
```

The truncation happened at the end as we used truncating = 'post' option. When the post truncation was applied, it impacted the first sequence [ 2, 6, 3, 7, 8] and truncated to length 4 resulting in the sequence [ 2, 6, 3, 7].

## CHAPTER-2

### Literature Survey

As mentioned, often the application at hand requires generation of headline styled summaries from text. Such summaries are typically not more than 10-15 words in length. The headline of a text, especially a news article is a compact, grammatical and coherent representation of important pieces of information in the news article. Headlines help readers to quickly identify information that is of interest to them. Although newspaper articles are usually accompanied by headlines, there are numerous other types of news text sources, such as transcripts of radio and television broadcasts and machine translated texts where such summary information is missing.

A system that can automatically generate headline styled summaries can be useful in the following potential applications.

- Summarizing emails, web pages for portable wireless devices, WAP enabled mobile phones and PDAs which have limited display and bandwidth.
- Generating a table of contents styled summary for machine generated texts or machine translated documents.
- To present compressed descriptions of search result web pages in search engines

Headlines extracted from search result web pages can be used to augment a user search query. The resultant query can be used to further re-rank and improve upon the search results.

The advantage of Summarization approaches is that it alleviates the need to treat headline generation as a special problem and one can simply take an existing text summarization system and request it to generate highly compressed summaries as headlines. But the problem with resorting to summarization approaches for headline generation is that, for summarization systems when the compression rate falls below 10%, the quality of generated summaries is poor. Since headlines are typically no more than 10-15 words, the compression ratio is in fact far less than 10% for many news articles. This would mean that text summarization methods will create poor headlines. Another problem with summarization approaches is that most of the

techniques we discussed above are extractive in nature which constrains their use in headline generation in other ways. For example: approaches that treat a full sentence as the minimum unit for a summary may result in longer than required headlines. Another problem is that extractive techniques would pick only the phrases and words present in the article for inclusion in the headline. But often we see that headlines do not borrow the exact same words as present in the news article. The example below makes the point clear where the words *attacks* and *fighters* are not present in the article but are used to refer to *the act of striking* and *insurgents* respectively. Given we train our model on sufficiently large corpora we can learn the *attacks* is a good substitution for words like *struck* and that *insurgents* can also be referred to as *fighters*.

Finally, another scenario where extractive summarization approaches are not suitable

is cross-lingual headline generation in which news articles are present in one language and headlines need to be generated in a different language. But statistical or corpus based techniques can be used without any specific changes for cross-lingual headline generation just as they are used in the routine scenario. Cross-lingual headline generation can indeed be very useful in cases where say a native language A (English) speaker is looking for language B (French) news articles on a specific topic or event.

## STEPS TO FOLLOW

### **Importing the necessary libraries** Building the Machine Learning Model for Heading Generation

Importing libraries before we start working on them. Here using **Keras** and **TensorFlow** as the main libraries for our model as it is a highly productive interface for solving such problems, with a deep learning approach.

Now process the data so that we can use this data to train our machine learning model with the task of making a topic.

## Generating sequences for Building the Machine Learning Model for Heading Generation

Natural language processing operations require data entry in the form of a token sequence. The first step after data purification is to generate a sequence of n-gram tokens.

N-gram is the closest sequence of n elements of a given sample of text or vocal corpus. Items can be words, letters, phonemes, letters, or base pairs. In this case, n-gr is a sequence of words in the corpus of titles.

The tokenizer is an API found in TensorFlow Keras that is used to make sentences into a token. We defined our text data as sentences (each with a comma) and with multiple strings.

Since in-depth reading models do not understand the text, we need to convert the text into a numerical representation. For this purpose, the first step is to make tokens. The Tokenizer API from TensorFlow Keras divides sentences into words and converts these into numbers. Tokenization is the process of issuing tokens from a corpus:

1. A Quick Rundown of Tokenization
2. The True Reasons behind Tokenization
3. Which Tokenization (Word, Character, or Subword) Should we Use?
4. Implementing Tokenization– Byte Pair Encoding in Python

### *A Quick Rundown of Tokenization*

Tokenization is a common task in Natural Language Processing (NLP). It's a fundamental step in both traditional NLP methods like Count Vectorizer and Advanced Deep Learning-based architectures like [Transformers](#).

*Tokens are the building blocks of Natural Language.*

Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization.

For example, consider the sentence: “Never give up”.

The most common way of forming tokens is based on space. Assuming space as a delimiter, the tokenization of the sentence results in 3 tokens – Never-give-up. As each token is a word, it becomes an example of Word tokenization.

Similarly, tokens can be either characters or subwords. For example, let us consider “smarter”:

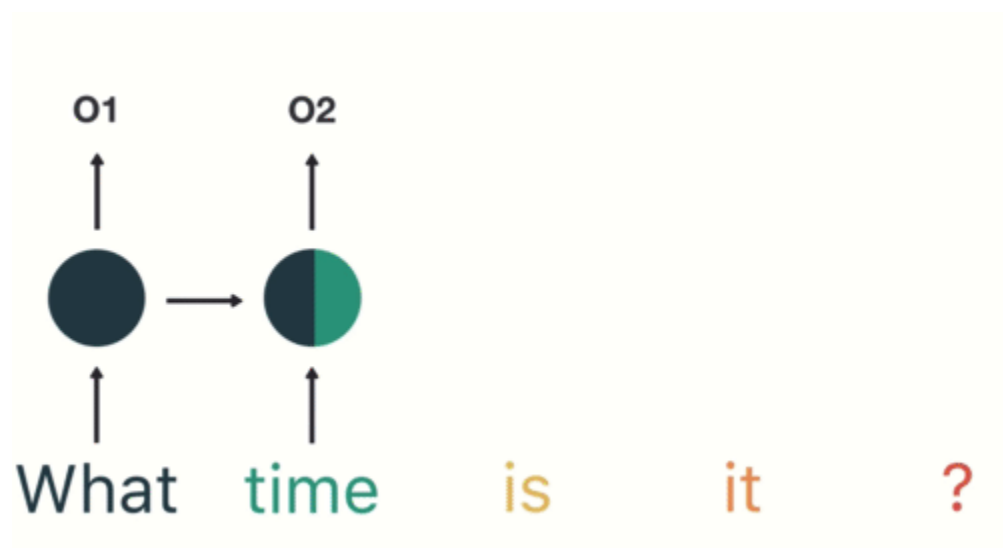
1. Character tokens: s-m-a-r-t-e-r
2. Subword tokens: smart-er

But then is this necessary? Do we really need tokenization to do all of this?

### *The True Reasons behind Tokenization*

As tokens are the building blocks of Natural Language, the most common way of processing the raw text happens at the token level.

For example, Transformer based models – the State of The Art (SOTA) Deep Learning architectures in NLP – process the raw text at the token level. Similarly, the most popular deep learning architectures for NLP like RNN, GRU, and LSTM also process the raw text at the token level.



Working of Recurrent Neural Network

As shown here, RNN receives and processes each token at a particular timestep.

Hence, Tokenization is the foremost step while modeling text data. Tokenization is performed on the corpus to obtain tokens. The following tokens are then used to prepare a vocabulary.

Vocabulary refers to the set of unique tokens in the corpus. Remember that vocabulary can be constructed by considering each unique token in the corpus or by considering the top K Frequently Occurring Words.

*Creating Vocabulary is the ultimate goal of Tokenization.*

*One of the simplest hacks to boost the performance of the NLP model is to create a vocabulary out of top K frequently occurring words.*

Now, let's understand the usage of the vocabulary in Traditional and Advanced Deep Learning-based NLP methods.

- Traditional NLP approaches such as Count Vectorizer and TF-IDF use vocabulary as features. Each word in the vocabulary is treated as a unique feature:

	I	play	cricket	football	tennis
Doc 1	1	1	1	1	1
Doc 2	1	1	0	1	0
Doc 3	0	1	1	0	0
Doc 4	1	1	0	0	1

*Traditional NLP: Count Vectorizer*

- In Advanced Deep Learning-based NLP architectures, vocabulary is used to create the tokenized input sentences. Finally, the tokens of these sentences are passed as inputs to the model

## *Which Tokenization Should you use?*

As discussed earlier, tokenization can be performed on word, character, or subword level. It's a common question – which Tokenization should we use while solving an NLP task? Let's address this question here.

### Word Tokenization

Word Tokenization is the most commonly used tokenization algorithm. It splits a piece of text into individual words based on a certain delimiter. Depending upon delimiters, different word-level tokens are formed. [Pretrained Word Embeddings](#) such as Word2Vec and GloVe comes under word tokenization.

But, there are few drawbacks to this.

### **Drawbacks of Word Tokenization**

One of the major issues with word tokens is dealing with **Out Of Vocabulary (OOV) words**. OOV words refer to the new words which are encountered at testing. These new words do not exist in the vocabulary. Hence, these methods fail in handling OOV words.

But wait – don't jump to any conclusions yet!

- A small trick can rescue word tokenizers from OOV words. The trick is to form the vocabulary with the Top K Frequent Words and replace the rare words in training data with **unknown tokens (UNK)**. This helps the model to learn the representation of OOV words in terms of UNK tokens



- So, during test time, any word that is not present in the vocabulary will be mapped to a UNK token. This is how we can tackle the problem of OOV in word tokenizers.
- The problem with this approach is that the entire information of the word is lost as we are mapping OOV to UNK tokens. The structure of the word might be helpful in representing the word accurately. And another issue is that every OOV word gets the same representation

**HOLD ON**



**LET ME PROCESS THIS**

Another issue with word tokens is connected to the size of the vocabulary. Generally, pre-trained models are trained on a large volume of the text corpus. So, just imagine building the vocabulary with all the unique words in such a large corpus. This explodes the vocabulary!

This opens the door to Character Tokenization.

### Character Tokenization

Character Tokenization splits a piece of text into a set of characters. It overcomes the drawbacks we saw above about Word Tokenization.

- Character Tokenizers handles OOV words coherently by preserving the information of the word. It breaks down the OOV word into characters and represents the word in terms of these characters
- It also limits the size of the vocabulary. Want to talk a guess on the size of the vocabulary? 26 since the vocabulary contains a unique set of characters

## **Drawbacks of Character Tokenization**

Character tokens solve the OOV problem but the length of the input and output sentences increases rapidly as we are representing a sentence as a sequence of characters. As a result, it becomes challenging to learn the relationship between the characters to form meaningful words.

This brings us to another tokenization known as Subword Tokenization which is in between a Word and Character tokenization.

### *Subword Tokenization*

Subword Tokenization splits the piece of text into subwords (or n-gram characters). For example, words like lower can be segmented as low-er, smartest as smart-est, and so on.

Transformed based models – the SOTA in NLP – rely on Subword Tokenization algorithms for preparing vocabulary. Now, I will discuss one of the most popular Subword Tokenization algorithm known as Byte Pair Encoding (BPE).

Welcome to Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a widely used tokenization method among transformer-based models. BPE addresses the issues of Word and Character Tokenizers:

- BPE tackles OOV effectively. It segments OOV as subwords and represents the word in terms of these subwords
- The length of input and output sentences after BPE are shorter compared to character tokenization

BPE is a word segmentation algorithm that merges the most frequently occurring character or character sequences iteratively. Here is a step by step guide to learn BPE.

## Steps to learn BPE

1. Split the words in the corpus into characters after appending </w>
2. Initialize the vocabulary with unique characters in the corpus
3. Compute the frequency of a pair of characters or character sequences in corpus
4. Merge the most frequent pair in corpus
5. Save the best pair to the vocabulary
6. Repeat steps 3 to 5 for a certain number of iterations

We will understand the steps with an example.

low	lower	newest
low	lower	newest
low	widest	newest
low	widest	newest
low	widest	newest

Consider a corpus:

1a) Append the end of the word (say `</w>`) symbol to every word in the corpus:

low</w>	lower</w>	newest</w>
low</w>	lower</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>
low</w>	widest</w>	newest</w>

1b) Tokenize words in a corpus into characters:

l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>

2. Initialize the vocabulary:

d	e	i	l	n	o	s	t	w

**Iteration 1:**

3. Compute frequency:

Frequency

d-e (3)	l-o (7)	t-</w> (8)
e-r (2)	n-e (5)	w-</w> (5)
<b>e-s (8)</b>	o-w (7)	w-e (7)
e-w (5)	r-</w> (2)	w-i (3)
i-d (3)	s-t (8)	

4. Merge the most frequent pair:

Corpus

l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>

5. Save the best pair:

Vocabulary

d	e	i	l	n	o	s	t	w
<b>es</b>								

Repeat steps 3-5 for every iteration from now. Let me illustrate for one more iteration.

**Iteration 2:**

3. Compute frequency:

Frequency

d-es (3)	l-o (7)	w-</w> (5)
e-r (2)	n-e (5)	w-es (5)
e-w (5)	o-w (7)	w-e (2)
<b>es-t (8)</b>	r-</w> (2)	w-i (3)
i-d (3)	t-</w> (8)	

#### 4. Merge the most frequent pair:

Corpus		
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	l o w e r </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>
l o w </w>	w i d e s t </w>	n e w e s t </w>

#### 5. Save the best pair:

Vocabulary								
d	e	i	l	n	o	s	t	w
e s	e s t							

After 10 iterations, BPE merge operations looks like:

Vocabulary								
d	e	i	l	n	o	s	t	w
e s	e s t	e s t </w>	l o	l o w	l o w </w>	n e	n e w	n e w e s t </w>

Pretty straightforward, right?

## Applying BPE to OOV words

But, how can we represent the OOV word at test time using BPE learned operations? Any ideas? Let's answer this question now.

At test time, the OOV word is split into sequences of characters. Then the learned operations are applied to merge the characters into larger known symbols.

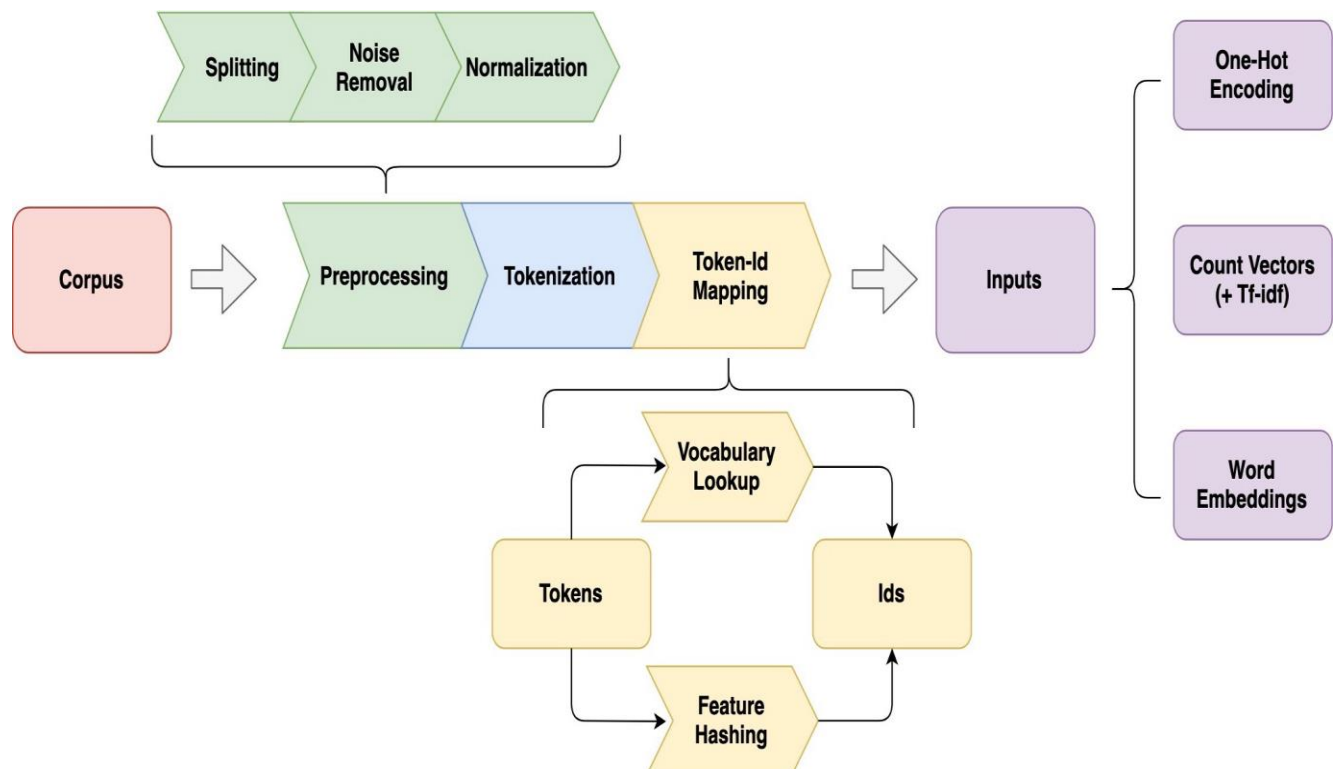
– Neural Machine Translation of Rare Words with Subword Units, 2016

Here is a step by step procedure for representing OOV words:

1. Split the OOV word into characters after appending `</w>`
2. Compute pair of character or character sequences in a word
3. Select the pairs present in the learned operations
4. Merge the most frequent pair
5. Repeat steps 2 and 3 until merging is possible

### Implementing Tokenization – Byte Pair Encoding in Python

We are now aware of how BPE works – learning and applying to the OOV words.



## Padding the sequences for Building the Machine Learning Model for Title Generation

In any raw text data, there will naturally be sentences of different lengths. However, all neural networks need to be input in the same size. For this purpose, wrapping is done. The use of the 'pre' or 'post' pad depends on the analysis. In some cases, wrapping at first is appropriate while not for others. For example, if we use Recurrent Neural Network (RNN) to detect spam detection, then initial wrapping may be appropriate as RNN can read long-distance patterns. Early wrap allows us to keep track of the end which is why RNN can use these sequences to predict the next. However, any support should be made after careful consideration and business knowledge.

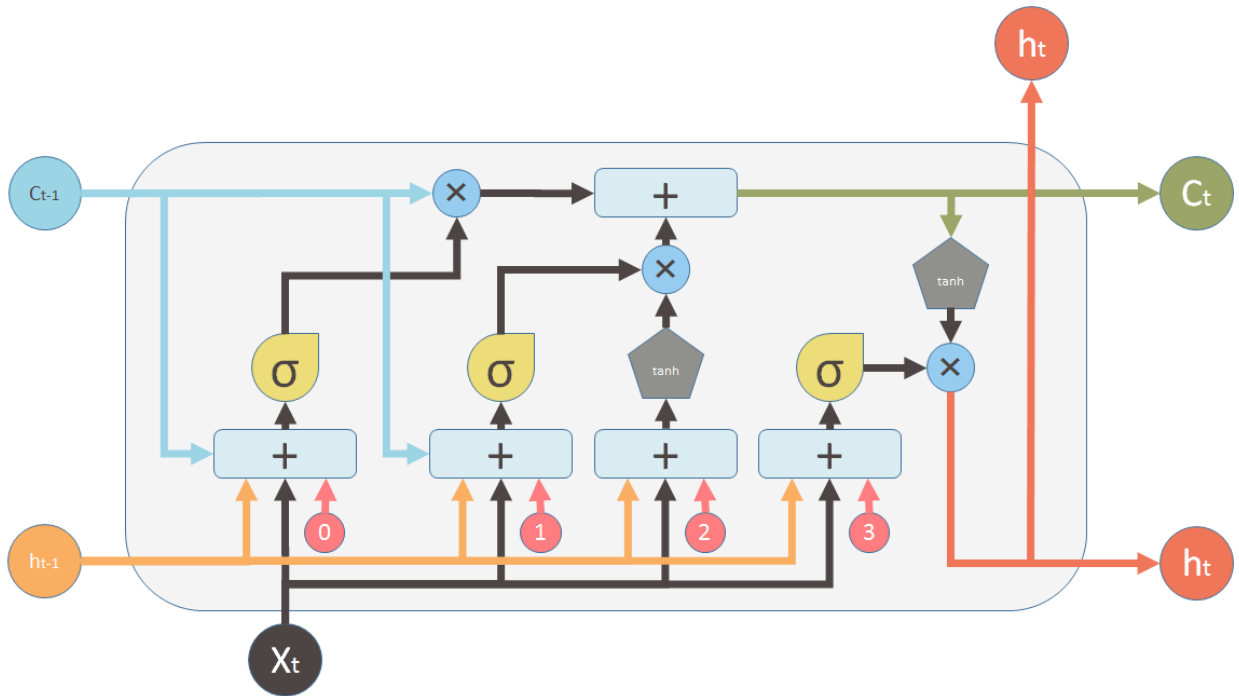
Since sequences can vary in length, the length of the sequence must be proportional. When using neural networks, we usually feed input to the network while waiting for the result. In practice, it is better to process data in batches than to do one at a time. The `pad_sequences()` is a function in the Keras deep learning library that can be used to pad variable-length sequences.

This is done using matrices [batch length x sequence length], where the length of the sequence corresponds to the longest sequence. In this case, we complete the sequence with the symbol (frequency 0) to match the size of the matrix. This process of filling the token sequence is called filling. To enter data from the training model, I need to create predictions and labels.




I will build an n-gram sequence as a prediction and the following n-gram word as a label:





# LSTM Model






**Inputs:**

-  Input vector
-  Memory from previous block
-  Output of previous block


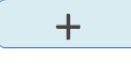
**outputs:**

-  Memory from current block
-  Output of current block

**Nonlinearities:**

-  Sigmoid
-  Hyperbolic tangent
- Bias:** 

**Vector operations:**

-  Element-wise multiplication
-  Element-wise Summation / Concatenation

In recurrent neural networks, the activation outputs are propagated in both directions, i.e. from input to output and

outputs to inputs, unlike direct-acting neural networks where outputs and activation are propagated in only one direction. This creates loops in the architecture of the neural network which acts as a “memory state” of neurons.

As a result, the RNN preserves a state through the stages of time or “remembers” what has been learned over time. The state of memory has its advantages, but it also has its disadvantages. The gradient that disappears is one of them.

In this problem, while learning with a large number of layers, it becomes really difficult for the network to learn and adjust the parameters of the previous layers. To solve this problem, a new type of RNN has been developed; LSTM (long-term memory).

## **Title Generator with LSTM Model**

The LSTM model contains an additional state (the state of the cell) which essentially allows the network to learn what to store in the long term state, what to delete and what to read. . The LSTM of this model contains three layers:

- Input layer: takes the sequence of words as input
- LSTM Layer: Calculates the output using LSTM units.
- Dropout layer: a regularization layer to avoid over fitting
- Output layer: calculates the probability of the next possible word on output.

now use LSTM Model to build a Heading Generator job model with Machine Learning:

### **RESULTS AND OUTPUT:**

Now that model is ready and trained using data, it is time to predict the title based on the input name. The input name is completed first, the sequence is

completed before being transferred to a trained model to retrieve the predicted sequence:

## CODE

```
from pandas.core.arrays import categorical
import pandas as pd
import string
import numpy as np
import json
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Embedding, LSTM, Dense, Dropout
from keras.preprocessing.text import Tokenizer
from keras.callbacks import EarlyStopping
from keras.models import Sequential
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
tf.random.set_seed(2)
from numpy.random import seed
seed(1)
import pandas as pd

#load all the datasets
df1 = pd.read_csv("/content/drive/MyDrive/archive data/USvideos.csv")
df2 = pd.read_csv("/content/drive/MyDrive/archive data/CAvideos.csv")
df3 = pd.read_csv("/content/drive/MyDrive/archive data/GBvideos.csv")

#load the datasets containing the category names
data1 = json.load(open('/content/drive/MyDrive/archive data/US_category_id.json'))
data2 = json.load(open('/content/drive/MyDrive/archive data/CA_category_id.json'))
data3 = json.load(open('/content/drive/MyDrive/archive data/GB_category_id.json'))
def category_extractor(data):
    i_d = [data['items'][i]['id'] for i in range(len(data['items']))]
    title = [data['items'][i]['snippet']['title'] for i in range(len(data['items']))]
    i_d = list(map(int, i_d))
    category = zip(i_d, title)
    category = dict(category)
    return category
```

```

#create a new category column by mapping the category names to their id
df1['category_title'] = df1['category_id'].map(category_extractor(data1))
df2['category_title'] = df2['category_id'].map(category_extractor(data2))
df3['category_title'] = df3['category_id'].map(category_extractor(data3))
#join the dataframes
df = pd.concat([df1, df2, df3], ignore_index=True)

#drop rows based on duplicate videos
df = df.drop_duplicates('video_id')

#collect only titles of entertainment videos
#feel free to use any category of video that you want
entertainment = df[df['category_title'] == 'Entertainment']['title']
entertainment = entertainment.tolist()

#remove punctuations and convert text to lowercase
def clean_text(text):
    text = ''.join(e for e in text if e not in string.punctuation).lower()

    text = text.encode('utf8').decode('ascii', 'ignore')
    return text

corpus = [clean_text(e) for e in entertainment]

tokenizer = Tokenizer()
def get_sequence_of_tokens(corpus):
    #get tokens
    tokenizer.fit_on_texts(corpus)
    total_words = len(tokenizer.word_index) + 1

    #convert to sequence of tokens
    input_sequences = []
    for line in corpus: token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)): n_gram_sequence = token_list[:i+1]
    input_sequences.append(n_gram_sequence)

    return input_sequences, total_words
inp_sequences, total_words = get_sequence_of_tokens(corpus)

def generate_padded_sequences(input_sequences):
    max_sequence_len = max([len(x) for x in input_sequences])
    input_sequences = np.array(pad_sequences(input_sequences, maxlen=max_se
quence_len, padding='pre'))
    predictors, label = input_sequences[:, :-1], input_sequences[:, -1]

```

```

    label = to_categorical(label, num_classes = total_words)
    return predictors, label, max_sequence_len
predictors, label, max_sequence_len = generate_padded_sequences(inp_sequences)
def create_model(max_sequence_len, total_words):
    input_len = max_sequence_len - 1;
    model = Sequential()

    # Add Input Embedding Layer
    model.add(Embedding(total_words, 10, input_length=input_len))

    # Add Hidden Layer 1 - LSTM Layer
    model.add(LSTM(100))
    model.add(Dropout(0.1))

    # Add Output Layer
    model.add(Dense(total_words, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    return model
model = create_model(max_sequence_len, total_words)
model.fit(predictors, label, epochs=20, verbose=5)

def generate_text(seed_text, next_words, model, max_sequence_len):
    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
    return seed_text.title()

```

Now let's take a look at our Heading production model:

```
print(generate_text("HAPPY", 5, model, max_sequence_len))
```

```
Output: The Secret Of HAPPY
```

**CONCLUSION AND FUTURE SCOPE:** This model concludes that we can get a suitable heading easily for anything we are in making. This can help to get a better heading in no time. A model of text generation language using machine learning, which will be used for the task of Heading generator for youtube videos or even for your blogs.