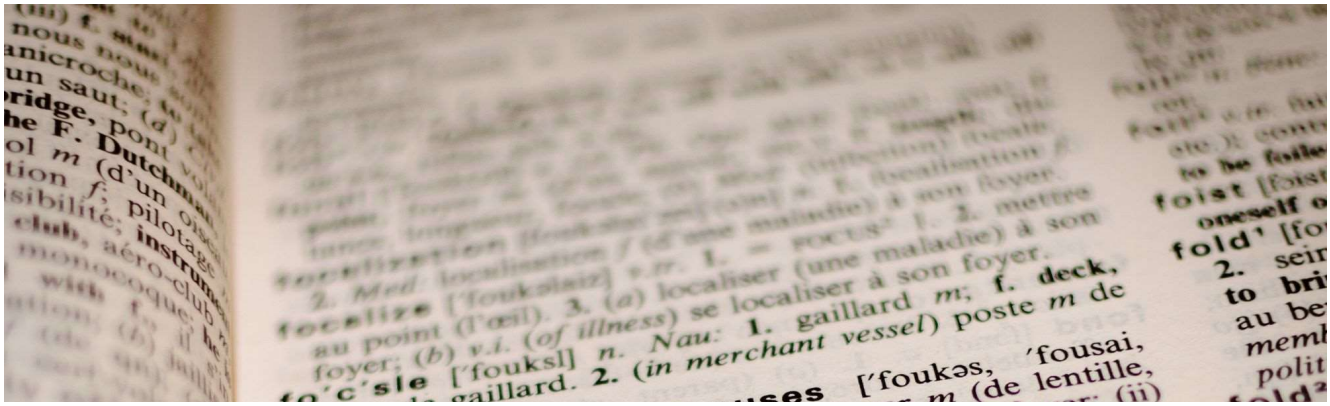
FLOYDHUB

MACHINE LEARNING

A Gentle Introduction to Text Summarization in Machine Learning

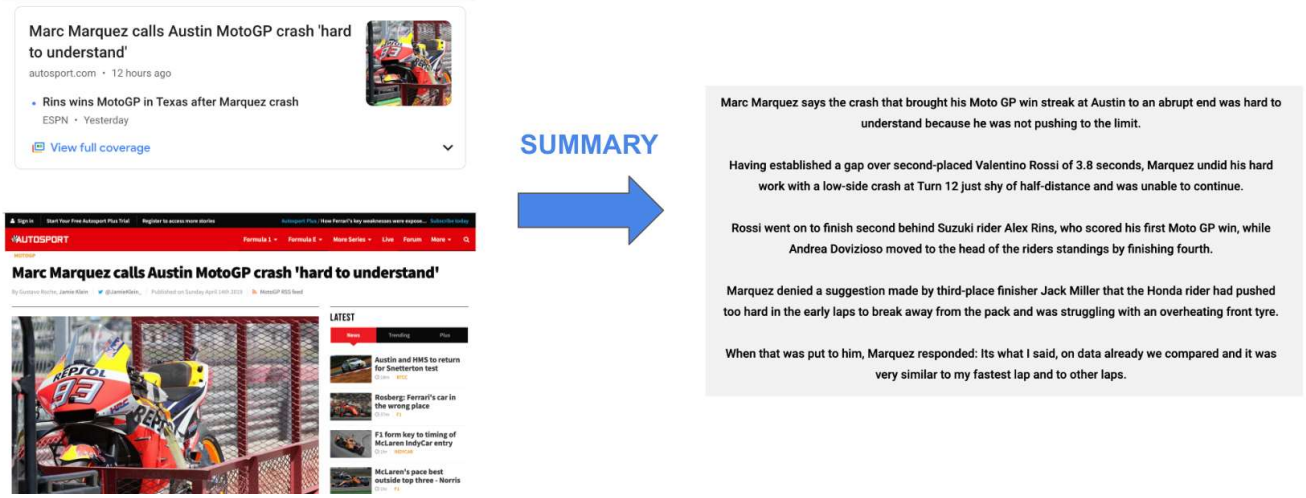
Text summarization is a common problem in the fields of machine learning and natural language processing (NLP). In this article, we'll explore how to create a simple extractive text summarization algorithm.

Alfrick Opidi
Apr 15, 2019 • 14 min read



Have you ever summarized a lengthy document into a short paragraph? How long did you take? Manually generating a summary can be time consuming and tedious. Automatic text summarization promises to overcome such difficulties and allow you to generate the key ideas in a piece of writing easily.

Text summarization is the technique for generating a concise and precise summary of voluminous texts while focusing on the sections that convey useful information, and without losing the overall meaning.
Automatic text summarization aims to transform lengthy documents into shortened versions, something which could be difficult and costly to undertake if done manually. Machine learning algorithms can be trained to comprehend documents and identify the sections that convey important facts and information before producing the required summarized texts. For example, the image below is of this news article that has been fed into a machine learning algorithm to generate a summary.



An online news article that has been summarized using a text summarization machine learning algorithm

The need for text summarization

With the present explosion of data circulating the digital space, which is mostly non-structured textual data, there is a need to develop automatic text summarization tools that allow people to get insights from them easily. Currently, we enjoy quick access to enormous amounts of information. However, most of this information is redundant, insignificant, and may not convey the intended meaning. For example, if you are looking for specific information from an online news article, you may have to dig through its content and spend a lot of time weeding out the unnecessary stuff before getting the information you want. Therefore, using automatic text summarizers capable of extracting useful information that leaves out inessential and insignificant data is becoming vital. Implementing summarization can enhance the readability of documents, reduce the time spent in researching for information, and allow for more information to be fitted in a particular area.

The main types of text summarization

Broadly, there are two approaches to summarizing texts in NLP: extraction and abstraction.

Extraction-based summarization

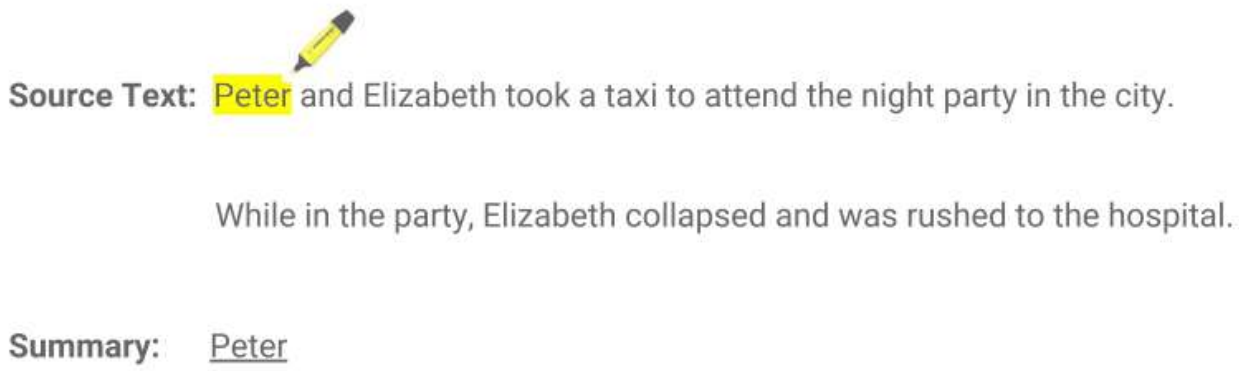
In extraction-based summarization, a subset of words that represent the most important points is pulled from a piece of text and combined to make a summary. Think

of it as a highlighter—which selects the main information from a source text.

Highlighter = Extractive-based summarization

In machine learning, extractive summarization usually involves weighing the essential sections of sentences and using the results to generate summaries.

Different types of algorithms and methods can be used to gauge the weights of the sentences and then rank them according to their relevance and similarity with one another—and further joining them to generate a summary. Here's an example:



Extractive-based summarization in action.

As you can see above, the extracted summary is composed of the words highlighted in bold, although the results may not be grammatically accurate.

Abstraction-based summarization

In abstraction-based summarization, advanced deep learning techniques are applied to

In abstraction-based summarization, advanced deep learning techniques are applied to paraphrase and shorten the original document, just like humans do. Think of it as a pen—which produces novel sentences that may not be part of the source document.

Pen = Abstraction-based summarization

Since abstractive machine learning algorithms can generate new phrases and sentences that represent the most important information from the source text, they can assist in overcoming the grammatical inaccuracies of the extraction techniques. Here is an example:

Source Text: Peter and Elizabeth took a taxi to attend the night party in the city.

While in the party, Elizabeth collapsed and was rushed to the hospital.

Summary: Elizabeth was hospitalized after attending a party with Peter.

Abstraction-based summary in action.

Although abstraction performs better at text summarization, developing its algorithms requires complicated deep learning techniques and sophisticated language modeling.

To generate plausible outputs, abstraction-based summarization approaches must address a wide variety of NLP problems, such as natural language generation, semantic representation, and inference permutation.

As such, extractive text summarization approaches are still widely popular. In this article, we'll be focusing on an extraction-based method.

How to perform text summarization

Let's use a short paragraph to illustrate how extractive text summarization can be performed.

Here is the paragraph:

"Peter and Elizabeth took a taxi to attend the night party in the city. While in the party, Elizabeth collapsed and was rushed to the hospital. Since she was diagnosed with a brain injury, the doctor told Peter to stay besides her until she gets well. Therefore, Peter stayed with her at the hospital for 3 days without leaving."

Here are the steps to follow to summarize the above paragraph, while trying to maintain its intended meaning, as much as possible.

Step 1: Convert the paragraph into sentences

First, let's split the paragraph into its corresponding sentences. The best way of doing the conversion is to extract a sentence whenever a period appears.

1. Peter and Elizabeth took a taxi to attend the night party in the city

2. While in the party, Elizabeth collapsed and was rushed to the hospital

3. Since she was diagnosed with a brain injury, the doctor told Peter to stay besides her until she gets well

4. Therefore, Peter stayed with her at the hospital for 3 days without leaving

Step 2: Text processing

Next, let's do text processing by removing the stop words (extremely common words with little meaning such as "and" and "the"), numbers, punctuation, and other special characters from the sentences.

Performing the filtering assists in removing redundant and insignificant information which may not provide any added value to the text's meaning

which may not provide any added value to the text's meaning.

Here is the result of the text processing:

1. Peter Elizabeth took taxi attend night party city

2. Party Elizabeth collapse rush hospital

3. Diagnose brain injury doctor told Peter stay besides get well

4. Peter stay hospital days without leaving

Step 3: Tokenization

Tokenizing the sentences is done to get all the words present in the sentences. Here is a list of the words:

```
['peter','elizabeth','took','taxi','attend','night','party','city','party','elizabeth','collapse','rush','hospital', 'diagnose','brain', 'injury', 'doctor','told','peter','stay','besides','get','well','peter', 'stayed','hospital','days','without','leaving']
```

Step 4: Evaluate the weighted occurrence frequency of the words

Thereafter, let's calculate the weighted occurrence frequency of all the words. To achieve this, let's divide the occurrence frequency of each of the words by the frequency of the most recurrent word in the paragraph, which is "Peter" that occurs three times.

Here is a table that gives the weighted occurrence frequency of each of the words.

Word	Frequency	Weighted Frequency
peter	3	1
elizabeth	2	0.67
took	1	0.33
taxi	1	0.33
attend	1	0.33
night	1	0.33
party	2	0.67
city	1	0.33

city	1	0.33
collapse	1	0.33
rush	1	0.33
hospital	2	0.67
diagnose	1	0.33
brain	1	0.33
injury	1	0.33
doctor	1	0.33
told	1	0.33
stay	2	0.67
besides	1	0.33
get	1	0.33
well	1	0.33
days	1	0.33
without	1	0.33
leaving	1	0.33

Step 5: Substitute words with their weighted frequencies

Let's substitute each of the words found in the original sentences with their weighted frequencies. Then, we'll compute their sum.

Since the weighted frequencies of the insignificant words, such as stop words and special characters, which were removed during the processing stage, is zero, it's not necessary to add them.

Sentence	Add weighted frequencies	Sum
1	Peter and Elizabeth took a taxi to attend the night party in the city	1 + 0.67 + 0.33 + 0.33 + 0.33 + 0.33 + 0.67 + 0.33	3.99
2	While in the party, Elizabeth collapsed and was rushed to the hospital	0.67 + 0.67 + 0.33 + 0.33 + 0.67	2.67
3	Since she was diagnosed with a brain injury, the doctor told Peter to stay besides her until she gets well.	0.33 + 0.33 + 0.33 + 0.33 + 1 + 0.33 + 0.33 + 0.33 + 0.33 +0.33	3.97
4	Therefore, Peter stayed with her at the hospital for 3 days without leaving	1 + 0.67 + 0.67 + 0.33 + 0.33 + 0.33	3.33

From the sum of the weighted frequencies of the words, we can deduce that the first sentence carries the most weight in the paragraph. Therefore, it can give the best representative summary of what the paragraph is about.

Furthermore, if the first sentence is combined with the third sentence, which is the second-most weighty sentence in the paragraph, a better summary can be generated.

The above example just gives a basic illustration of how to perform extraction-based text summarization in machine learning. Now, let's see how we can apply the concept above in creating a real-world summary generator.

Text summarization of a Wikipedia article

Let's get our hands dirty by creating a text summarizer that can shorten the information found in a lengthy web article. To keep things simple, apart from Python's NLTK toolkit, we'll not use any other machine learning library.

Here is the code blueprint of the summarizer:

```
# Creating a dictionary for the word frequency table
frequency_table = _create_dictionary_table(article)

# Tokenizing the sentences
sentences = sent_tokenize(article)

# Algorithm for scoring a sentence by its words
sentence_scores = _calculate_sentence_scores(sentences, frequency_table)

# Getting the threshold
threshold = _calculate_average_score(sentence_scores)

# Producing the summary
article_summary = _get_article_summary(sentences, sentence_scores, 1.5 * threshold)

print(article_summary)
```

Here are the steps for creating a simple text summarizer in Python.

Here are the steps for creating a simple text summarizer in Python:

Step 1: Preparing the data

In this example, we want to summarize the information found on this Wikipedia article, which just gives an overview of the major happenings during the 20th century.

To enable us to fetch the article's text, we'll use the Beautiful Soup library.

Here is the code for scraping the article's content:

```
import bs4 as BeautifulSoup
import urllib.request

# Fetching the content from the URL
fetched_data = urllib.request.urlopen('https://en.wikipedia.org/wiki/20th_century')

article_read = fetched_data.read()

# Parsing the URL content and storing in a variable

article_parsed = BeautifulSoup.BeautifulSoup(article_read,'html.parser')

# Returning <p> tags
paragraphs = article_parsed.find_all('p')

article_content = ''

# Looping through the paragraphs and adding them to the variable
for p in paragraphs:
    article_content += p.text
```

In the above code, we begin by importing the essential libraries for fetching data from the web page. The **BeautifulSoup** library is used for parsing the page while the **urllib library** is used for connecting to the page and retrieving the HTML.

BeautifulSoup converts the incoming text to Unicode characters and the outgoing text to UTF-8 characters, saving you the hassle of managing different charset encodings while scraping text from the web.

We'll use the `urlopen` function from the `urllib.request` utility to open the web

page. Then, we'll use the `read` function to read the scraped data object. For parsing the data, we'll call the `BeautifulSoup` object and pass two parameters to it; that is, the `article_read` and the `html.parser`.

The `find_all` function is used to return all the `<p>` elements present in the HTML. Furthermore, using `.text` enables us to select only the texts found within the `<p>` elements.

Step 2: Processing the data

To ensure the scrapped textual data is as noise-free as possible, we'll perform some basic text cleaning. To assist us to do the processing, we'll import a list of **stopwords** from the **nltk** library.

We'll also import **PorterStemmer**, which is an algorithm for reducing words into their root forms. For example, *cleaning*, *cleaned*, and *cleaner* can be reduced to the root *clean*.

Furthermore, we'll create a dictionary table having the frequency of occurrence of each of the words in the text. We'll loop through the text and the corresponding words to eliminate any stop words.

Then, we'll check if the words are present in the `frequency_table`. If the word was previously available in the dictionary, its value is updated by 1. Otherwise, if the word is recognized for the first time, its value is set to 1.

For example, the frequency table should look like the following:

Word	Frequency
century	7
world	4
United States	3
computer	1

Here is the code:

```
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
```

```
def _create_dictionary_table(text_string) -> dict:

    # Removing stop words
    stop_words = set(stopwords.words("english"))

    words = word_tokenize(text_string)

    # Reducing words to their root form
    stem = PorterStemmer()

    # Creating dictionary for the word frequency table
    frequency_table = dict()
    for wd in words:
        wd = stem.stem(wd)
        if wd in stop_words:
            continue
        if wd in frequency_table:
            frequency_table[wd] += 1
        else:
            frequency_table[wd] = 1

    return frequency_table
```

Step 3: Tokenizing the article into sentences

To split the `article_content` into a set of sentences, we'll use the built-in method from the **nltk** library.

```
from nltk.tokenize import word_tokenize, sent_tokenize

sentences = sent_tokenize(article)
```

Step 4: Finding the weighted frequencies of the sentences

To evaluate the score for every sentence in the text, we'll be analyzing the frequency of occurrence of each term. In this case, we'll be scoring each sentence by its words; that is, adding the frequency of each important word found in the sentence.

Take a look at the following code:

```
def _calculate_sentence_scores(sentences, frequency_table) -> dict:
```

```
    # Algorithm for scoring a sentence by its words
    sentence_weight = dict()

    for sentence in sentences:
        sentence_wordcount = (len(word_tokenize(sentence)))
        sentence_wordcount_without_stop_words = 0
        for word_weight in frequency_table:
            if word_weight in sentence.lower():
                sentence_wordcount_without_stop_words += 1
                if sentence[:7] in sentence_weight:
                    sentence_weight[sentence[:7]] += frequency_table[word_weight]
                else:
                    sentence_weight[sentence[:7]] = frequency_table[word_weight]

        sentence_weight[sentence[:7]] = sentence_weight[sentence[:7]] /        sent

    return sentence_weight
```

Importantly, to ensure long sentences do not have unnecessarily high scores over short sentences, we divided each score of a sentence by the number of words found in that sentence.

Also, to optimize the dictionary's memory, we arbitrarily added **sentence[:7],** which refers to the first 7 characters in each sentence. However, for longer documents, where you are likely to encounter sentences with the same first `n_chars`, it's better to use hash functions or smart index functions to take into account such edge-cases and avoid collisions.

Step 5: Calculating the threshold of the sentences

To further tweak the kind of sentences eligible for summarization, we'll create the average score for the sentences. With this threshold, we can avoid selecting the sentences with a lower score than the average score.

Here is the code:

```
def _calculate_average_score(sentence_weight) -> int:

    # Calculating the average score for the sentences
    sum_values = 0
    for entry in sentence_weight:
```

```
            sum_values += sentence_weight[entry]

        # Getting sentence average value from source text
        average_score = (sum_values / len(sentence_weight))

        return average_score
```

Step 6: Getting the summary

Lastly, since we have all the required parameters, we can now generate a summary for the article.

Here is the code:

```
def _get_article_summary(sentences, sentence_weight, threshold):
    sentence_counter = 0
    article_summary = ''

    for sentence in sentences:
        if sentence[:7] in sentence_weight and sentence_weight[sentence[:7]] >= (th
            article_summary += " " + sentence
            sentence_counter += 1

    return article_summary
```

Wrapping Up

Here is an image that showcases the workflow for creating the summary generator.

A basic workflow of creating a summarization algorithm

Here is the entire code for the simple extractive text summarizer in machine learning:

```
#importing libraries
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize, sent_tokenize
import bs4 as BeautifulSoup
import urllib.request

#fetching the content from the URL
fetched_data = urllib.request.urlopen('https://en.wikipedia.org/wiki/20th_century')

article_read = fetched_data.read()

#parsing the URL content and storing in a variable
article_parsed = BeautifulSoup.BeautifulSoup(article_read,'html.parser')

#returning <p> tags
paragraphs = article_parsed.find_all('p')

article_content = ''

#looping through the paragraphs and adding them to the variable
for p in paragraphs:
    article_content += p.text


def _create_dictionary_table(text_string) -> dict:

    #removing stop words
    stop_words = set(stopwords.words("english"))

    words = word_tokenize(text_string)

    #reducing words to their root form
    stem = PorterStemmer()

    #creating dictionary for the word frequency table
    frequency_table = dict()
    for wd in words:
        wd = stem.stem(wd)
        if wd in stop_words:
            continue
        if wd in frequency_table:
            frequency_table[wd] += 1
        else:
            frequency_table[wd] = 1
```

```
    return frequency_table


def _calculate_sentence_scores(sentences, frequency_table) -> dict:

    #algorithm for scoring a sentence by its words
    sentence_weight = dict()

    for sentence in sentences:
        sentence_wordcount = (len(word_tokenize(sentence)))
        sentence_wordcount_without_stop_words = 0
        for word_weight in frequency_table:
            if word_weight in sentence.lower():
                sentence_wordcount_without_stop_words += 1
                if sentence[:7] in sentence_weight:
                    sentence_weight[sentence[:7]] += frequency_table[word_weight]
                else:
                    sentence_weight[sentence[:7]] = frequency_table[word_weight]

        sentence_weight[sentence[:7]] = sentence_weight[sentence[:7]] / sentence_wo



    return sentence_weight


def _calculate_average_score(sentence_weight) -> int:

    #calculating the average score for the sentences
    sum_values = 0
    for entry in sentence_weight:
        sum_values += sentence_weight[entry]

    #getting sentence average value from source text
    average_score = (sum_values / len(sentence_weight))

    return average_score

def _get_article_summary(sentences, sentence_weight, threshold):
    sentence_counter = 0
    article_summary = ''

    for sentence in sentences:
        if sentence[:7] in sentence_weight and sentence_weight[sentence[:7]] >= (th
            article_summary += " " + sentence
            sentence_counter += 1

    return article_summary

def _run_article_summary(article):
```

```
    #creating a dictionary for the word frequency table
    frequency_table = _create_dictionary_table(article)

    #tokenizing the sentences
    sentences = sent_tokenize(article)

    #algorithm for scoring a sentence by its words
    sentence_scores = _calculate_sentence_scores(sentences, frequency_table)

    #getting the threshold
    threshold = _calculate_average_score(sentence_scores)

    #producing the summary
    article_summary = _get_article_summary(sentences, sentence_scores, 1.5 * thresh

    return article_summary

if __name__ == '__main__':
    summary_results = _run_article_summary(article_content)
    print(summary_results)
```

You can click the following button to run the code on the FloydHub Notebook:

Run on FloydHub

In this case, we applied a threshold of 1.5x of the average score. It's the hyperparameter value that generated for us good results after a couple of trials. Of course, you can fine-tune the value according to your preferences and improve the summarization outcomes.

Here is an image of the summarized version of the Wikipedia article.

```
 It is distinct from the century known as the 1900s which began on January 1, 1900 and ended on December 31, 19
99. Terms like ideology, world war, genocide, and nuclear war entered common usage. Humans explored space for t
he first time, taking their first footsteps on the Moon. However, these same wars resulted in the destruction o
f the imperial system. The victorious Bolsheviks then established the Soviet Union, the world's first communist
state. In total, World War II left some 60 million people dead. During the century, the social taboo of sexism
fell. Communications and information technology, transportation technology, and medical advances had radically
altered daily lives. Since the US was in a dominant position, a major part of the process was Americanization.
Terrorism, dictatorship, and the spread of nuclear weapons were pressing global issues. Millions were infected
with HIV, the virus which causes AIDS. This includes deaths caused by wars, genocide, politicide and mass murde
rs. Later in the 20th century, the development of computers led to the establishment of a theory of computation
```

A Wikipedia article summarized using a summarization algorithm

As you can see, running the code summarizes the lengthy Wikipedia article and gives a

simplistic overview of the main happenings in the 20th century.

Nonetheless, the summary generator can be improved to make it better at producing a concise and precise summary of voluminous texts.

Taking things a notch higher...

Of course, this article just brushed the surface of what you can achieve with a text summarization algorithm in machine learning.

To learn more about the subject, especially about abstractive text summarization, here are some useful resources you can use:

- Is it possible to combine the two approaches (abstractive and extractive)? It is the main idea behind the pointer-generator network that gets the best of both worlds by combining both extraction(pointing) and abstraction(generating).

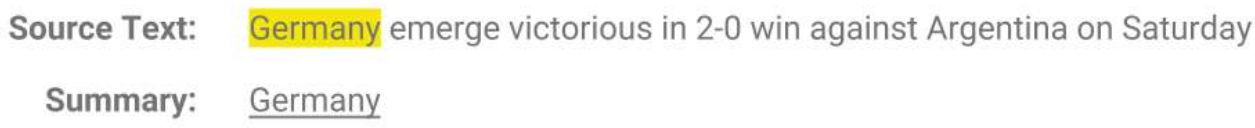


Image from "Taming Recurrent Neural Networks for Better Summarization"

- How to use WikiHow, a large-scale text summarization dataset—This paper introduces WikiHow, a new large-scale text summarization dataset that comprises of more than 230,000 articles extracted from the WikiHow online knowledge base. Most of the presently available datasets are not large enough for training sequence-to-sequence models, they may provide only limited summaries, and they are more suited to performing extractive summarization. However, the WikiHow dataset is large-scale, high-quality, and capable of achieving optimal results in abstractive summarization.
- How a pretraining-based encoder-decoder framework can be used in text summarization—This paper introduces a unique two-stage model that is based on a sequence-to-sequence paradigm. The model makes use of BERT (you can bet that we will continue to read about BERT in all 2019) on both encoder and decoder sides and focuses on reinforced objective during the learning process. When the model was assessed on some benchmark datasets, the outcome

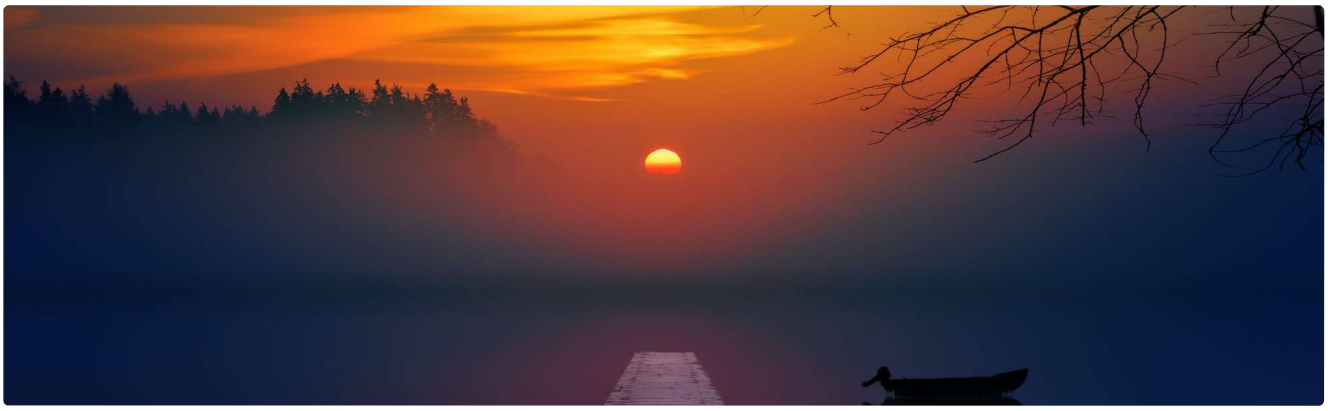
> revealed that the approach performed better at text summarization, particularly when compared to other traditional systems.

Special appreciation to the entire team at FloydHub, especially Alessio, for their valuable feedback and support in enhancing the quality and the flow of this article. You guys rock!

About Alfrick Opidi

Alfrick is a web developer with a deep interest in exploring the world of machine learning. Currently, he's involved in projects that implement machine learning concepts in producing agile and futuristic web applications. In his free time, he engages in

technical writing to demystify complex machine learning concepts for humans. See him as an all-round tech geek with a keen eye on making the latest developments in the industry accessible and fun to learn. Alfrick is also a FloydHub AI Writer. You can find out more about him here. You can connect with Alfrick on LinkedIn and GitHub.



FloydHub has shut down

FloydHub - our ML platform used by thousands of Data Scientists and AI enthusiasts was shut down on August 20, 2021.

Naren Thiagarajan
Aug 21, 2021 • 1 min read