# Unsupervised Text Summarization using Sentence Embeddings

Kushal Chauhan
Aug 6, 2018 · 13 min read

Greetings to the readers! I am a pre-final year graduate student pursuing an integrated B. Tech+M.Tech degree in Information Technology. I am currently working as an **NLP Research Intern** at **Jatana.ai**. In this article, I will describe the approach that I used to perform Text Summarization in Python, one of the awesome list of tasks that I was assigned to by my mentors at Jatana.

## What is Text Summarization?

> Text summarization is the process of distilling the most important information from a source (or sources) to produce an abridged version for a particular user (or users) and task (or tasks).

— Page 1, Advances in Automatic Text Summarization, 1999.

Humans are generally quite good at this task as we have the capacity to understand the meaning of a text document and extract salient features to summarize the documents using our own words. However, automatic methods for text summarization are crucial in today's world where there is an over-abundance of data and lack of manpower as well as time to interpret the data. There are many reasons why Automatic Text Summarization is useful:

1. Summaries reduce reading time.

2. When researching documents, summaries make the selection process easier.

3. Automatic summarization improves the effectiveness of indexing.

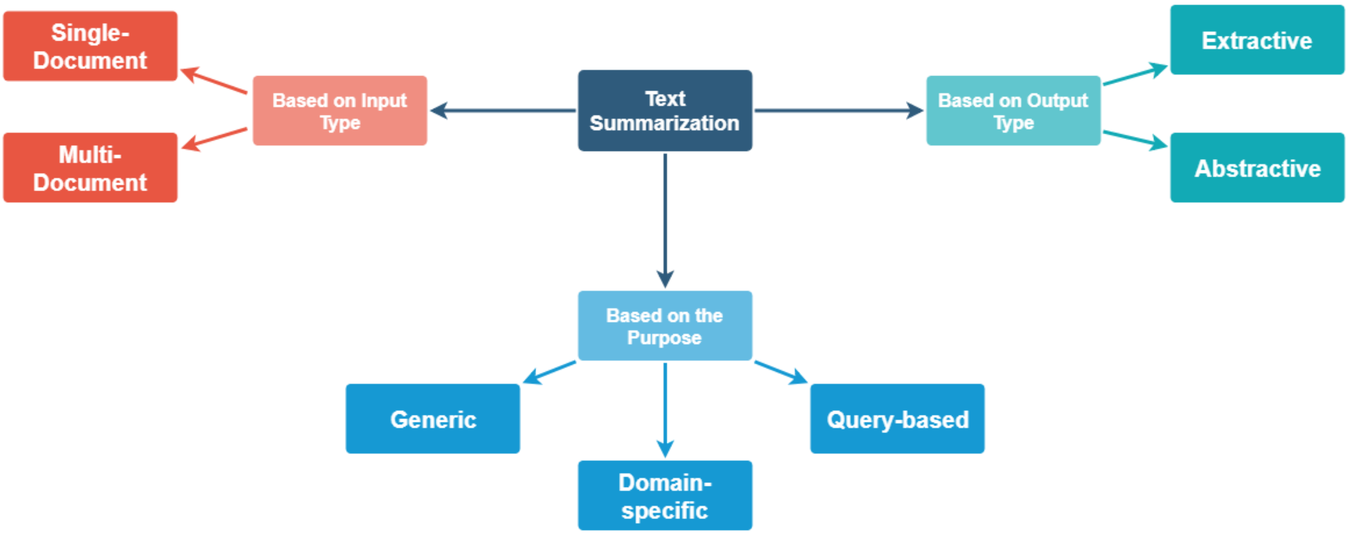
4. Automatic summarization algorithms are less biased than human summarizers.

5. Personalized summaries are useful in question-answering systems as they provide personalized information.

6. Using automatic or semi-automatic summarization systems enables commercial abstract services to increase the number of text documents they are able to process.

## Types of Text Summarization Methods:

Text summarization methods can be classified into different types.



Types of Text Summarization approaches

### Based on input type:

1. *S*ingle Document, where the input length is short. Many of the early summarization systems dealt with single document summarization.

2. Multi Document, where the input can be arbitrarily long.

### Based on the purpose:

1. Generic, where the model makes no assumptions about the domain or content of the text to be summarized and treats all inputs as homogeneous. The majority of the work that has been done revolves around generic summarization.

2. Domain-specific, where the model uses domain-specific knowledge to form a more accurate summary. For example, summarizing research papers of a specific domain, biomedical documents, etc.

3. Query-based, where the summary only contains information which answers natural language questions about the input text.

**Based on output type:**

1. Extractive, where important sentences are selected from the input text to form a summary. Most summarization approaches today are extractive in nature.

2. Abstractive, where the model forms its own phrases and sentences to offer a more coherent summary, like what a human would generate. This approach is definitely a more appealing, but much more difficult than extractive summarization.

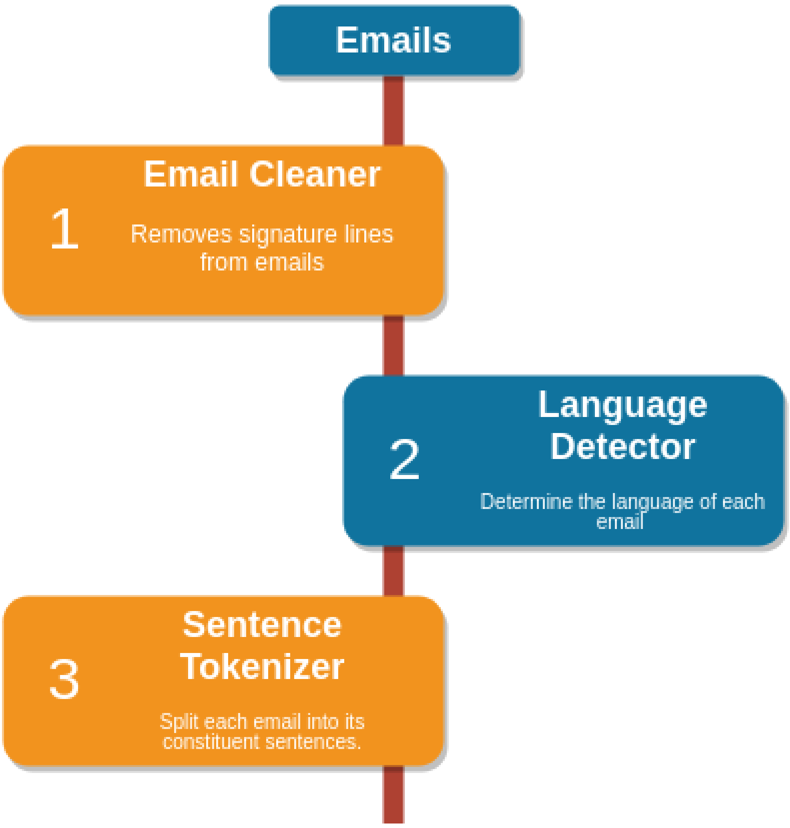
## My Task

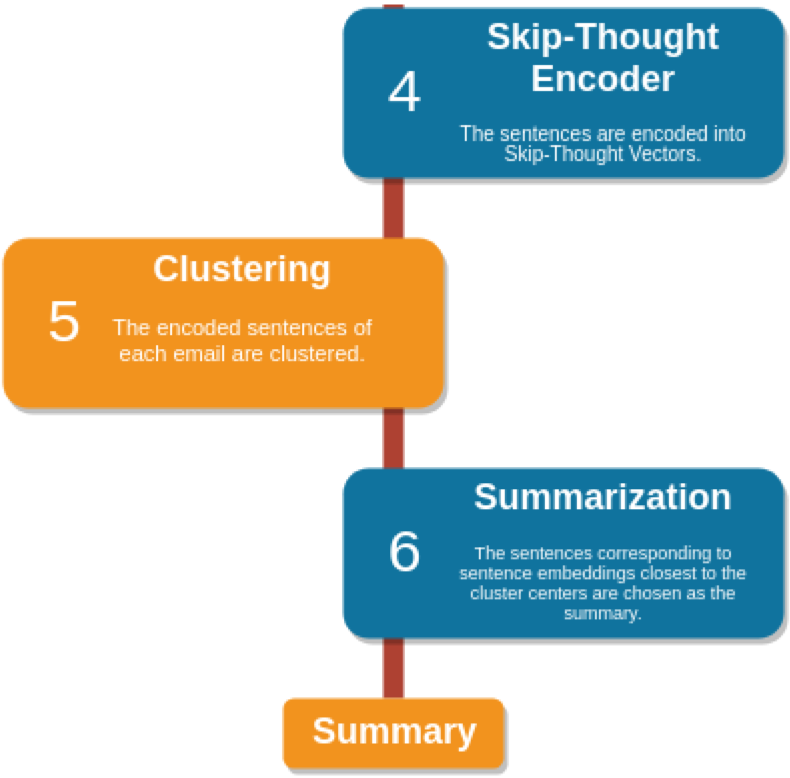
The task was to perform Text Summarization on emails in languages such as English, Danish, French, etc. using Python. Most publicly available datasets for text summarization are for long documents and articles. As the structure of long documents and articles significantly differs from that of short emails, models trained with supervised methods may suffer from poor domain adaptation. Therefore, I chose to explore unsupervised methods for unbiased prediction of summaries.

Now, let us try to understand the various steps which constitute the model pipeline.

## Text Summarization Model Pipeline

The approach I adopted for Text Summarization was inspired from this paper. Let's break it down in several steps:


Emails
1
Email Cleaner
Removes signature lines from emails
2
Language Detector
Determine the language of each email
3
Sentence Tokenizer
Split each email into its constituent sentences.



4 Skip-Thought Encoder

The sentences are encoded into Skip-Thought Vectors.

5 Clustering

The encoded sentences of each email are clustered.

6 Summarization

The sentences corresponding to sentence embeddings closest to the cluster centers are chosen as the summary.

Summary

## Step-1: Email Cleaning

To motivate this step, let us first see what some typical emails looks like:

*An example of an English email:*

```
Hi Jane,

Thank you for keeping me updated on this issue. I'm happy to hear
that the issue got resolved after all and you can now use the app in
its full functionality again.
Also many thanks for your suggestions. We hope to improve this
feature in the future.

In case you experience any further problems with the app, please
don't hesitate to contact me again.

Best regards,

John Doe
Customer Support

1600 Amphitheatre Parkway
Mountain View, CA
United States
```

*An example of a Norwegian email:*

```
Hei

Grunnet manglende dekning på deres kort for månedlig trekk, blir
dere nå overført til årlig fakturering.
I morgen vil dere motta faktura for hosting og drift av nettbutikk
for perioden 05.03.2018-05.03.2019.
Ta gjerne kontakt om dere har spørsmål.

Med vennlig hilsen
John Doe - SomeCompany.no
04756 | johndoe@somecompany.no

Husk å sjekk vårt hjelpesenter, kanskje du finner svar der:
https://support.somecompany.no/
```

*An example of an Italian email:*

```
Ciao John,

Grazie mille per averci contattato! Apprezziamo molto che abbiate
trovato il tempo per inviarci i vostri commenti e siamo lieti che vi
piaccia l'App.

Sentitevi liberi di parlare di con i vostri amici o di sostenerci
lasciando una recensione nell'App Store!

Cordiali saluti,

Jane Doe
Customer Support

One Infinite Loop
Cupertino
CA 95014
```

As one can see, the salutation and signature lines at the beginning and the end of an email contributes no value for the task of summary generation. So, it is necessary to remove these lines from the email, which we know, shouldn't contribute to the summary. This makes for a simpler input which a model can perform better with.

As salutation and signature lines can vary from email to email and from one language to the other, removing them will require matching against a regular expression. For implementing this module I used a slightly modified version of code found in the Mailgun Talon GitHub repository so that it also supports other languages. The module also removes newline characters. The shorter version of the code goes like this:

```python
# clean() is a modified version of extract_signature() found in bruteforce.py in the Git
cleaned_email, _ = clean(email)

lines = cleaned_email.split('\n')
lines = [line for line in lines if line != '']
cleaned_email = ' '.join(lines)
```

**email_cleaner.py** hosted with ❤ by **GitHub** view raw

Instead of modifying the code to create your own *clean()* you can also use:

```python
from talon.signature.bruteforce import extract_signature
cleaned_email, _ = extract_signature(email)
```

**talon_cleaner.py** hosted with ❤ by **GitHub** view raw

The cleaned versions of the above emails will look like:

*The cleaned English email:*

```
Thank you for keeping me updated on this issue. I'm happy to hear
that the issue got resolved after all and you can now use the app in
its full functionality again. Also many thanks for your suggestions.
We hope to improve this feature in the future. In case you
experience any further problems with the app, please don't hesitate
to contact me again.
```

*The cleaned Norwegian email:*

```
Grunnet manglende dekning på deres kort for månedlig trekk, blir
dere nå overført til årlig fakturering. I morgen vil dere motta
faktura for hosting og drift av nettbutikk for perioden 05.03.2018-
05.03.2019. Ta gjerne kontakt om dere har spørsmål.
```

*The cleaned Italian email:*

```
Grazie mille per averci contattato! Apprezziamo molto che abbiate
trovato il tempo per inviarci i vostri commenti e siamo lieti che vi
piaccia l'App. Sentitevi liberi di parlare di con i vostri amici o
di sostenerci lasciando una recensione nell'App Store.
```

Once we are done with the preprocessing step, we can proceed to explore the rest of the summarization pipeline.

## Step-2: Language Detection

As the emails to be summarized can be of any language, the first thing one needs to do is to determine which language an email is in. Many Python libraries are available which use machine learning techniques to identify the language a piece of text is written in. Some examples are *polyglot*, *langdetect* and *textblob*. I used *langdetect* for my purpose and it supports 55 different languages. Language detection can be performed by just one simple function call:

```
from langdetect import detect
lang = detect(cleaned_email) # lang = 'en' for an English email
```

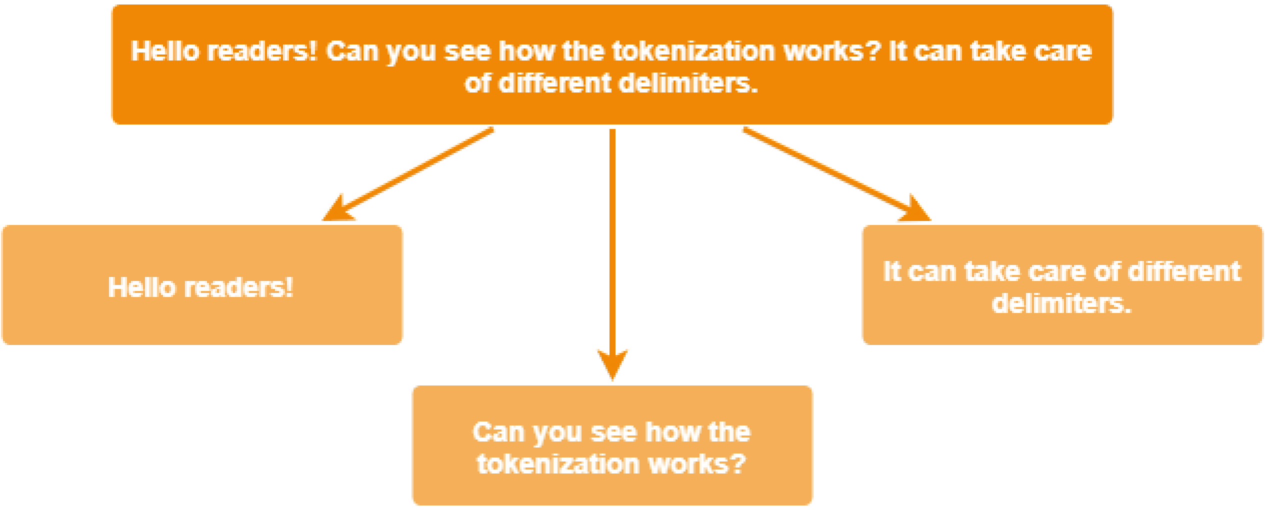
detect_lang.py hosted with ❤ by GitHub view raw

## Step-3: Sentence Tokenization

Once the languages identification is performed for every email, we can use this information to split each email into its constituent sentences using specific rules for sentence delimiters for each language. NLTK's sentence tokenizer will do this job for us:

```
from nltk.tokenize import sent_tokenize
sentences = sent_tokenize(email, language = lang)
```

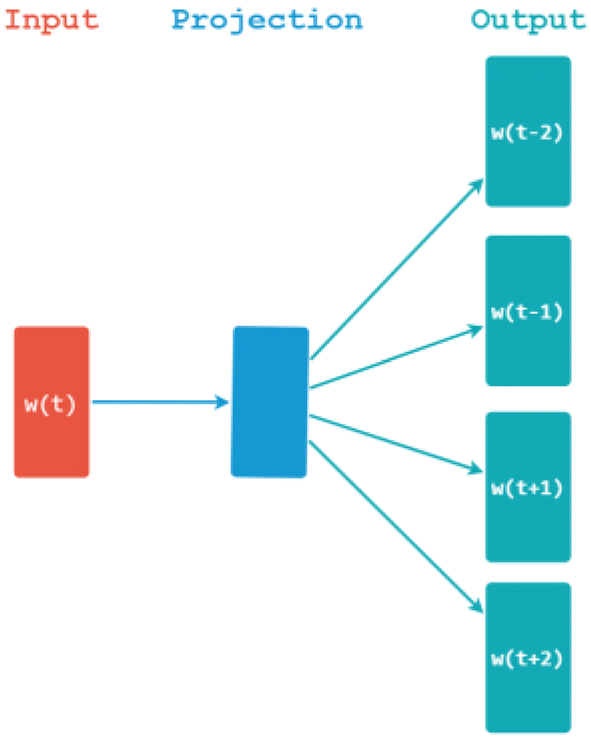
split_sentences.py hosted with ❤ by GitHub view raw


Hello readers! Can you see how the tokenization works? It can take care of different delimiters.
Hello readers!
Can you see how the tokenization works?
It can take care of different delimiters.

NLTK's Sentence Tokenizer

## Step-4: Skip-Thought Encoder

We need a way to generate fixed length vector representations for each sentence in our emails. These representations should encode the inherent semantics and the meaning of the corresponding sentence. The well known Skip-Gram Word2Vec method for generating word embeddings can give us word embeddings for individual words that are present in our model's vocabulary (some fancier approaches can also generate embeddings for words which are not in the model vocabulary using subword information).



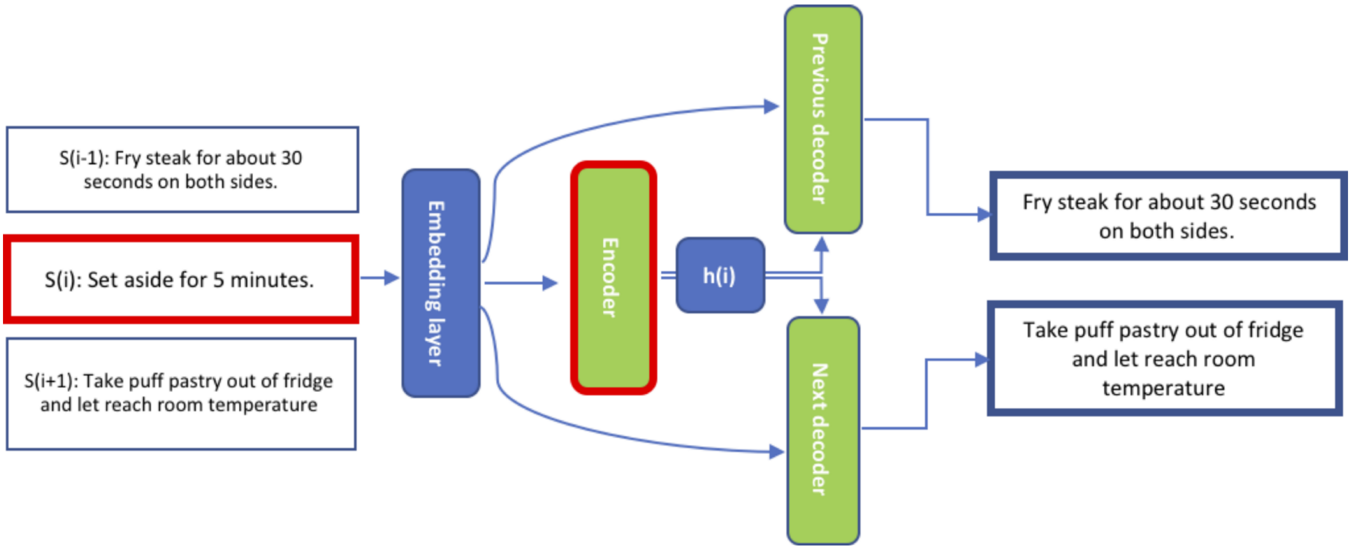
The Skip-Gram Word2Vec Model is trained to predict surrounding words given an input word.

For sentence embeddings, one easy way is to take a weighted sum of the word vectors for the words contained in the sentence. We take a weighted sum because frequently occurring words such as '*and*', '*to*' and '*the*', provide little or no information about the sentence. Some rarely occurring words, which are unique to a few sentences have much more representative power. Hence, we take the weights as being inversely related to the frequency of word occurrence. This method is described in detail in this paper.

However, these unsupervised methods do not take the sequence of words in the sentence into account. This may incur undesirable losses in model performance. To overcome this, I chose to instead train a Skip-Thought sentence encoder in a supervised manner using Wikipedia dumps as training data. The Skip-Thoughts model consists of two parts:

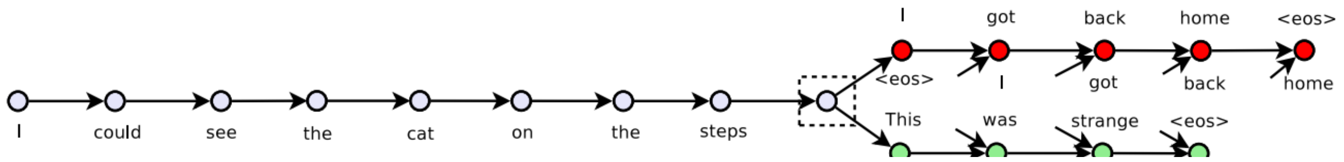
1. **Encoder Network:** The encoder is typically a GRU-RNN which generates a fixed length vector representation *h*(i) for each sentence *S(i)* in the input. The encoded representation *h(i)* is obtained by passing final hidden state of the GRU cell (i.e. after it has seen the entire sentence) to multiple dense layers.

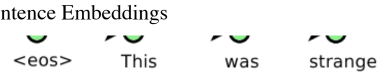
2. **Decoder Network:** The decoder network takes this vector representation *h*(i) as input and tries to generate two sentences — *S(i-1)* and *S(i+1)*, which could occur before and after the input sentence respectively. Separate decoders are implemented for generation of previous and next sentences, both being GRU-RNNs. The vector representation *h(i)* acts as the initial hidden state for the GRUs of the decoder networks.


S(i-1): Fry steak for about 30 seconds on both sides.
S(i): Set aside for 5 minutes.
S(i+1): Take puff pastry out of fridge and let reach room temperature
Embedding layer
Encoder
h(i)
Previous decoder
Next decoder
Fry steak for about 30 seconds on both sides.
Take puff pastry out of fridge and let reach room temperature

Skip Thoughts Model Overview

Given a dataset containing a sequence of sentences, the decoder is expected to generate the previous and next sentences, word by word. The encoder-decoder network is trained to minimize the sentence reconstruction loss, and in doing so, the encoder learns to generate vector representations which encode enough information for the decoder, so that it can generate neighboring sentences. These learned representations are such that embeddings of semantically similar sentences are closer to each other in vector space, and therefore are suitable for clustering. The sentences in our emails are given as input to the encoder network to obtain the desired vector representations. This Skip-Thoughts approach for obtaining sentence embeddings is described in detail in this paper.



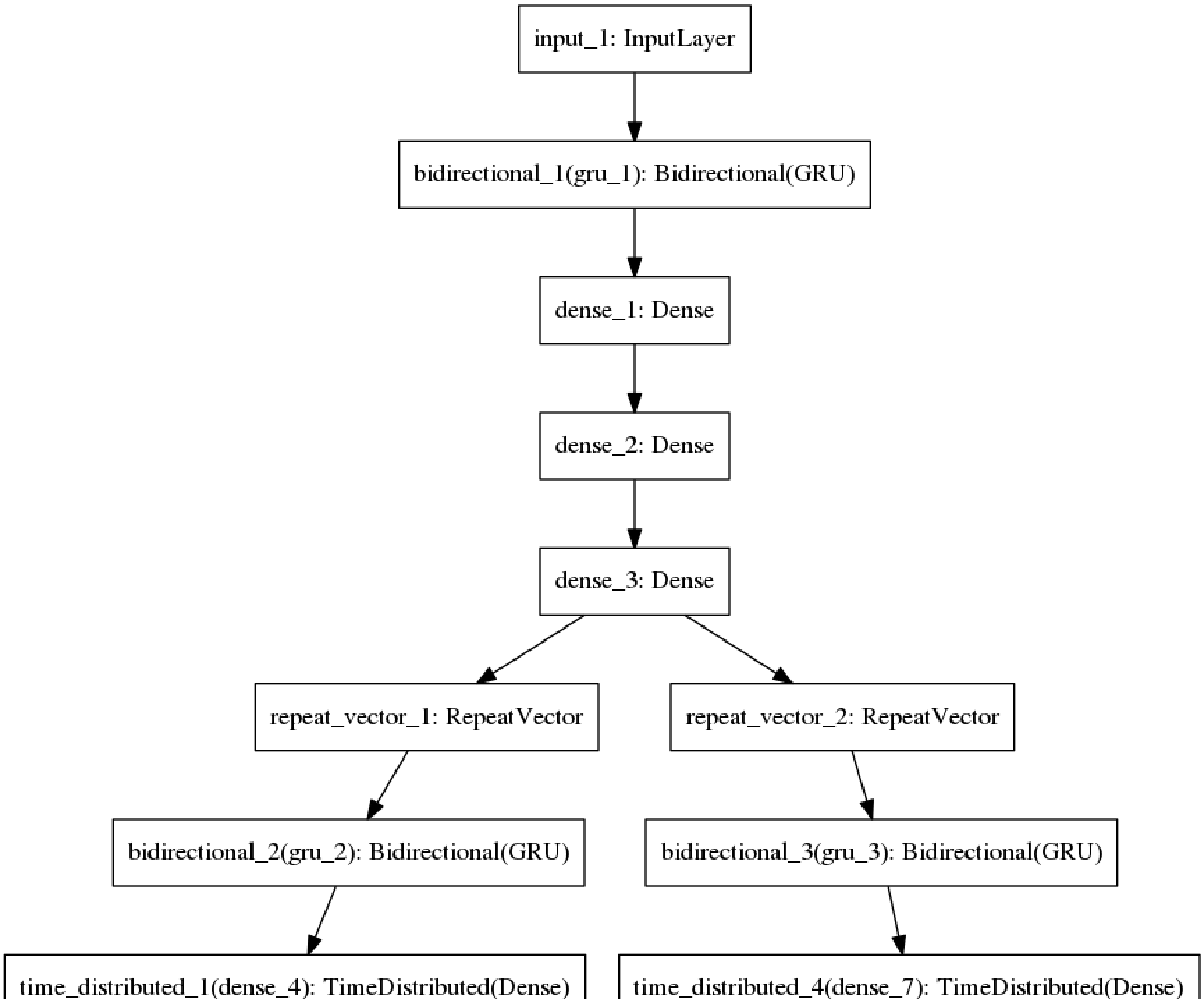


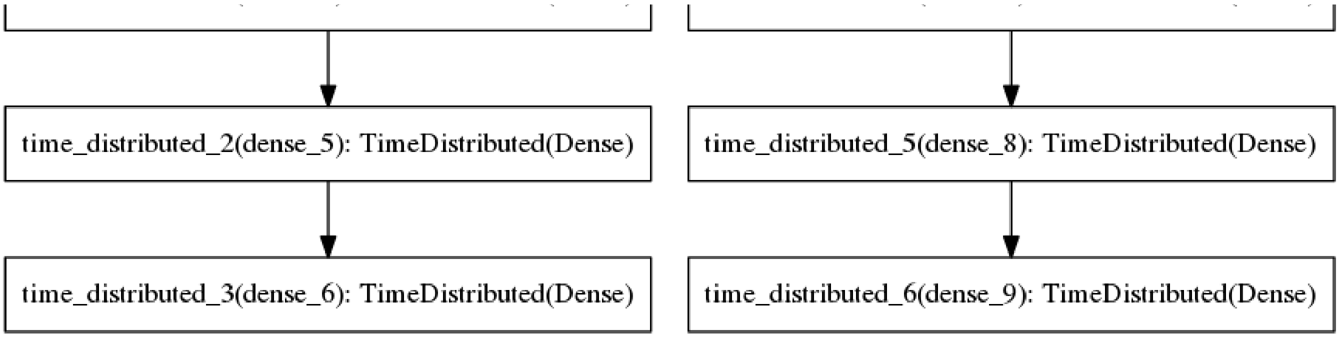
Given a sentence (the grey dots), the model attempts to predict the preceding sentence (red dots) and the next sentence (green dots). Image Source: https://arxiv.org/pdf/1506.06726.pdf

For implementation, I have used code made open-source by the author of the skip-thoughts paper. It is written in Theano and can be found here. The task of obtaining embeddings for every sentence in an email can be done in a few lines of code:

```
# The 'skipthoughts' module can be found at the root of the GitHub  repository linked ab
import skipthoughts

# You would need to download pre-trained models first
model = skipthoughts.load_model()

encoder = skipthoughts.Encoder(model)
encoded =  encoder.encode(sentences)
```

**skipthought_encode.py** hosted with ❤ by **GitHub** view raw


input_1: InputLayer
bidirectional_1(gru_1): Bidirectional(GRU)
dense_1: Dense
dense_2: Dense
dense_3: Dense
repeat_vector_1: RepeatVector
repeat_vector_2: RepeatVector
bidirectional_2(gru_2): Bidirectional(GRU)
bidirectional_3(gru_3): Bidirectional(GRU)
time_distributed_1(dense_4): TimeDistributed(Dense)
time_distributed_4(dense_7): TimeDistributed(Dense)


time_distributed_2(dense_5): TimeDistributed(Dense)
time_distributed_5(dense_8): TimeDistributed(Dense)
time_distributed_3(dense_6): TimeDistributed(Dense)
time_distributed_6(dense_9): TimeDistributed(Dense)

Skip-Thoughts Encoder-Decoder Architecture

## Step-5: Clustering

After generating sentence embeddings for each sentence in an email, the approach is to cluster these embeddings in high-dimensional vector space into a pre-defined number of clusters. The number of clusters will be equal to desired number of sentences in the summary. I chose the numbers of sentences in the summary to be equal to the square root of the total number of sentence in the email. One can also have it as being equal to, say, 30% of the total number of sentences. Here's the code that can do the clustering for you:

```python
import numpy as np
from sklearn.cluster import KMeans

n_clusters = np.ceil(len(encoded)**0.5)
kmeans = KMeans(n_clusters=n_clusters)
kmeans = kmeans.fit(encoded)
```

clustering.py hosted with ❤ by GitHub view raw

## Step-6: Summarization

Each cluster of sentence embeddings can be interpreted as a set of semantically similar sentences whose meaning can be expressed by just one candidate sentence in the summary. The candidate sentence is chosen to be the sentence whose vector representation is closest to the cluster center. Candidate sentences corresponding to each cluster are then ordered to form a summary for an email. The order of the candidate sentences in the summary is determined by the positions of the sentences in their corresponding clusters in the original email. For example, a candidate sentence is chosen as the first line in the summary if most of the sentences that lie in its cluster occur at the beginning of the email. The following lines of code implements this:

```python
from sklearn.metrics import pairwise_distances_argmin_min

```

```python
avg = []
for j in range(n_clusters):
    idx = np.where(kmeans.labels_ == j)[0]
    avg.append(np.mean(idx))
closest, _ = pairwise_distances_argmin_min(kmeans.cluster_centers_, encoded)
ordering = sorted(range(n_clusters), key=lambda k: avg[k])
summary = ' '.join([email[closest[idx]] for idx in ordering])
```

summarize.py hosted with ❤ by GitHub view raw

As this method essentially extracts some candidate sentences from the text to form a summary, it is known as **Extractive Sumarization**.

Sample summaries obtained for the above emails are down below:

*For the English email:*

```
I'm happy to hear that the issue got resolved after all and you can
now use the app in its full functionality again. Also many thanks
for your suggestions. In case you experience any further problems
with the app, please don't hesitate to contact me again.
```

*For the Danish email:*

```
Grunnet manglende dekning på deres kort for månedlig trekk, blir
dere nå overført til årlig fakturering. I morgen vil dere motta
faktura for hosting og drift av nettbutikk for perioden 05.03.2018-
05.03.2019. Ta gjerne kontakt om dere har spørsmål.
```

*For the Italian email:*

```
Apprezziamo molto che abbiate trovato il tempo per inviarci i vostri
commenti e siamo lieti che vi piaccia l'App. Sentitevi liberi di
parlare di con i vostri amici o di sostenerci lasciando una
recensione nell'App Store.
```

# Training

Pre-trained models were available for encoding English sentences (see the repository for more details). For Danish sentences however, the skip-thought model had to be trained. The data was taken from Danish Wikipedia dumps that you can get here. The .bz2 archive was extracted and the resultant .xml was parsed to strip off the html so that only the plain-text remained. There are many tools available for parsing Wikipedia dumps and none of them are perfect. They can also take a lot of time depending upon the approach used for parsing. I used the tool from here, which isn't the best available, but is free and can do the job in a reasonable amount of time. Simple pre-processing was performed on the resultant plain-text like removing newlines. By doing so, a large amount of training data was made available for the skip-thoughts model to train on for days.

The training data thus generated, consisted of 2,712,935 Danish Sentences from Wikipedia articles. The training process also requires pre-trained Wor2Vec word vectors. For this, I used the Facebook fastText's pretrained vectors (just the `wiki.da.vec` file and not `wiki.da.bin` , hence not using the vocabulary expansion feature) for Danish. The pre-trained vectors had a vocabulary size of 312,956 words. As these word vectors were also trained on Danish Wikipedia, out-of-vocabulary words were quite rare. The training code used is also available in the repository.

## Implementation Details

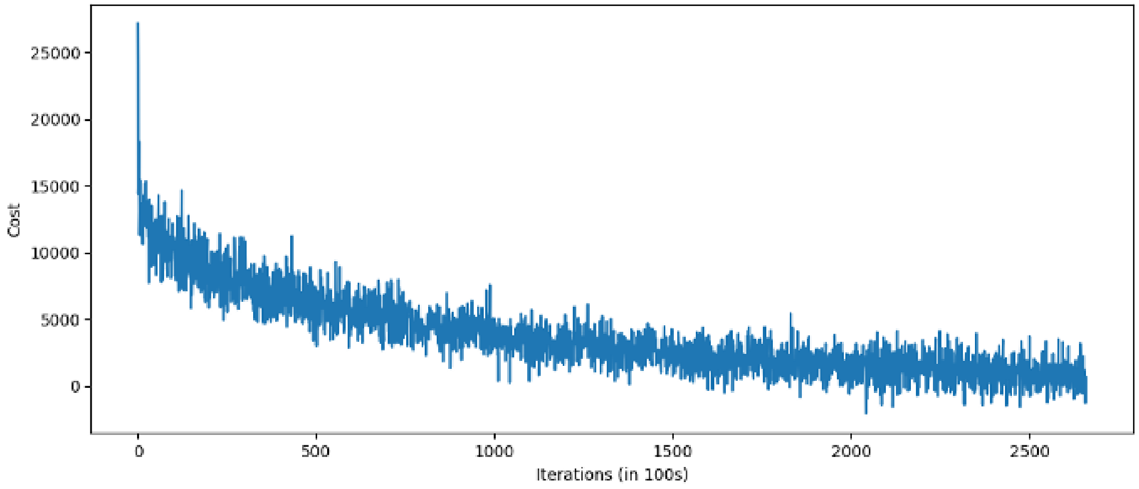
Below is a simplified version of the module which supports only English emails, but implements all of the steps mentioned above and works surprisingly well. The module along with instructions on how to run it are present in this GitHub repository for your reference. Feel free to fork and modify the code as you wish!

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Module for E-mail Summarization
*****************************************************************************
Input Parameters:
    emails: A list of strings containing the emails
Returns:
    summary: A list of strings containing the summaries.
*****************************************************************************
"""



# ***************************************************************************
import numpy as np
from talon.signature.bruteforce import extract_signature
from langdetect import detect
from nltk.tokenize import sent_tokenize
import skipthoughts
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin_min
# ***************************************************************************


def preprocess(emails):
    """
    Performs preprocessing operations such as:
        1. Removing signature lines (only English emails are supported)
        2. Removing new line characters.
    """
    n_emails = len(emails)
    for i in range(n_emails):
        email = emails[i]
        email, _ = extract_signature(email)
        lines = email.split('\n')
        for j in reversed(range(len(lines))):
            lines[j] = lines[j].strip()
            if lines[j] == '':
                lines.pop(j)
        emails[i] = ' '.join(lines)


def split_sentences(emails):
    """
    Splits the emails into individual sentences
    """
    n_emails = len(emails)
    for i in range(n_emails):
        email = emails[i]
        sentences = sent_tokenize(email)
        for j in reversed(range(len(sentences))):
            sent = sentences[j]
            sentences[j] = sent.strip()
            if sent == '':
                sentences.pop(j)
        emails[i] = sentences


def skipthought_encode(emails):
    """
    Obtains sentence embeddings for each sentence in the emails
    """
    enc_emails = [None]*len(emails)
    cum_sum_sentences = [0]
    sent_count = 0
    sent_count = 0
    for email in emails:
        sent_count += len(email)
        cum_sum_sentences.append(sent_count)

    all_sentences = [sent for email in emails for sent in email]
    print('Loading pre-trained models...')
    model = skipthoughts.load_model()
    encoder = skipthoughts.Encoder(model)
    print('Encoding sentences...')
    enc_sentences = encoder.encode(all_sentences, verbose=False)

    for i in range(len(emails)):
        begin = cum_sum_sentences[i]
        end = cum_sum_sentences[i+1]
        enc_emails[i] = enc_sentences[begin:end]
    return enc_emails


def summarize(emails):
    """
    Performs summarization of emails
    """
    n_emails = len(emails)
    summary = [None]*n_emails
    print('Preprecesing...')
    preprocess(emails)
    print('Splitting into sentences...')
    split_sentences(emails)
    print('Starting to encode...')
    enc_emails = skipthought_encode(emails)
    print('Encoding Finished')
    for i in range(n_emails):
        enc_email = enc_emails[i]
        n_clusters = int(np.ceil(len(enc_email)**0.5))
        kmeans = KMeans(n_clusters=n_clusters, random_state=0)
        kmeans = kmeans.fit(enc_email)
        avg = []
        closest = []
        for j in range(n_clusters):
            idx = np.where(kmeans.labels_ == j)[0]
            avg.append(np.mean(idx))
        closest, _ = pairwise_distances_argmin_min(kmeans.cluster_centers_,\
                                                   enc_email)
        ordering = sorted(range(n_clusters), key=lambda k: avg[k])
        summary[i] = ' '.join([emails[i][closest[idx]] for idx in ordering])
    print('Clustering Finished')
    return summary
```

## Results

- As you can notice, this method for summarization fares much better when the email consists of several sentences, instead of just 2–3 sentences. For a three sentence email, the summary will consist of two sentences, which shouldn't be the case. Also, these three sentences may be conveying entirely different things, and omitting information from any one sentence is not desirable. Extractive methods normally are not preferred for summarization of short inputs, for this very reason. Supervised Seq2Seq models are better suited for this task. However in this case, emails are generally longer in length and extractive methods work surprisingly well.

- One disadvantage of using skip-thought vectors is that the model can take a long time to train. Although acceptable results were obtained after 2–3 days of training, the Danish Skip-Thoughts model was trained for about a week. The cost fluctuated a lot during iterations since it was normalized by sentence length.



Cost vs. No. of Iterations Plot

- To see how well the Skip-Thoughts model works, we can look at most similar pairs of sentences in our dataset:

```
I can assure you that our developers are already aware of the issue
and are trying to solve it as soon as possible.
AND
I have already forwarded your problem report to our developers and
```

```
they will now investigate this issue with the login page in further
detail in order to detect the source of this problem.
--------------------------------------------------------------------
I am very sorry to hear that.
AND
We sincerely apologize for the inconvenience caused.
--------------------------------------------------------------------
Therefore, I would kindly ask you to tell me which operating system
you are using the app on.
AND
Can you specify which device you are using as well as the Android or
iOS version it currently has installed?
```

As is evident from above, the model works surprisingly well and can flag similar sentences even when the sentences largely differ in length and the vocabulary used is entirely different.

## Possible Improvements

The approach presented here works pretty good, but is not perfect. There are many improvements which can be made by increasing the model complexity:

1. Quick-Thought Vectors, a recent advancement over the Skip-Thoughts approach can cause a significant reduction in training time and better performance.

2. The skip-thought encoded representations have a dimensionality of 4800. These high dimensional vectors are not best suitable for clustering purposes because of the Curse of Dimensionality. The dimensionality of the vectors can be reduced before clustering using an Autoencoder or an LSTM-Autoencoder to impart further sequence information in the compressed representations.

3. Instead of using extractive approaches, abstractive summarization can be implemented by training a decoder network which can convert the encoded representations of the cluster centers back into natural language sentences. Such a decoder can be trained by data which can be generated by the skip-thought encoder. However, very careful hyper-parameter tuning and architecture decisions will need to be made for the decoder if we want it to generate plausible and grammatically correct sentences.

## Infrastructure Setup

All of the above experiments were performed on a n1-highmem-8 Google Cloud instance sporting an Octa-Core Intel(R) Xeon(R) CPU and an Nvidia Tesla K80 GPU with 52 GB RAM.

A special thanks to my mentor Rahul Kumar for his advice and useful suggestions along the way, without whom this would not have been possible. Also I owe a debt of gratitude to Jatana.ai for giving me this wonderful opportunity and the necessary resources to accomplish the same.

Thanks to Rahul Kumar.

Machine Learning Artificial Intelligence Deep Learning Text Summarization NLP