Unsupervised Speech Summarization

EKLAVYA PROJECT
TEAM MEMBERS:

SANKET BARHATE	RITIKA MANGLA
DHWANI PANJWANI	SUYASH GATKAL

Table of Contents:

ABSTRACT

Text summarization techniques typically employ various mechanisms to either identify highly relevant sentences in the text or remove redundant phrases/sentences. We propose to cluster sentences projected to a high dimensional vector space to identify groups of sentences that are semantically similar to each other and select representatives from these clusters to form a summary

1.INTRODUCTION

- This project aims to develop a Text Summary of speech using *Sentence Embeddings* and *Frequency distribution method.*
- Text Summarization is the process of condensing source text into a shorter version, preserving its information content and overall meaning.
- It is important to summarize text because:

    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 summarizer.

- Summarization tasks are categorized in the following types:

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

    2. Abstractive summarization- 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.

- Summarization involves the following tasks:

1. Sentence tokenization
2. Generate sentence embeddings
3. Clustering
4. Summarization

2.BACKGROUND

2.1. Natural Language Processing (NLP):

2.1.1. NLTK: The NLTK module is a massive tool kit, aimed at helping you with the entire **Natural Language Processing (NLP**) methodology. NLTK will aid you with everything from splitting sentences from paragraphs, splitting up words, recognizing the part of speech of those words, highlighting the main subjects, and then even with :helping your machine to understand what the text is all about.

For installing NLTK, use the following command in terminal/ console:

```
pip install nltk
import nltk
nltk.download()
```

2.1.2.Token - Each "entity" that is a part of whatever was split up based on rules. For examples, each word is a token when a sentence is "tokenized" into words. Each sentence can also be a token, if you tokenized the sentences out of a paragraph.

```
from nltk.tokenize import sent_tokenize, word_tokenize

EXAMPLE_TEXT = "Hello Mr. Smith, how are you doing today? The
weather is great, and Python is awesome. The sky is
pinkish-blue. You shouldn't eat cardboard."

print(sent_tokenize(EXAMPLE_TEXT))
```

The above code will output the sentences, split up into a list of sentences, which you can do things like iterate through with a for loop.

Output:

```
['Hello Mr. Smith, how are you doing today?', 'The weather is
great, and Python is awesome.', 'The sky is pinkish-blue.',
"You shouldn't eat cardboard."]
```

In the above code, use `word_tokenize` to tokenize sentences into words.

2.1.3. Stop words: The process of converting data to something a computer can understand is referred to as "pre-processing." One of the major forms of pre-processing is going to be filtering out useless data. In natural language processing, useless words (data), are referred to as stop words.

For example: (‘they', 'own', 'an', 'be', 'some', 'for', 'do', 'its', 'yours', 'such', 'into', 'of', 'most', 'itself', 'other', 'off', 'is', 's', 'am', 'or', etc.)

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

example_sent = "This is a sample sentence, showing off the stop words
filtration."

stop_words = set(stopwords.words('english'))

word_tokens = word_tokenize(example_sent)

filtered_sentence = [w for w in word_tokens if not w in stop_words]

filtered_sentence = []

for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)

print(word_tokens)
print(filtered_sentence)
```

Output:

```
['This', 'is', 'a', 'sample', 'sentence', ',', 'showing', 'off',
'the', 'stop', 'words', 'filtration', '.']
['This', 'sample', 'sentence', ',', 'showing', 'stop', 'words',
'filtration', '.']
```

2.1.4.Stemming: It is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma.

For example:

Words: ran, running, run. Stem word: Run.

Words: ridden,ride. Stem word: Ride.

Note: In linguistics and **NLP**, **corpus** (literally Latin for body) refers to a collection of texts. Such collections may be formed of a single language of texts, or can span multiple languages -- there are numerous reasons for which multilingual corpora (the plural of **corpus**) may be useful.

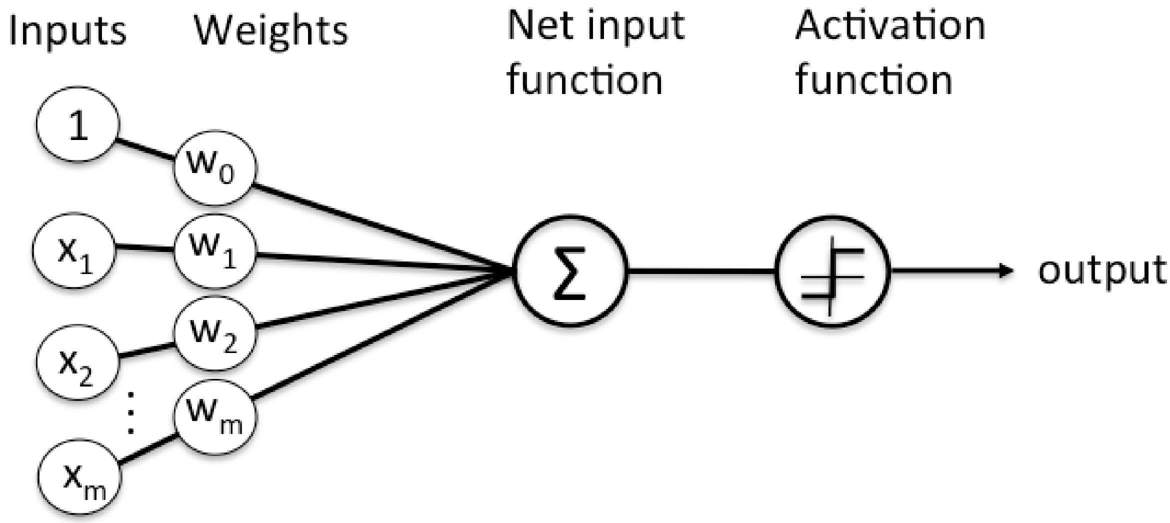
2.2.Neural Networks: Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Neural networks help us cluster and classify. You can think of them as a clustering and classification layer on top of the data you store and manage. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on. (Neural networks can also extract features that are fed to other algorithms for clustering and classification; so you can think of deep neural networks as components of larger machine-learning applications involving algorithms for reinforcement learning, classification and regression.)

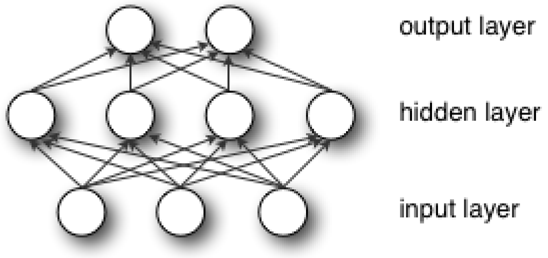
2.2.1. Neural Networks elements: Deep learning is the name we use for “stacked neural networks”; that is, networks composed of several layers.

The layers are made of ***nodes.*** A node is just a place where computation happens, loosely patterned on a neuron in the human brain, which fires when it encounters sufficient stimuli. A node combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input, thereby assigning significance to inputs with regard to the task the algorithm is trying to learn; e.g. which input is most helpful is classifying data without error? These input-weight products are summed and then the sum is passed through a node’s so-called ***activation function***, to determine whether and to what extent that signal should progress further through the network to affect the ultimate outcome, say, an act of classification. If the signals passes through, the neuron has been "activated."

Here's a diagram of what one node might look like.



A node layer is a row of those neuron-like switches that turn on or off as the input is fed through the net. Each layer's output is simultaneously the subsequent layer's input, starting from an initial input layer receiving your data.



Pairing the model's adjustable weights with input features is how we assign significance to those features with regard to how the neural network classifies and clusters input.

2.2.2. Gradient Descent

The name for one commonly used optimization function that adjusts weights according to the error they caused is called "gradient descent."

Gradient is another word for slope, and slope, in its typical form on an x-y graph, represents how two variables relate to each other: rise over run, the change in money over the change in time, etc. In this particular case, the slope we care about describes the relationship between the network's error and a single weight; i.e. that is, how does the error vary as the weight is adjusted.

As a neural network learns, it slowly adjusts many weights so that they can map signal to meaning correctly. The relationship between network *Error* and each of those *weights* is a derivative, *dE/dw*, that measures the degree to which a slight change in a weight causes a slight change in the error.

Each weight is just one factor in a deep network that involves many transforms; the signal of the weight passes through activations and sums over several layers, so we use the chain rule of calculus to march back through the networks activations and outputs and finally arrive at the weight in question, and its relationship to overall error.

The chain rule in calculus states that

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

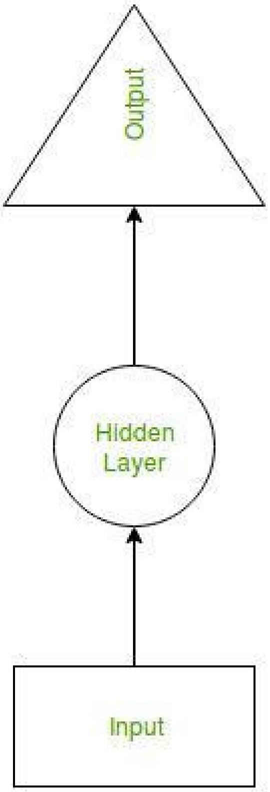
In a feedforward network, the relationship between the net's error and a single weight will look something like this:

$$\frac{dError}{dweight} = \frac{dError}{dactivation} * \frac{dactivation}{dweight}$$

That is, given two variables, *Error* and *weight*, that are mediated by a third variable, *activation*, through which the weight is passed, you can calculate how a change in *weight* affects a change in *Error* by first calculating how a change in *activation*affects a change in *Error*, and how a change in *weight* affects a change in *activation*.

2.2.3.Recurrent Neural Network

Recurrent Neural Network(RNN) are a type of neural network where the **output from previous step are fed as input to the current step**. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is **Hidden state**, which remembers some information



about a sequence.
RNN have a **“memory”** which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

- **Formula for calculating current state:**

  where:

  h_t -> current state
  h_{t-1} -> previous state
  x_t -> input state

- **Formula for applying Activation function(tanh):**

  where:

  w_{hh} -> weight at recurrent neuron
  w_{xh} -> weight at input neuron

- **Formula for calculating output:**

Y_t -> output
W_{hy} -> weight at output layer

Training through RNN

1. A single time step of the input is provided to the network.
2. Then calculate its current state using set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

2.2.4.Loss Function:

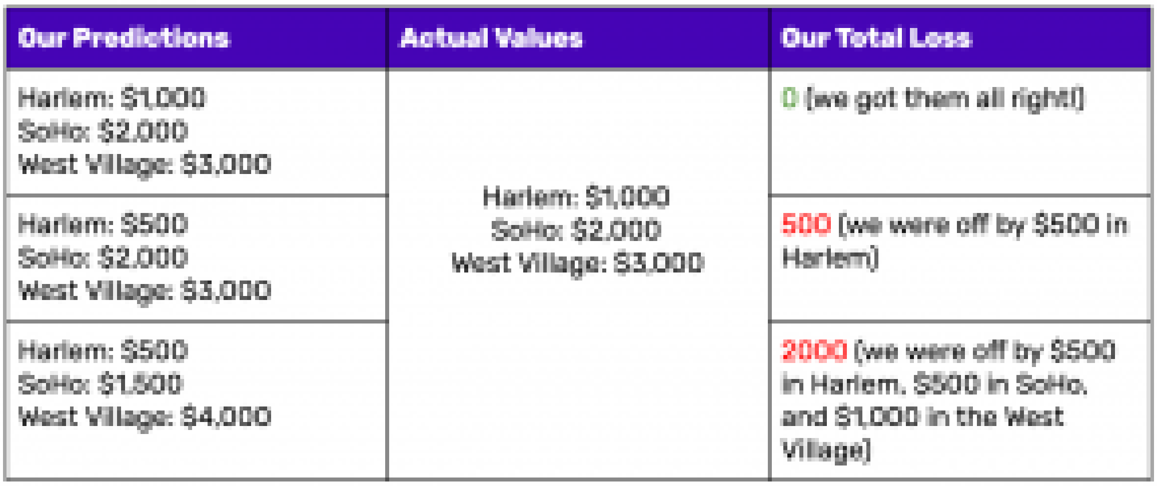
The loss function is the bread and butter of modern machine learning; it takes your algorithm from theoretical to practical and transforms neural networks from glorified matrix multiplication into *deep learning*.

This post will explain the role of loss functions and how they work, while surveying a few of the most popular from the past decade..

What's a Loss Function?

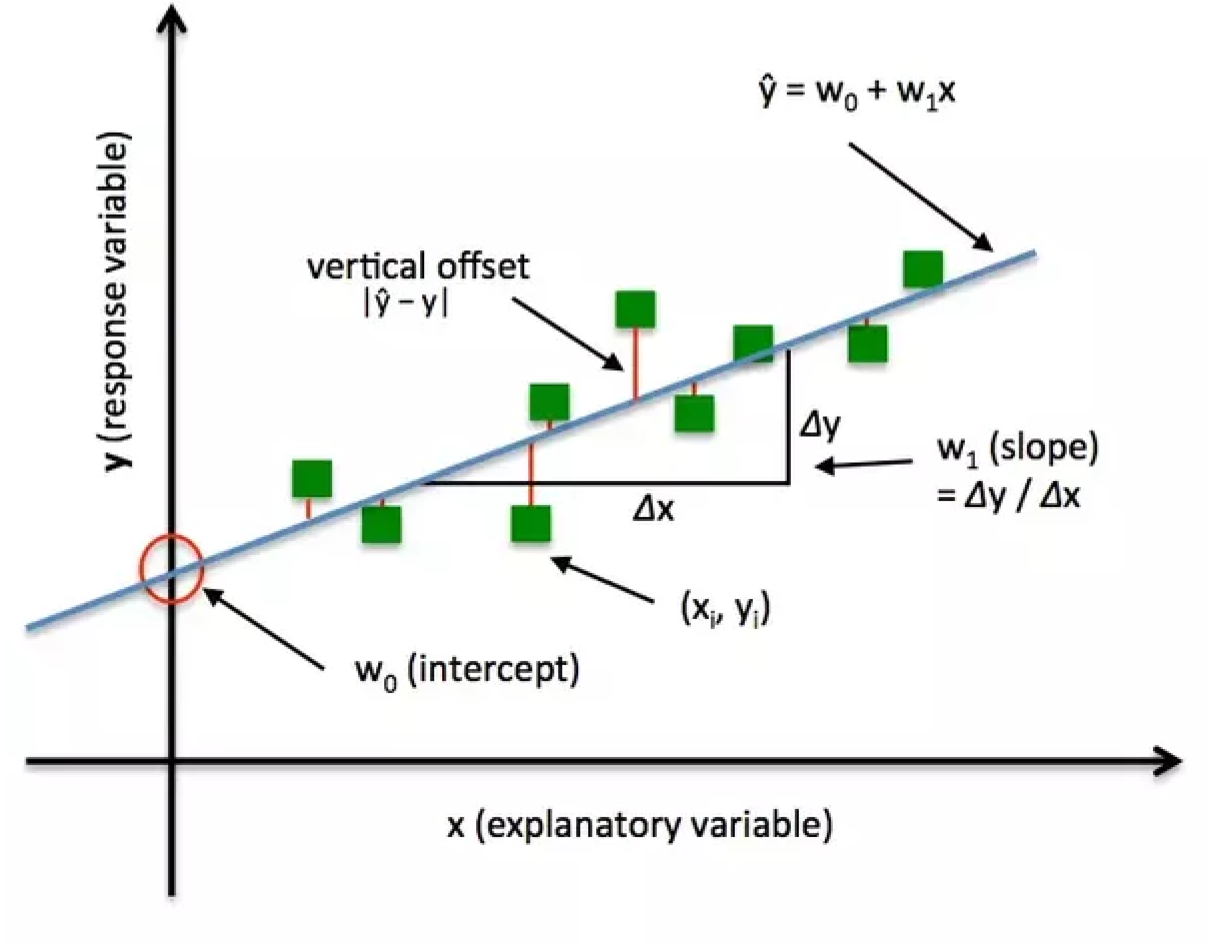
At its core, a loss function is incredibly simple: it's a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you're getting anywhere.

In fact, we can design our own (very) basic loss function to further explain how it works. For each prediction that we make, our loss function will simply measure the absolute difference between our prediction and the actual value. In mathematical notation, it might look something like *abs(y_predicted – y).* Here's what some situations might look like if we were trying to predict how expensive the rent is in some NYC apartments:



Our Predictions	Actual Values	Our Total Loss
Harlem: $1,000 SoHo: $2,000 West Village: $3,000	Harlem: $1,000 SoHo: $2,000 West Village: $3,000	0 (we got them all right!)
Harlem: $500 SoHo: $2,000 West Village: $3,000		500 (we were off by $500 in Harlem)
Harlem: $500 SoHo: $1,500 West Village: $4,000		2000 (we were off by $500 in Harlem, $500 in SoHo, and $1,000 in the West Village)

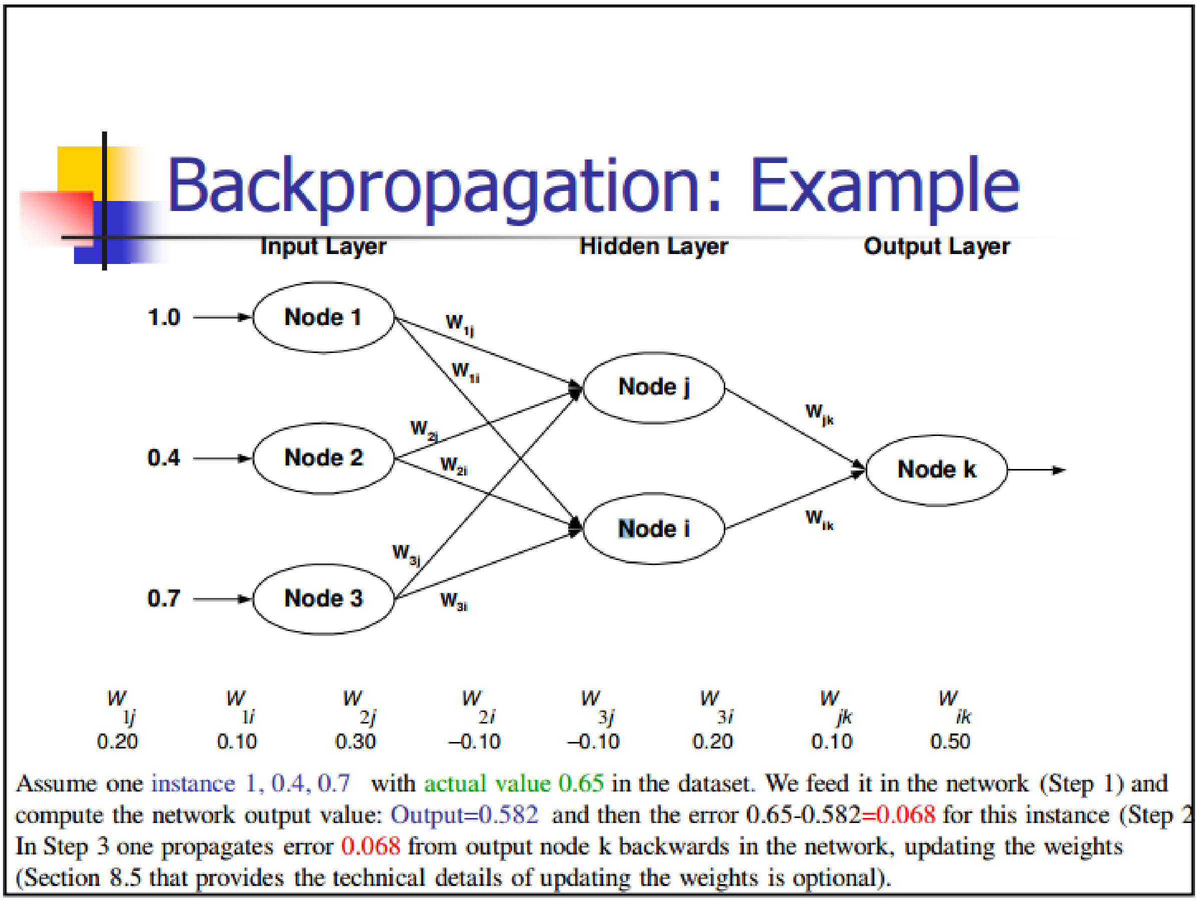
Notice how in the loss function *we* defined, it doesn't matter if our predictions were too high or too low. All that matters is how incorrect we were, directionally agnostic. This is *not* a feature of all loss functions: in fact, your loss function will vary significantly based on the domain and unique context of the problem that you're applying machine learning to. In your project, it may be much worse to guess too high than to guess too low, and the loss function you select must reflect that.



2.2.5.Back Propogation:

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.

A diagrammatical explanation of back propagation is given below.



Backpropagation: Example

w_{1j}	w_{1i}	w_{2j}	w_{2i}	w_{3j}	w_{3i}	w_{jk}	w_{ik}
0.20	0.10	0.30	–0.10	–0.10	0.20	0.10	0.50

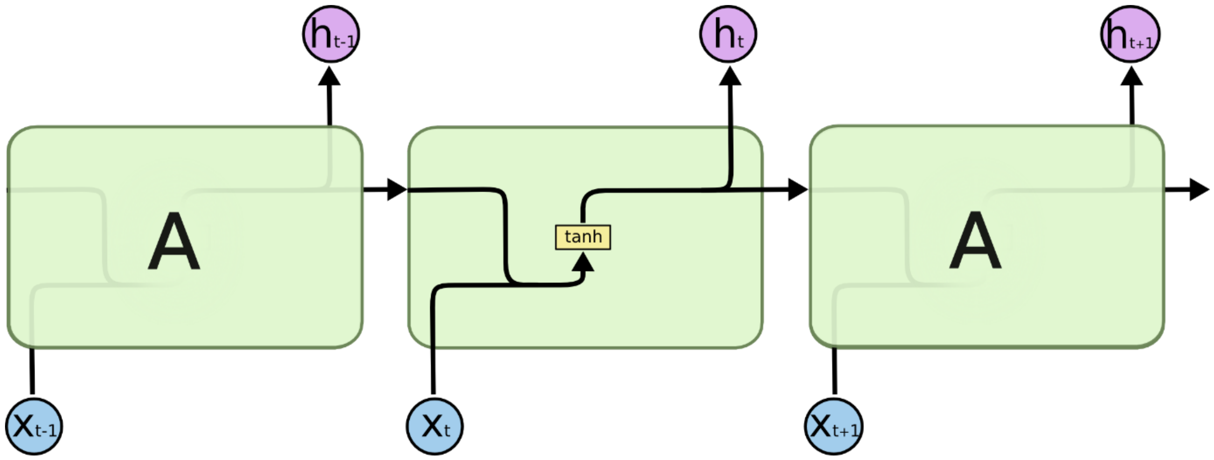
Assume one instance 1, 0.4, 0.7 with actual value 0.65 in the dataset. We feed it in the network (Step 1) and compute the network output value: Output=0.582 and then the error 0.65-0.582=0.068 for this instance (Step 2 In Step 3 one propagates error 0.068 from output node k backwards in the network, updating the weights (Section 8.5 that provides the technical details of updating the weights is optional).

2.2.6.LSTM:

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1] They work tremendously well on a large variety of problems, and are now widely used.

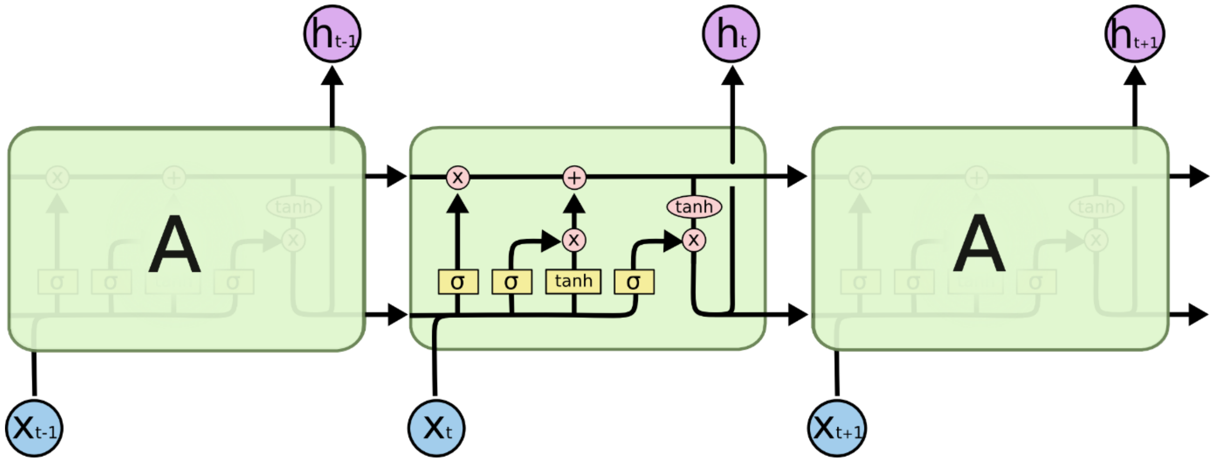
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



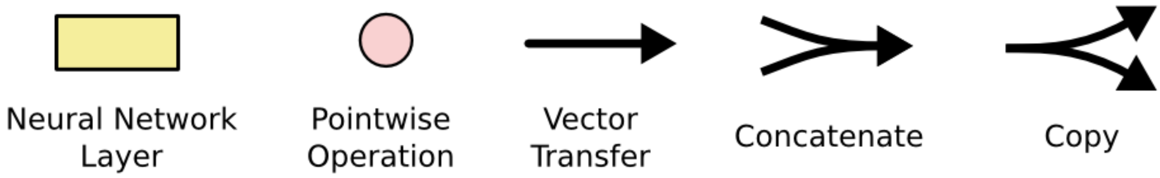
The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

Let us go over a diagram to understand this.



In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

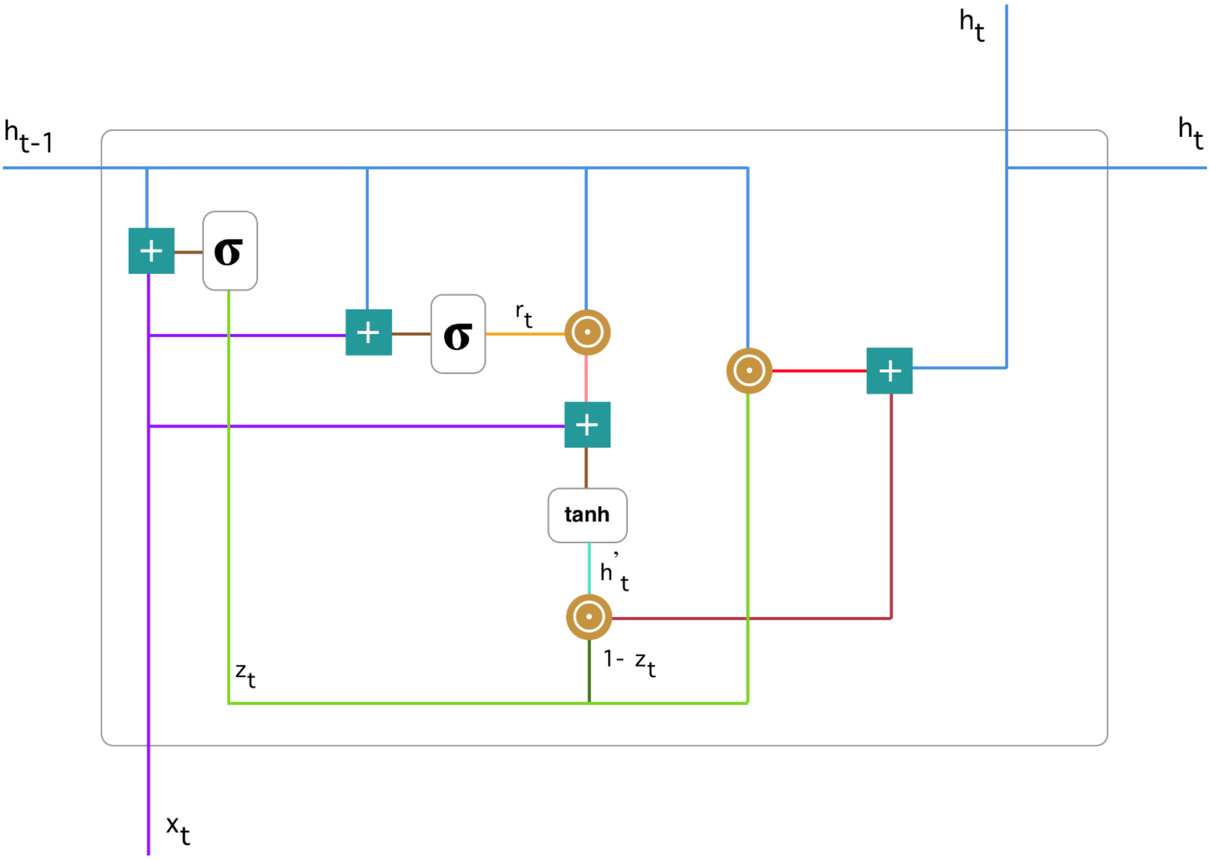
2.2.7.GRU:

Gated Recurrent Units (GRU) are a variation on Long Short Term Memory (LSTM) recurrent neural networks. Both LSTM and GRU networks have additional parameters that control when and how their memory is updated.

Both GRU and LSTM networks can capture both long and short term dependencies in sequences, but GRU networks involve less parameters and so are faster to train.

Conceptually, a GRU network has a reset and forget "gate" that helps ensure its memory doesn't get taken over by tracking short term dependencies. The network learns how to use its gates to protect its memory so that it's able to make longer term predictions.

A diagrammatic explanation of GRU is:



3.Text Summarization using Sentence Scoring Method:

This type of summarization is uses following steps:

1. Tokenize words and remove stop words.
2. Create frequency table of words - how many times each word appears in the text
3. Assign score to each sentence depending on the words it contains and the frequency table
4. Build summary by adding every sentence above a certain score threshold

Libraries used :

```
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize, sent_tokenize
```

Removing stop words and making frequency table:

First, we create three arrays -one for stop words, and one for every word in the body of text and one for stemmer.
Second, we create a dictionary for the word frequency table. For this, we should only use the words that are not part of the stopWords array.

```
stemmer = SnowballStemmer("english")
stopWords = set(stopwords.words("english"))
words = word_tokenize(text)

freqTable = dict()
for word in words:
    word = word.lower()
    if word in stopWords:
        continue

    word = stemmer.stem(word)

    if word in freqTable:
        freqTable[word] += 1
    else:
        freqTable[word] = 1
```

Assigning a score to every sentence:

Create array of sentences using sent_tokenize and define a dictionary to keep the score of each sentence.

```
sentences = sent_tokenize(text)
sentenceValue = dict()
```

Now go through every sentence and give it a score depending on the words it has (adding the frequency of every non-stop word in a sentence).

```
for sentence in sentences:
    for word, freq in freqTable.items():
        if word in sentence.lower():
            if sentence in sentenceValue:
                sentenceValue[sentence] += freq
            else:
                sentenceValue[sentence] = freq
```

Note: A potential issue with our score algorithm is that long sentences will have an advantage over short sentences. To solve this, divide every sentence score by the number of words in the sentence.

Now find the average score of a sentence.

```
sumValues = 0
for sentence in sentenceValue:
  sumValues += sentenceValue[sentence]
average = int(sumValues/ len(sentenceValue))
```

This average score is used to create a threshold value for sentences to added in summary.

```
summary = ' '
for sentence in sentences:
    if sentence in sentenceValue and sentenceValue[sentence] > (1.2 * average):
      summary += " " + sentence
```

4.Text Summarization using Sentence Embeddings Method:

Introduction:

We propose an unsupervised text summarization approach by clustering sentence embeddings trained to embed paraphrases near each other. Clusters of sentences are then converted to a summary by selecting a representative from each cluster.When representing sentences in a high-dimensional vector space, the goal is typically to directly or indirectly embed sentences such that sentences close in meaning are embedded near each other in the vector space. Thus, sentences that form a cluster in the vector space are likely to be close in meaning to each other. We exploit this assumption to perform summarization.

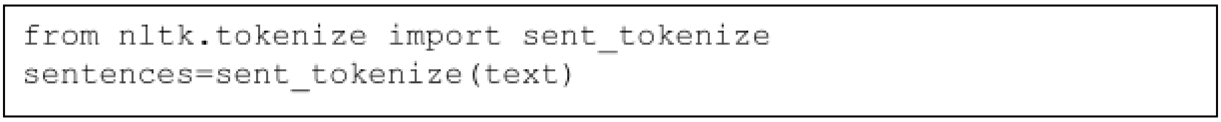
Step 1: Sentence Tokenization

Natural Language toolkit has very important module **tokenize** which further compromises of sub-modules:

1. word tokenize
2. sentence tokenize

After converting speech to text, we will split the text into sentences based on specific rules of sentence delimiters.

The code for this is:



```
from nltk.tokenize import sent_tokenize
sentences=sent_tokenize(text)
```

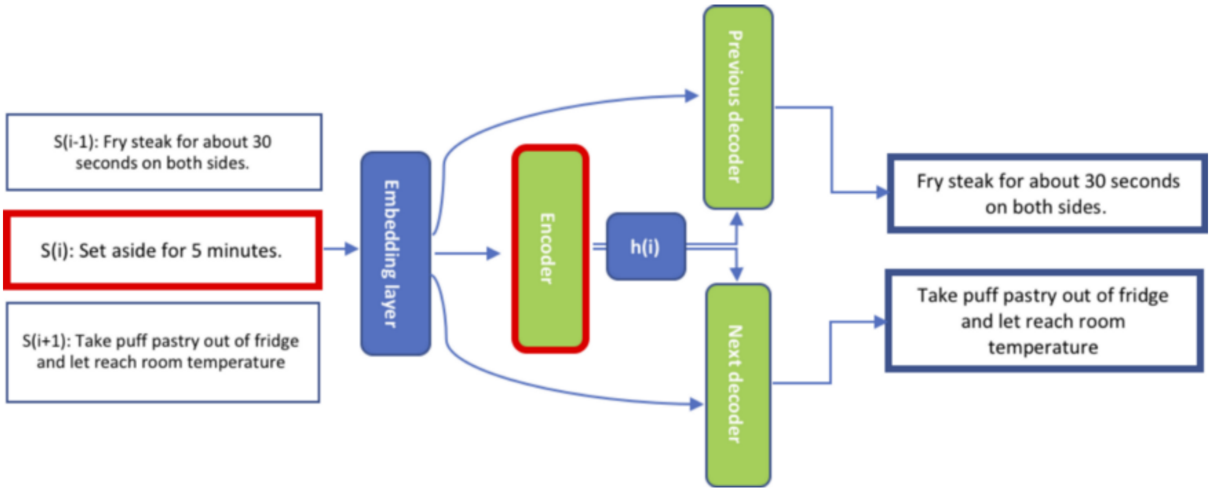
Step 2: Create Sentence Embeddings

This work aims to learn how to combine word embeddings to obtain sentence embeddings that satisfy the property that sentences that are paraphrases of each other are embedded near each other in the vector space. This is done in a supervised manner using known paraphrases. The authors compare different technqiues for combining word embeddings and test the learned embeddings on prediction of textual similarity and entailment, and in sentiment classification. They find that averaging word embeddings learned in a supervised manner performs best for prediction of textual similarity and entailment.

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.

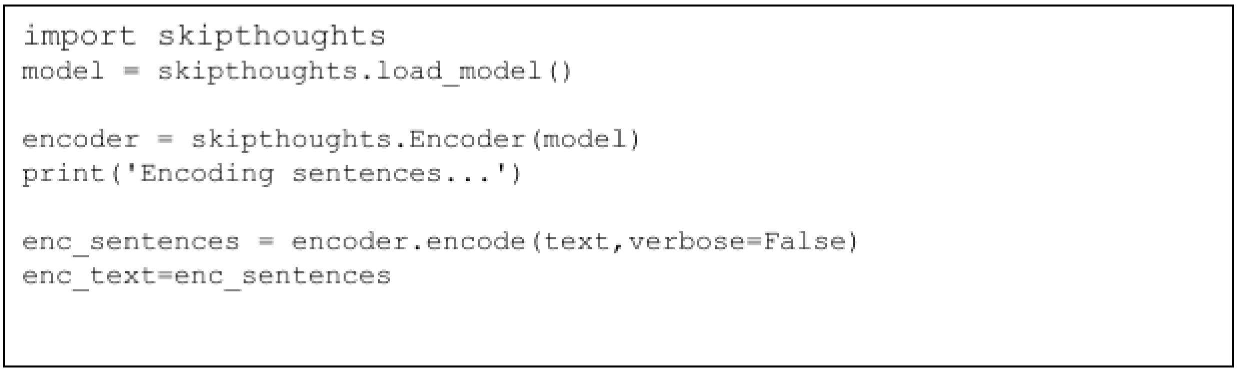
However, these unsupervised methods do not take the sequence of words in the sentence into account. This may incur undesirable losses in model performance. To overcome this, I chose to instead train a Skip-Thought sentence encoder in a supervised manner using Wikipedia dumps as training data. The Skip-Thoughts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

The code for this is as follows:



```
import skipthoughts
model = skipthoughts.load_model()

encoder = skipthoughts.Encoder(model)
print('Encoding sentences...')

enc_sentences = encoder.encode(text,verbose=False)
enc_text=enc_sentences
```

Step 3: Clustering

K-means clustering is a clustering algorithm that aims to partition n observations into k clusters.

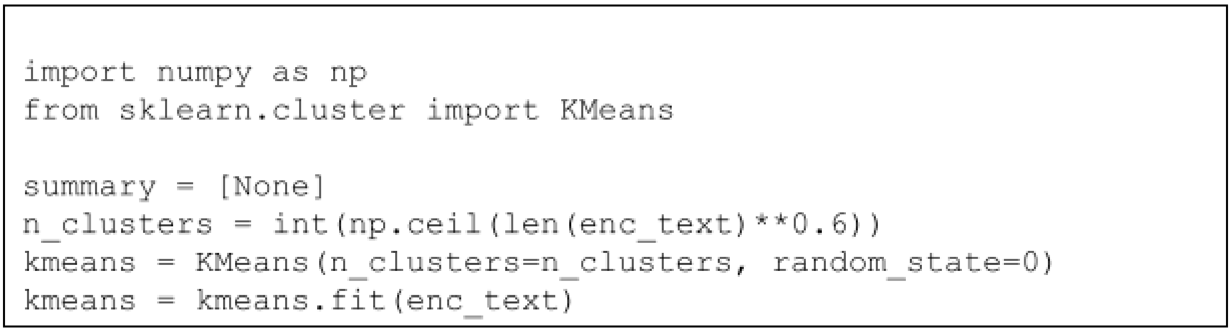
There are 3 steps:

- Initialisation – K initial "means" (centroids) are generated at random
- Assignment – K clusters are created by associating each observation with the nearest centroid
- Update – The centroid of the clusters becomes the new mean

Assignment and Update are repeated iteratively until convergence

The end result is that the sum of squared errors is minimised between points and their respective centroids.

In this code , we chose the total number of sentences in the summary to be equal to the square root of the maximum number of sentences in the document.The fit function is used to compute the k-means clustering.



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

summary = [None]
n_clusters = int(np.ceil(len(enc_text)**0.6))
kmeans = KMeans(n_clusters=n_clusters, random_state=0)
kmeans = kmeans.fit(enc_text)
```

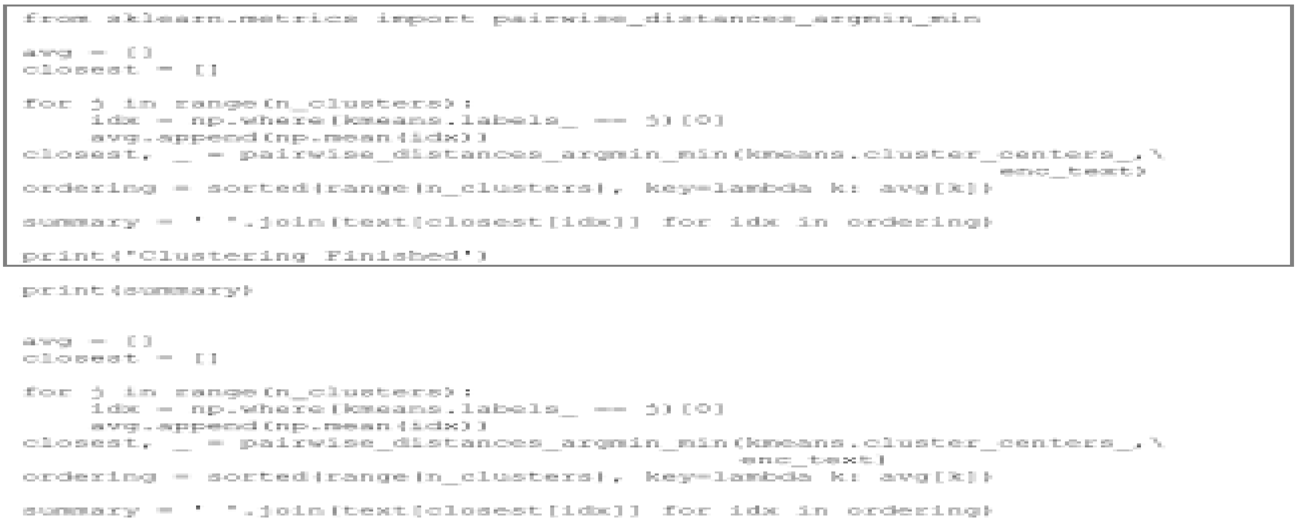
The code for the above process is:

Step 4: Summarization

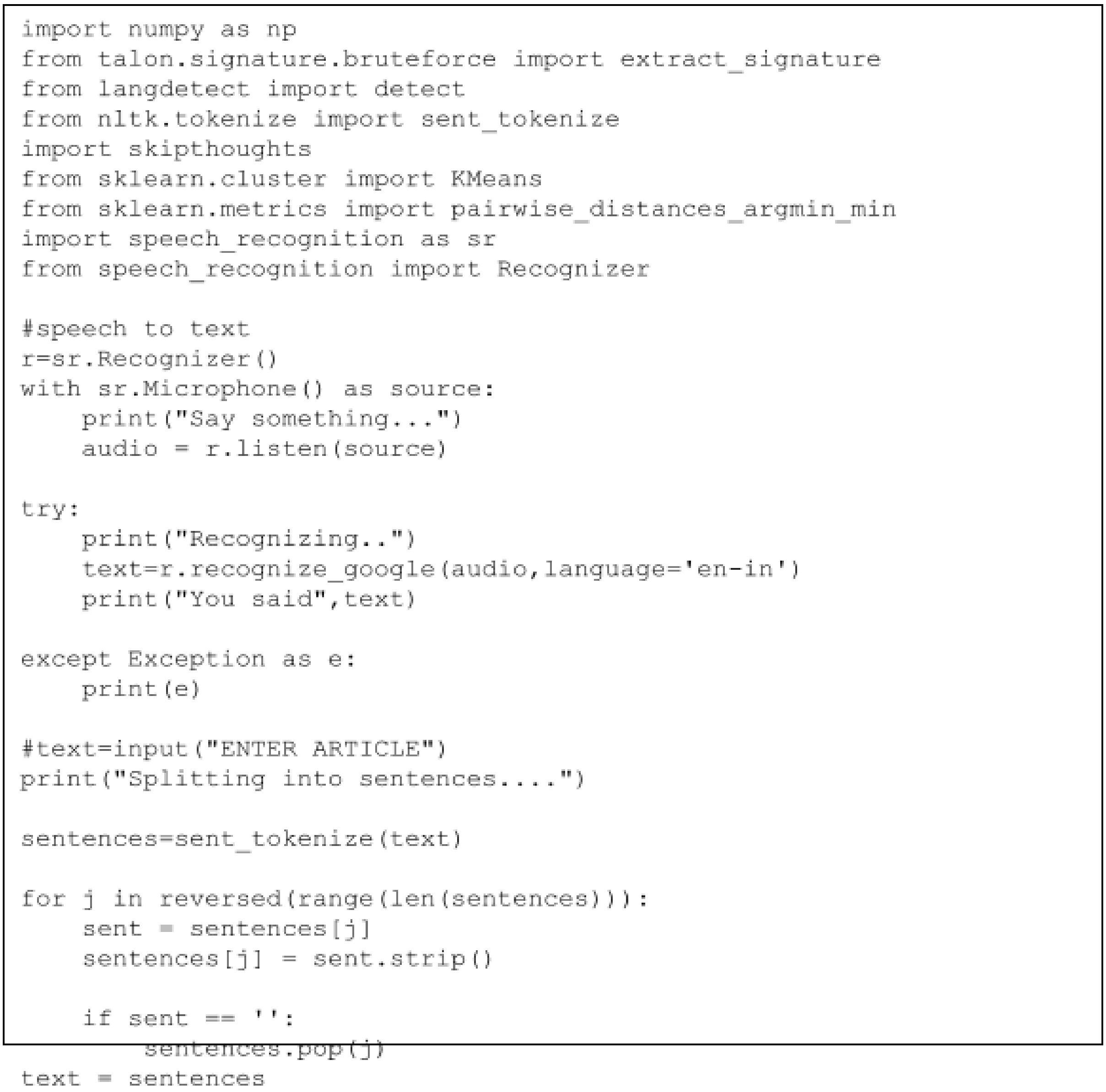
Clustering is an unsupervised machine learning approach, but it be used to improve the accuracy of supervised machine learning algorithms as well by clustering the data points into similar groups and using these cluster labels as independent variables in the supervised machine learning algorithm.

Each cluster of sentence embeddings can be interpreted as a set of semantically similar sentences whose meaning can be expressed by just one candidate. 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 the document.The order of the sentences in the summary is determined by the position of the sentences in the cluster.

The code that implements this is:



```
from sklearn.metrics import pairwise_distances_argmin_min

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_text)
ordering = sorted(range(n_clusters), key=lambda k: avg[k])

summary = ' '.join(text[closest[idx]] for idx in ordering)

print('Clustering Finished')

print(summary)


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_text)
ordering = sorted(range(n_clusters), key=lambda k: avg[k])

summary = ' '.join(text[closest[idx]] for idx in ordering)
```



```
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
import speech_recognition as sr
from speech_recognition import Recognizer

#speech to text
r=sr.Recognizer()
with sr.Microphone() as source:
    print("Say something...")
    audio = r.listen(source)

try:
    print("Recognizing..")
    text=r.recognize_google(audio,language='en-in')
    print("You said",text)

except Exception as e:
    print(e)

#text=input("ENTER ARTICLE")
print("Splitting into sentences....")

sentences=sent_tokenize(text)

for j in reversed(range(len(sentences))):
    sent = sentences[j]
    sentences[j] = sent.strip()

    if sent == '':
        sentences.pop(j)
text = sentences
```

The entire code for unsupervised text summarization using word embeddings is**:**

```
enc_text=[None]

#all_sentences=sent_tokenize(text) #[sent for sent in text]


print('Loading pre-trained models...')
model = skipthoughts.load_model()

encoder = skipthoughts.Encoder(model)
print('Encoding sentences...')

enc_sentences = encoder.encode(text,verbose=False)
enc_text=enc_sentences

summary = [None]

n_clusters = int(np.ceil(len(enc_text)**0.6))

kmeans = KMeans(n_clusters=n_clusters, random_state=0)
kmeans = kmeans.fit(enc_text)

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_text)
ordering = sorted(range(n_clusters), key=lambda k: avg[k])

summary = ' '.join(text[closest[idx]] for idx in ordering)

print('Clustering Finished')

print(summary)

f= open("SUMMARY","w+")
f.write(summary)
f.close()
```

5.Google Cloud

As our project curtails to translating an ongoing conversation into text we decided to use Google Cloud API.

5.1.Using google cloud for Speech-to-text:

Features:

1. Automatic Speech Recognition
2. Automatic Punctuation
3. Speaker Diarization
4. Real-time Streaming or pre-recorded Audio Support
5. Auto-Detect Language

5.2.How to use google speech-to-text API?

Using Client libraries:

Cloud Speech-to-Text enables easy integration of Google speech recognition technologies into developer applications. You can send audio data to the Speech-to-Text API, which then returns a text transcription of that audio file.

1.Set up a GCP Console project.

- Login to your google cloud console
- Create or select a project.
- Enable the Google Speech-to-Text API for that project.
- Create a service account.
- Download a private key as JSON.

2. Set the environment variable **GOOGLE_APPLICATION_CREDENTIALS** to the file path of the JSON file that contains your service account key. This variable only applies to your current shell session, so if you open a new session, set the variable again.

Replace **PATH** with the file path of the JSON file that contains your service account key.

For linux:

```
export GOOGLE_APPLICATION_CREDENTIALS="PATH"
```

For windows:

(In command prompt)

```
set GOOGLE_APPLICATION_CREDENTIALS=PATH
```

3.Install the client library

```
pip install --upgrade google-cloud-speech
```

Google How to guide for further information and programs:
https://cloud.google.com/speech-to-text/docs/how-to

Activating google cloud :

To activate google cloud credit/debit card details are required. After providing details, user gets credit of $300 which can be used to access google cloud platform.

6.Advancements:

- Multiple language detection.
- Advancements in abstractive summarization approach.
- Feature of time and duration based history of conversations.

7.References:

- **NLP Tutorials:**
  https://www.youtube.com/watch?v=FLZvOKSCkxY&list=PLQVvvaa0QuDf2JswnfiGkliBInZnIC4HL
  https://pythonprogramming.net/tokenizing-words-sentences-nltk-tutorial/
- **Basics of Neural networks:**
  https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- **Detailed course on deep learning:**
  https://www.coursera.org/specializations/deep-learning?
- **Reference to a similar module:**
  https://medium.com/jatana/unsupervised-text-summarization-using-sentence-embeddings-adb15ce83db1

8. Git-Hub Link:

https://github.com/DhwaniPanjwani/USS