Comparison of Neural Network based Approach and ML, NLP based Algorithms for Keyword Extraction from Short Text

Sarvesh Kaushik, Dr. Thenmozhi T.
School of Computer Science and Engineering,
Vellore Institute of Technology,
Vellore, Tamil Nadu, India

Abstract:- Large amounts of text data is generated on a daily basis through social media posts, reviews, emails, blogs and search queries etc. Most of this text data is unstructured. To help make sense of this large amount of text data we need keyword extraction which helps in obtaining the important word(keywords) or important phrases(key phrases) without having to go through all the text data ourselves.

However over the years it has been found to be quite difficult to extract keywords from short text (text spanning across one or maybe two sentences) and many of the traditional methods such as classification, RAKE, TextRank and TF-IDF have been found to be not as effective as we would wish them to be. In this paper, we compare the traditional methods and also propose an new Neural Network based algorithm, such as the sequence to sequence based encoder-decoder model which we show in this paper, for better keyword extraction form short text. We conduct some preliminary application based investigation on some sentences, which show the superiority of neural network method and can also form the basis of future research.

Keywords:- *extraction, text, data, short text, neural network, NLP.*

I. INTRODUCTION

Since the dawn of Data Science the topic of language processing and text classification, keyword extraction have been of extreme interest and study because of their usefulness and because they have immensely important secondary uses in apps and websites, which use NLP and keyword extraction for the working of their application. Examples include but are not limited to Twitter, Reddit, Instagram, Facebook etc .Apart from companies these keyword extraction methods help fuel a lot of research and industries.

With the increase in amount of textual data being generated every year, it has become more important to find important pieces of text and difficult to find the important information and ignore that which is of less use.

The process of finding or differentiating the most important words in a corpus, document, paragraph or sentence, that helps us in gaining information about the text in a more efficient way is keyword extraction. It helps in recognizing the main topics or summarize the textual content. Keywords extraction is a critical issue in many Natural Language Processing (NLP) applications and can improve the performance of many NLP systems. Since most of the data generated(more than 80 percent) is unstructured, it becomes difficult to analyze it efficiently. It helps in overcoming inconsistencies, and helps reduce the time that would have otherwise gone in going through vast amount of textual data, while trying to find meaning in social media posts, articles etc., in real time .Now the question appears how are we to extract keywords from a short text, since it has limited context and doesn't necessarily follow the syntax of written language There are many different approaches to keyword extraction, from Statistical approaches such as TF-IDF, RAKE, to Graphical approaches such as TextRank and then the newer versions which we wish to introduce are neural network based such as our Bidirectional Encoder Decoder Model.

II. LITERATURE SURVEY

A. Statistical Approaches

TF-IDF: Term Frequency – Inverse Document Frequency is a frequency based statistical approach in which we assess the importance of a word in particular textual data whether it be a corpus, document or a paragraph. In this the value of a word is corresponds to the number of times it appears, which is then counterbalanced by the number of documents that contain the word. Hence extremely common words lose significance since they appear frequently in many documents.

TF – Term Frequency is related to the number of times a particular word appears in a document

IDF- Inverse Document Frequency is related to the number of documents that contain a certain word, i.e. how common a word is in the corpus.

The algorithm helps decode the textual data by creating a vector corresponding to each word.

However the algorithm is not perfect and does have certain setbacks, the problem with the statistical approach is that it heavily relies on frequency and hence in this case the ind of corpus that is available influences the keyword extraction process negatively more so in the case of keyword extraction from short text. Also considering that most applications that will want to extract keyword extraction from short text will have short text that is generated at random by the user and is unpredictable, hence finding the corpus that will neither be biased and will be varied enough that it can handle the short text is difficult.TF-IDF is based on the bag-of-words (BoW) model, therefore it does not capture position in text, semantics, co-occurrences in different documents, etc. For this reason, TF-IDF is only useful as a lexical level feature Cannot capture semantics (e.g., as compared to topic models, word embeddings) Has no way to account for the context, i.e. The words preceding and succeeding the keyword. Hence the position in text and the context of use could be accounted for in some future research.

RAKE: Rapid Automatic Keyword Extraction is a well-known keyword extraction method which uses a list of stop words and phrase delimiters to detect the most relevant words or phrases in a piece of text. The first thing this method does is split the text into a list of words and remove stop words from that list. This returns a list of what is known as content words. Then, the algorithm splits the text at phrase delimiters and stop words to create candidate expressions. Once the text has been split, the algorithm creates a matrix of word co-occurrences. Each row shows the number of times that a given content word co-occurs with every other content word in the candidate phrases. After that matrix is built, words are given a score. That score can be calculated as the degree of a word in the matrix as the word frequency or as the degree of the word divided by its frequency. Based on the magnitude of this above obtained 'score' keywords are extracted.

In the context of short text there are some disadvantages to RAKE however. It doesn't have the context of outside word usage. If you don't have a comprehensive list of stop words the phrases can get quite long and less useful. It ends up with a lot more filler words and adverbs which is not a good thing in general, but could be quite useful in some cases. Does not work well for short text as it heavily relies on frequency.

B. Graphical Approach

TextRank: The TextRank algorithm is a graph-based algorithm, where the primary data model being used for it is a graph. Words in our input text represent nodes in the graph. Similarity scores between the words represent edges inside the graph. Two nodes are connected to each other by an edge. The two nodes represent two words from the text while the edge between them represents how similar the two words are. We first split our original document into words/phrases. Then calculate word embeddings using a vector representation algorithm. Compute similarity scores between every two nodes. Build the graph using the rules described above. Get top-n results from the graph as the most important n keywords

The disadvantage of TextRank is that it omits the keywords which have a lower chance to appear though being meaningful in context. This drawback usually stems from the graph-based nature of the technique that tends to ignore the linguistic similarity among words. For the TextRank algorithm to yield optimal results, the input data should be of a significant size so that the similarity between words can be effectively computed and its measure is meaningful. Hence this technique is impractical in our case when the size of our document is just a few words (i.e., One or two sentences).

III. PROPOSED METHODOLOGY

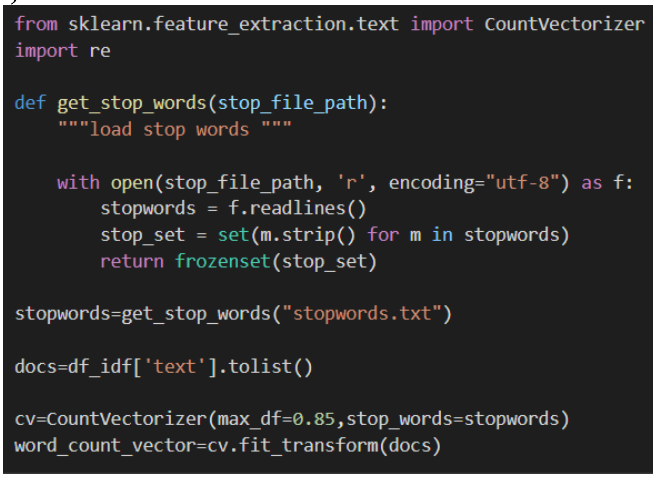
Bidirectional Encoder Decoder Model: The main motive for going with a neural network based model for extracting the relevant keywords rather than traditional ML, NLP based algorithms such as TextRank, TF-IDF, etc., is that these algorithms are mainly found suitable for longer pieces of text because of their high dependency on factors such as frequency of a particular word or value of a word in relation to its neighbors .Also the performance of these methods is influenced by the feature selection and manually defined rules. Since here we are dealing with comparatively shorter text these conventional methods are not effective. Therefore a sequence to sequence neural network based methodology was used where the keywords were assumed to be a separate language and an encoder-decoder model was trained to convert english sentences to keywords .Our encoder uses a Bidirectional Encoder Representations from Transformers model. It accepts a single element of the input sequence at each step, processes it ,performs tokenization, numericalization and forms word embeddings, collects information about that element and propagates it forward. It returns an intermediate vector that contains information about the entire input sequence to help the decoder make accurate predictions. The information contained in the input text is basically encapsulated as internal state vectors (or tensors) by the encoder and this intermediate output is then passed on to the decoder part of our model. The decoder is given the entire sentence in english, it predicts the output at each step. It generates the output phrase , containing the keywords, word by word, for which it must recognize the starting and end of each sentence. The initial states of the decoder are set to the final states of the encoder as the decoder is trained to generate the output based on the information gathered by the encoder. A forcing technique is used so that the input at each step is the actual output and not the predicted output from the last step. The loss is lastly calculated on the predicted output and the errors are back propagated to update the model weights.

IV. BASIC IMPLEMENTATION OF THE DIFFERENT ALGORITHMS

Important point to note that before applying all the algorithms the text preprocessing i.e. uploading dataset, tokenization ,lemmatization and removal of stop words was already done.

A. TF-IDF

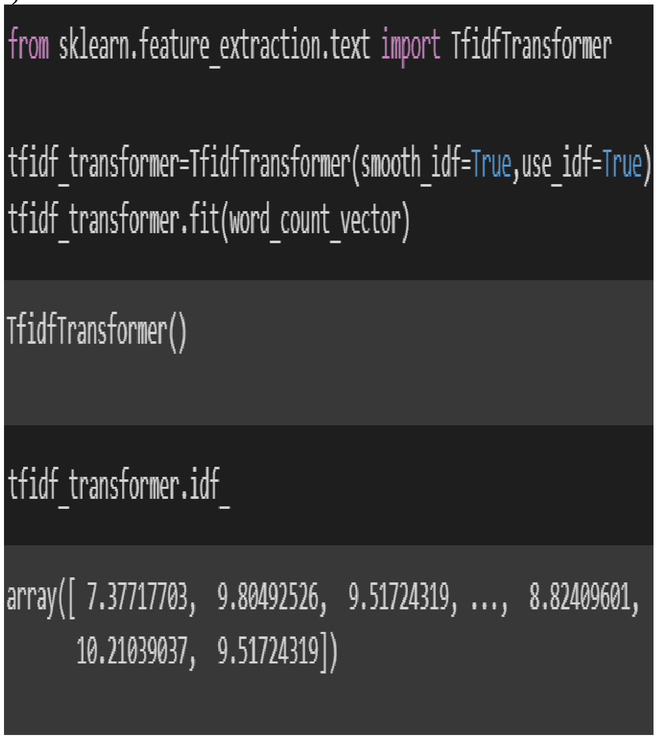
a)



```
from sklearn.feature_extraction.text import CountVectorizer
import re

def get_stop_words(stop_file_path):
    """load stop words """

    with open(stop_file_path, 'r', encoding="utf-8") as f:
        stopwords = f.readlines()
        stop_set = set(m.strip() for m in stopwords)
        return frozenset(stop_set)

stopwords=get_stop_words("stopwords.txt")

docs=df_idf['text'].tolist()

cv=CountVectorizer(max_df=0.85,stop_words=stopwords)
word_count_vector=cv.fit_transform(docs)
```

Fig. 1: a) Creating IDF to create vocabulary and generate word count

b)



```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer=TfidfTransformer(smooth_idf=True,use_idf=True)
tfidf_transformer.fit(word_count_vector)
```

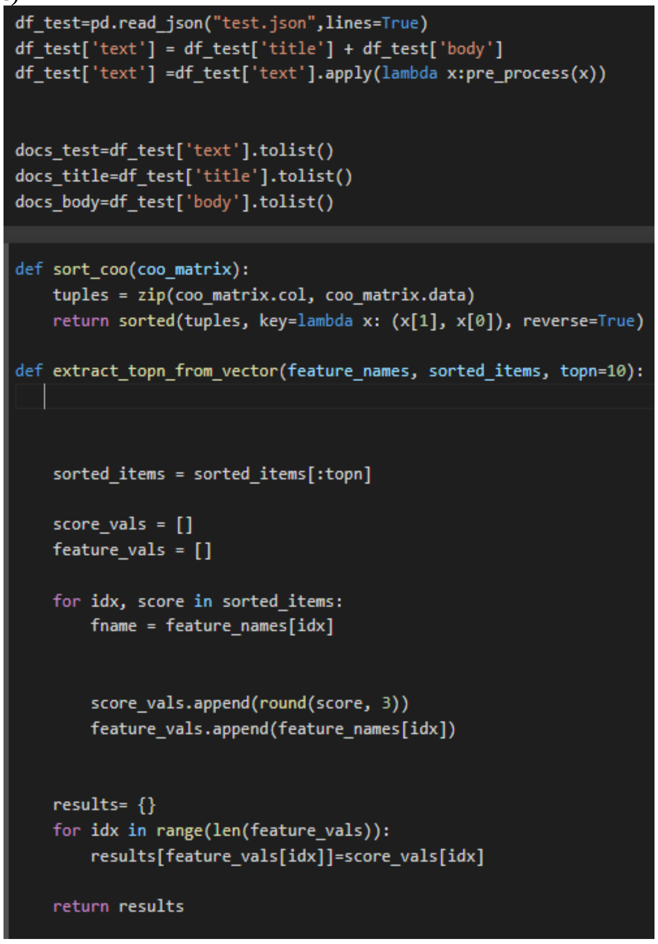
```
TfidfTransformer()
```

```
tfidf_transformer.idf_
```

```
array([ 7.37717703,  9.80492526,  9.51724319, ...,  8.82409601,
       10.21039037,  9.51724319])
```

Fig. 2: b) Applying TfIdf Transformer

c)

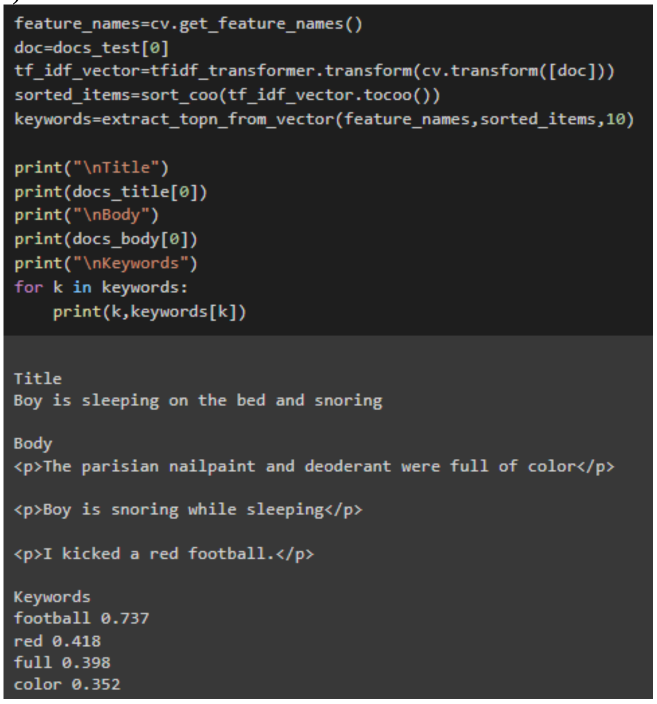


```
df_test=pd.read_json("test.json",lines=True)
df_test['text'] = df_test['title'] + df_test['body']
df_test['text'] =df_test['text'].apply(lambda x:pre_process(x))


docs_test=df_test['text'].tolist()
docs_title=df_test['title'].tolist()
docs_body=df_test['body'].tolist()
```

```
def sort_coo(coo_matrix):
    tuples = zip(coo_matrix.col, coo_matrix.data)
    return sorted(tuples, key=lambda x: (x[1], x[0]), reverse=True)

def extract_topn_from_vector(feature_names, sorted_items, topn=10):


    sorted_items = sorted_items[:topn]

    score_vals = []
    feature_vals = []

    for idx, score in sorted_items:
        fname = feature_names[idx]


        score_vals.append(round(score, 3))
        feature_vals.append(feature_names[idx])


    results= {}
    for idx in range(len(feature_vals)):
        results[feature_vals[idx]]=score_vals[idx]

    return results
```

Fig. 3: c) Computing TF-IDF score

d)



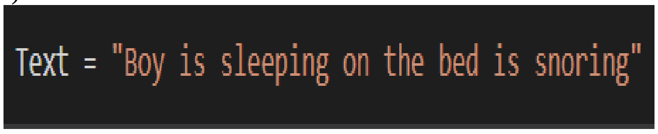
```
feature_names=cv.get_feature_names()
doc=docs_test[0]
tf_idf_vector=tfidf_transformer.transform(cv.transform([doc]))
sorted_items=sort_coo(tf_idf_vector.tocoo())
keywords=extract_topn_from_vector(feature_names,sorted_items,10)

print("\nTitle")
print(docs_title[0])
print("\nBody")
print(docs_body[0])
print("\nKeywords")
for k in keywords:
    print(k,keywords[k])
```

```
Title
Boy is sleeping on the bed and snoring

Body
<p>The parisian nailpaint and deoderant were full of color</p>

<p>Boy is snoring while sleeping</p>

<p>I kicked a red football.</p>

Keywords
football 0.737
red 0.418
full 0.398
color 0.352
```

Fig. 4: d) Extracting Keywords from the text along with the scores which indicate importance

B. *RAKE*

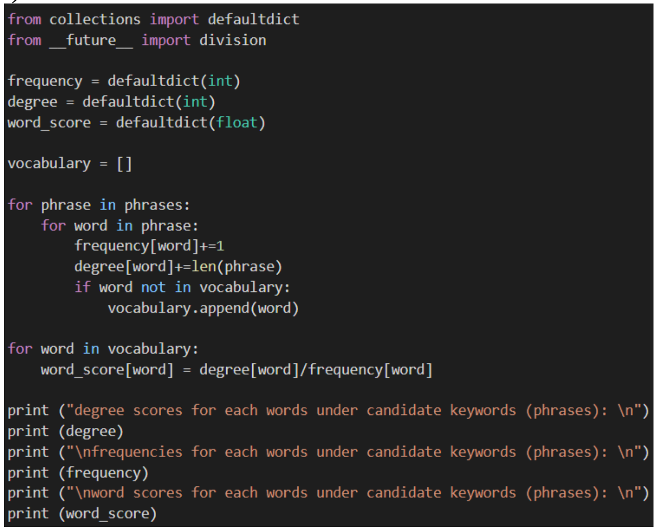
a)



```
Text = "Boy is sleeping on the bed is snoring"
```

Fig. 5: a) Input text

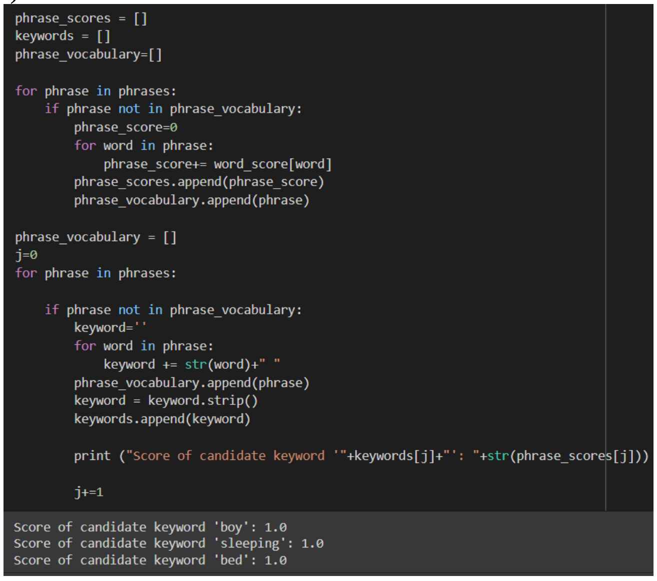
b)



```
from collections import defaultdict
from __future__ import division

frequency = defaultdict(int)
degree = defaultdict(int)
word_score = defaultdict(float)

vocabulary = []

for phrase in phrases:
    for word in phrase:
        frequency[word]+=1
        degree[word]+=len(phrase)
        if word not in vocabulary:
            vocabulary.append(word)

for word in vocabulary:
    word_score[word] = degree[word]/frequency[word]

print ("degree scores for each words under candidate keywords (phrases): \n")
print (degree)
print ("\nfrequencies for each words under candidate keywords (phrases): \n")
print (frequency)
print ("\nword scores for each words under candidate keywords (phrases): \n")
print (word_score)
```

Fig. 6: b) Frequency and Degree are Calculated and then word_scores are calculated

c)



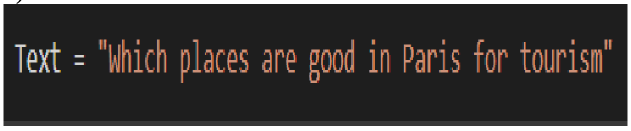
```
phrase_scores = []
keywords = []
phrase_vocabulary=[]

for phrase in phrases:
    if phrase not in phrase_vocabulary:
        phrase_score=0
        for word in phrase:
            phrase_score+= word_score[word]
        phrase_scores.append(phrase_score)
        phrase_vocabulary.append(phrase)

phrase_vocabulary = []
j=0
for phrase in phrases:

    if phrase not in phrase_vocabulary:
        keyword=''
        for word in phrase:
            keyword += str(word)+" "
        phrase_vocabulary.append(phrase)
        keyword = keyword.strip()
        keywords.append(keyword)

        print ("Score of candidate keyword '"+keywords[j]+"': "+str(phrase_scores[j]))

        j+=1
```

```
Score of candidate keyword 'boy': 1.0
Score of candidate keyword 'sleeping': 1.0
Score of candidate keyword 'bed': 1.0
```

Fig. 7: c) Keywords are extracted along with the scores which indicate importance

C. *TextRank*

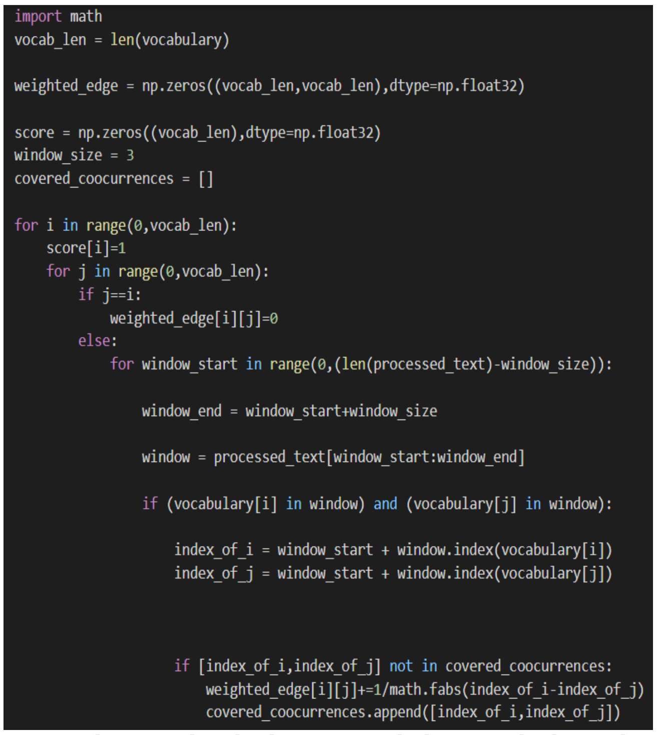
a)



```
Text = "Which places are good in Paris for tourism"
```

Fig. 8: .a) Input text

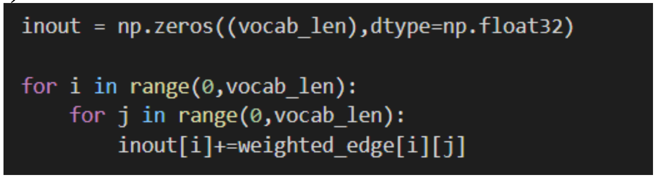
b)



```
import math
vocab_len = len(vocabulary)

weighted_edge = np.zeros((vocab_len,vocab_len),dtype=np.float32)

score = np.zeros((vocab_len),dtype=np.float32)
window_size = 3
covered_coocurrences = []

for i in range(0,vocab_len):
    score[i]=1
    for j in range(0,vocab_len):
        if j==i:
            weighted_edge[i][j]=0
        else:
            for window_start in range(0,(len(processed_text)-window_size)):

                window_end = window_start+window_size

                window = processed_text[window_start:window_end]

                if (vocabulary[i] in window) and (vocabulary[j] in window):

                    index_of_i = window_start + window.index(vocabulary[i])
                    index_of_j = window_start + window.index(vocabulary[j])



                    if [index_of_i,index_of_j] not in covered_coocurrences:
                        weighted_edge[i][j]+=1/math.fabs(index_of_i-index_of_j)
                        covered_coocurrences.append([index_of_i,index_of_j])
```

Fig. 9: b) a graph is built via a vocabulary in which words serve as the vertex of the graph

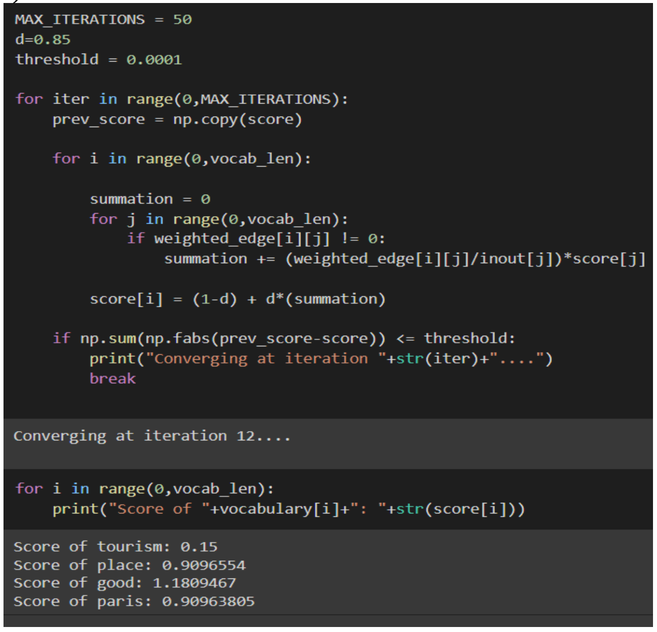
c)



```
inout = np.zeros((vocab_len),dtype=np.float32)

for i in range(0,vocab_len):
    for j in range(0,vocab_len):
        inout[i]+=weighted_edge[i][j]
```

Fig. 10: c) Calculating weighted summation of connections of a vertex.

d)



```
MAX_ITERATIONS = 50
d=0.85
threshold = 0.0001

for iter in range(0,MAX_ITERATIONS):
    prev_score = np.copy(score)

    for i in range(0,vocab_len):

        summation = 0
        for j in range(0,vocab_len):
            if weighted_edge[i][j] != 0:
                summation += (weighted_edge[i][j]/inout[j])*score[j]

        score[i] = (1-d) + d*(summation)

    if np.sum(np.fabs(prev_score-score)) <= threshold:
        print("Converging at iteration "+str(iter)+"....")
        break
```

```
Converging at iteration 12....
```

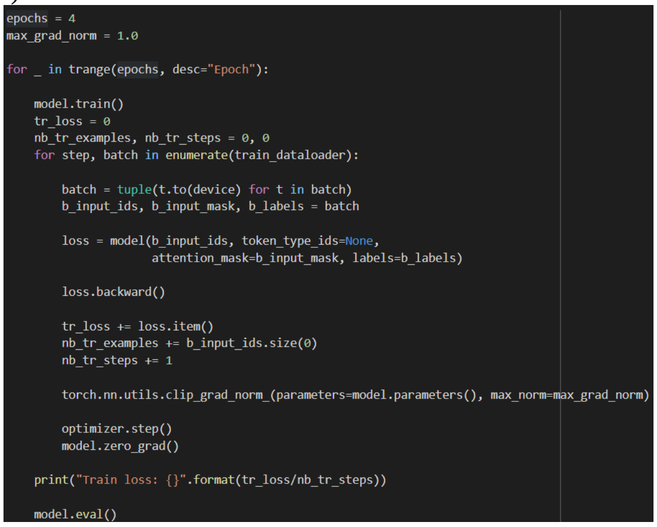
```
for i in range(0,vocab_len):
    print("Score of "+vocabulary[i]+": "+str(score[i]))
```

```
Score of tourism: 0.15
Score of place: 0.9096554
Score of good: 1.1809467
Score of paris: 0.90963805
```

Fig. 11: d) Scoring Vertices and then extracting keywords as per their importance based on scores.

D. Bidirectional Encoder Decoder

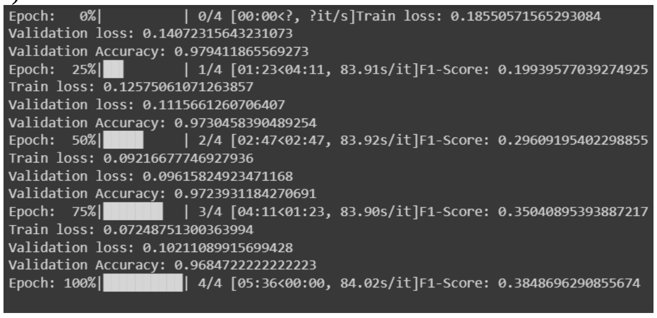
a)



```
epochs = 4
max_grad_norm = 1.0

for _ in trange(epochs, desc="Epoch"):

    model.train()
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    for step, batch in enumerate(train_dataloader):

        batch = tuple(t.to(device) for t in batch)
        b_input_ids, b_input_mask, b_labels = batch

        loss = model(b_input_ids, token_type_ids=None,
                     attention_mask=b_input_mask, labels=b_labels)

        loss.backward()

        tr_loss += loss.item()
        nb_tr_examples += b_input_ids.size(0)
        nb_tr_steps += 1

        torch.nn.utils.clip_grad_norm_(parameters=model.parameters(), max_norm=max_grad_norm)

        optimizer.step()
        model.zero_grad()

    print("Train loss: {}".format(tr_loss/nb_tr_steps))

    model.eval()
```

Fig. 12: a) Building the model

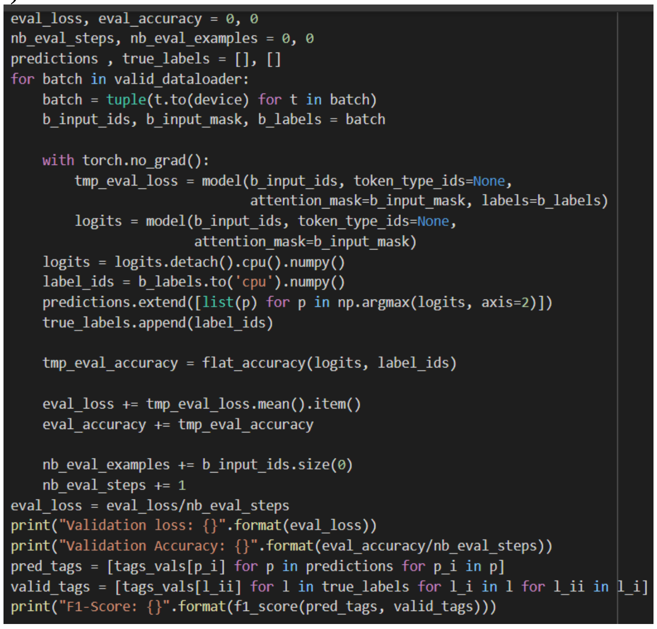
b)



```
Epoch:   0%|          | 0/4 [00:00<?, ?it/s]Train loss: 0.18550571565293084
Validation loss: 0.14072315643231073
Validation Accuracy: 0.979411865569273
Epoch:  25%|██        | 1/4 [01:23<04:11, 83.91s/it]F1-Score: 0.19939577039274925
Train loss: 0.12575061071263857
Validation loss: 0.1115661260706407
Validation Accuracy: 0.9730458390489254
Epoch:  50%|████      | 2/4 [02:47<02:47, 83.92s/it]F1-Score: 0.29609195402298855
Train loss: 0.09216677746927936
Validation loss: 0.09615824923471168
Validation Accuracy: 0.9723931184270691
Epoch:  75%|██████    | 3/4 [04:11<01:23, 83.90s/it]F1-Score: 0.35040895393887217
Train loss: 0.07248751300363994
Validation loss: 0.10211089915699428
Validation Accuracy: 0.9684722222222223
Epoch: 100%|████████| 4/4 [05:36<00:00, 84.02s/it]F1-Score: 0.3848696290855674
```

Fig. 13: b) Running the model

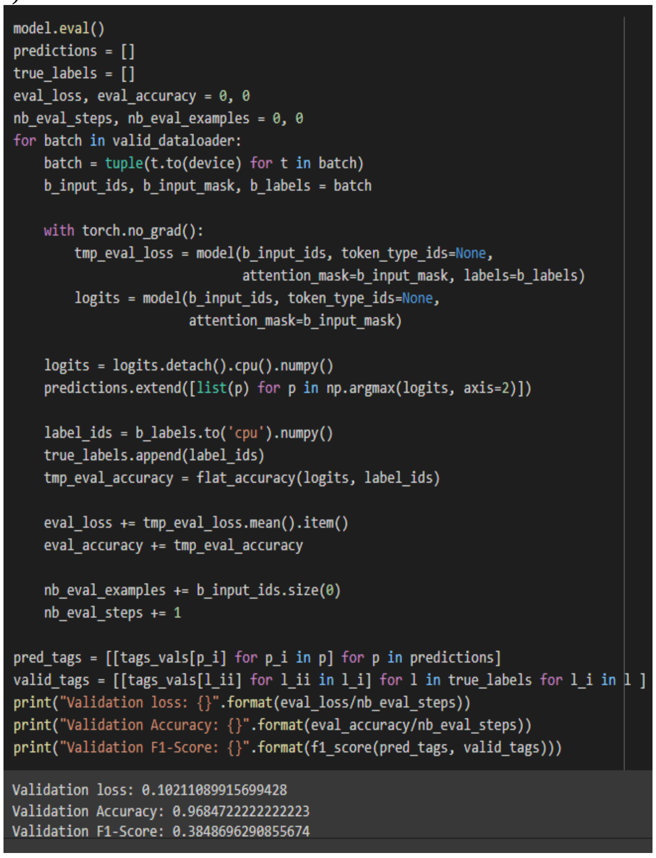
c)



```
eval_loss, eval_accuracy = 0, 0
nb_eval_steps, nb_eval_examples = 0, 0
predictions , true_labels = [], []
for batch in valid_dataloader:
    batch = tuple(t.to(device) for t in batch)
    b_input_ids, b_input_mask, b_labels = batch

    with torch.no_grad():
        tmp_eval_loss = model(b_input_ids, token_type_ids=None,
                              attention_mask=b_input_mask, labels=b_labels)
        logits = model(b_input_ids, token_type_ids=None,
                       attention_mask=b_input_mask)
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()
    predictions.extend([list(p) for p in np.argmax(logits, axis=2)])
    true_labels.append(label_ids)

    tmp_eval_accuracy = flat_accuracy(logits, label_ids)

    eval_loss += tmp_eval_loss.mean().item()
    eval_accuracy += tmp_eval_accuracy

    nb_eval_examples += b_input_ids.size(0)
    nb_eval_steps += 1
eval_loss = eval_loss/nb_eval_steps
print("Validation loss: {}".format(eval_loss))
print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_steps))
pred_tags = [tags_vals[p_i] for p in predictions for p_i in p]
valid_tags = [tags_vals[l_ii] for l in true_labels for l_i in l for l_ii in l_i]
print("F1-Score: {}".format(f1_score(pred_tags, valid_tags)))
```

Fig. 14

d)

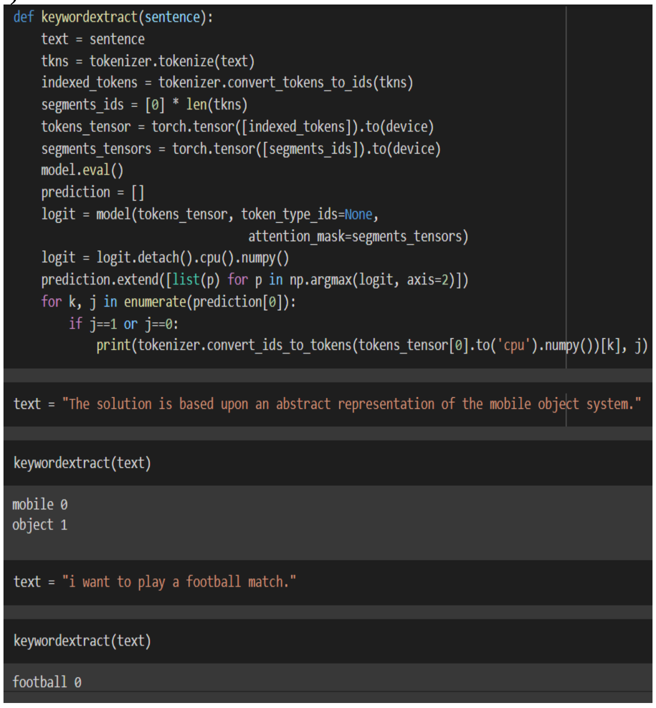


```
model.eval()
predictions = []
true_labels = []
eval_loss, eval_accuracy = 0, 0
nb_eval_steps, nb_eval_examples = 0, 0
for batch in valid_dataloader:
    batch = tuple(t.to(device) for t in batch)
    b_input_ids, b_input_mask, b_labels = batch

    with torch.no_grad():
        tmp_eval_loss = model(b_input_ids, token_type_ids=None,
                              attention_mask=b_input_mask, labels=b_labels)
        logits = model(b_input_ids, token_type_ids=None,
                       attention_mask=b_input_mask)

    logits = logits.detach().cpu().numpy()
    predictions.extend([list(p) for p in np.argmax(logits, axis=2)])

    label_ids = b_labels.to('cpu').numpy()
    true_labels.append(label_ids)
    tmp_eval_accuracy = flat_accuracy(logits, label_ids)

    eval_loss += tmp_eval_loss.mean().item()
    eval_accuracy += tmp_eval_accuracy

    nb_eval_examples += b_input_ids.size(0)
    nb_eval_steps += 1

pred_tags = [[tags_vals[p_i] for p_i in p] for p in predictions]
valid_tags = [[tags_vals[l_ii] for l_ii in l_i] for l in true_labels for l_i in l ]
print("Validation loss: {}".format(eval_loss/nb_eval_steps))
print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_steps))
print("Validation F1-Score: {}".format(f1_score(pred_tags, valid_tags)))
```

```
Validation loss: 0.10211089915699428
Validation Accuracy: 0.9684722222222223
Validation F1-Score: 0.3848696290855674
```

Fig. 15: .c)&d) Evaluating the model

e)



```
def keywordextract(sentence):
    text = sentence
    tkns = tokenizer.tokenize(text)
    indexed_tokens = tokenizer.convert_tokens_to_ids(tkns)
    segments_ids = [0] * len(tkns)
    tokens_tensor = torch.tensor([indexed_tokens]).to(device)
    segments_tensors = torch.tensor([segments_ids]).to(device)
    model.eval()
    prediction = []
    logit = model(tokens_tensor, token_type_ids=None,
                  attention_mask=segments_tensors)
    logit = logit.detach().cpu().numpy()
    prediction.extend([list(p) for p in np.argmax(logit, axis=2)])
    for k, j in enumerate(prediction[0]):
        if j==1 or j==0:
            print(tokenizer.convert_ids_to_tokens(tokens_tensor[0].to('cpu').numpy())[k], j)
```

```
text = "The solution is based upon an abstract representation of the mobile object system."
```

```
keywordextract(text)
```

```
mobile 0
object 1
```

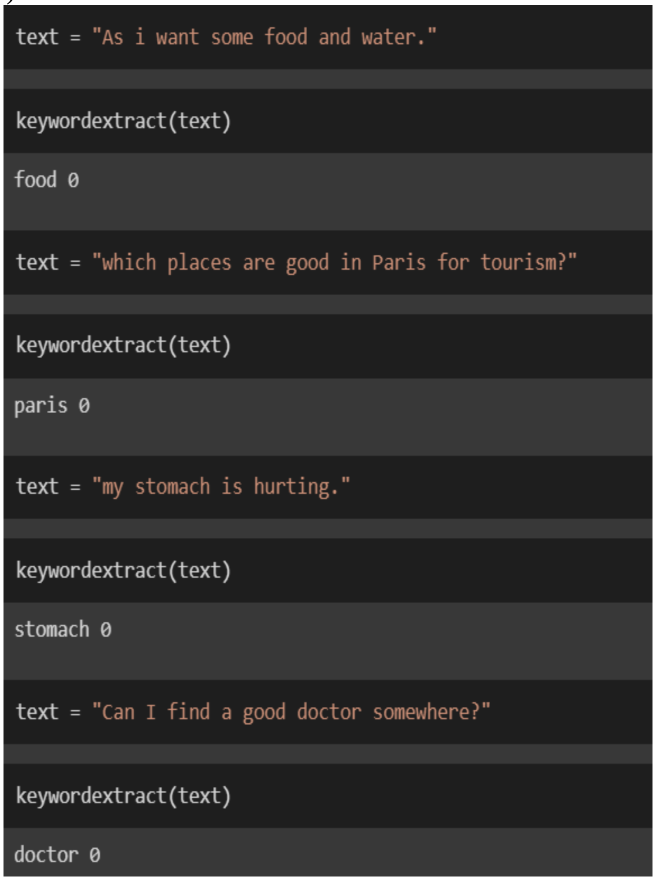
```
text = "i want to play a football match."
```

```
keywordextract(text)
```

```
football 0
```

Fig. 16

f)



```
text = "As i want some food and water."

keywordextract(text)
food 0

text = "which places are good in Paris for tourism?"

keywordextract(text)
paris 0

text = "my stomach is hurting."

keywordextract(text)
stomach 0

text = "Can I find a good doctor somewhere?"

keywordextract(text)
doctor 0
```

Fig. 17: e) Extracting Keywords f) Results

V. COMPARISON

As we note from the above preliminary investigation the neural network based encoder-decoder model outperforms in terms of keyword extraction. The only caveat is that the model does a lot of processing and hence needs a GPU to run it effectively but the result more than covers up for the processing requirements.

VI. RESULTS

From the comparison we can successfully conclude that our sequence to sequence based encoder-decoder model works better than traditional ML,NLP based methods when it comes to keyword extraction from short text.

In the future we can also adjust our algorithm to compare it with YAKE which works for keyword extraction across all languages and see how our neural network based model fares, and also see whether we can reduce the processing requirements of our model.

ACKNOWLEDGEMENT

I would like to thank Dr. Thenmozhi T., Associate Professor Grade 1, School of Computer Science and Engineering (SCOPE), Vellore Institute of Technology, for her unwavering support and guidance throughout the tenure of this research study that led to its successful completion.

REFERENCES

[1.] Zhang, Mingxi, Hangfei Hu, Guanying Su, Yuening Zhang, and Xiaohong Wang. "An extension of TF-IDF model for extracting feature terms." International Journal of Scientific Research and Innovative Technology 5, no. 5 (2018): 64-76.

[2.] Pan, Suhan, Zhiqiang Li, and Juan Dai. "An improved TextRank keywords extraction algorithm." In Proceedings of the ACM Turing Celebration Conference-China,(2019) pp. 1-7.

[3.] Rose, Stuart, Dave Engel, Nick Cramer, and Wendy Cowley. "Automatic keyword extraction from individual documents." Text mining: applications and theory 1 (2010): 1-20.

[4.] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems, (2014) 27.

[5.] 4. Tenney, Ian, Dipanjan Das, and Ellie Pavlick. "BERT rediscovers the classical NLP pipeline." arXiv preprint arXiv:1905. (2019) 05950.

[6.] Zhang, Yu, Mingxiang Tuo, Qingyu Yin, Le Qi, Xuxiang Wang, and Ting Liu. "Keywords extraction with deep neural network model." *Neurocomputing* 383 (2020): 113-121.