Github Issue Summarization

Gianluca Rea
m. 278722

January 23, 2023

Contents

1 Introduction

In natural language processing, there are two categories of summarization:

- Extractive Summarization
- Abstractive Summarization

Abstractive Summarization includes heuristic approaches to train the system in making an attempt to understand the whole context and generate a summary based on that understanding. This is a more human-like way of generating summaries, which are more effective than the extractive approaches.

Extractive Summarization essentially involves extracting particular pieces of text (usual sentences) based on predefined weights assigned to the important words where the selection of the text depends on the weights of the words in it. Usually, the default weights are assigned according to the frequency of occurrence of a word. Here, the length of the summary can be manipulated by defining the maximum and minimum number of sentences to be included in the summary.

In our case, we are going to implement an abstractive summarization. The problem concerns the auto-generation of GitHub titles from the summarization of the GitHub issue body.

2 The Dataset

For this task, I used the open-source project gharchive.org. The Dataset is called github-issues.

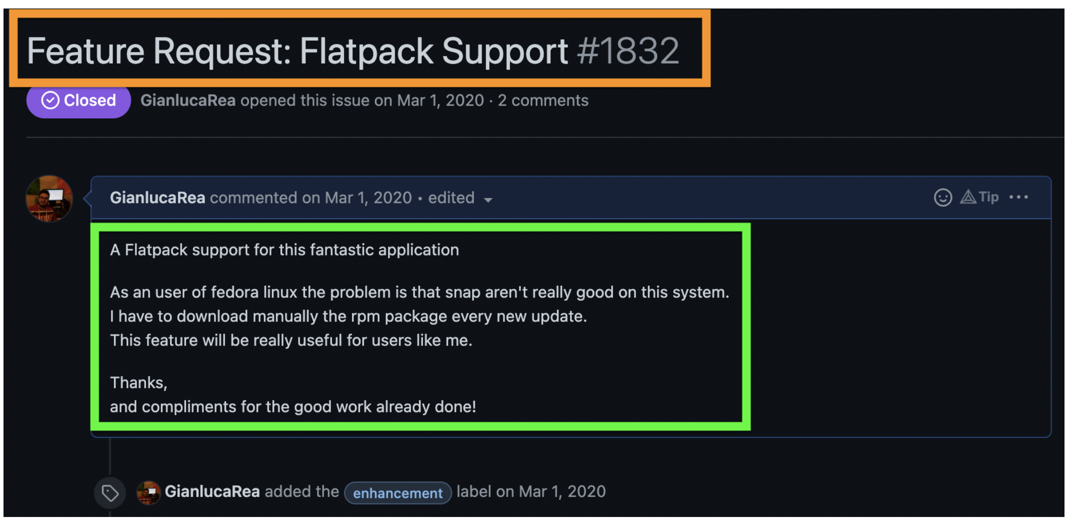

Figure 1: github issue

As we can see in the image above there are two main components of a GitHub issue. The first one is the title highlighted in orange meanwhile in green, we can see the body of the issue.

From figure 2 we can see what part of the dataset we are going to use. After the "issue URL" column drops we are left with two columns "issue title" and "body". Our task will be to summarize the body to have a new text title similar to the issue title. From the entire dataset, we also took only 50000 entries. This choice was forced by the low computational capacity of the computer.

	issue_title	body
0	can't load the addon. issue to: https://github.com/zhangyuanwei/node-images/issues error: /lib64/libc.so.6: version glibc_2.14' not found required by /usr/local/app/taf/fileserver.fileserver/bin/s...	can't load the addon. issue to: https://github.com/zhangyuanwei/node-images/issues error: /lib64/libc.so.6: version glibc_2.14' not found required by /usr/local/app/taf/fileserver.fileserver/bin/s...
1	hcl accessibility a11yblocking a11ymas mas4.2.10 hcl-makecode win10-edge -title screen reader-help-javascript-call a function narrator focus does not moving to expand side a documentation button a...	user experience: user who depends on screen reader will get confused if narrator focus does not retain on expand side a documentation button after pressing enter on collapse side a documentation b...
2	issue 1265: issue 1264: issue 1261: issue 1260: issue 1257: issue 1256: issue 1253: issue 1252: issue 1250: issue 1247: issue 1246: issue 1243: issue 1242: issue 1239: issue 1237: issue 1236: issu...	attachments: https:& x2f;& x2f;github.com& x2f;matisiekpl& x2f;czekolada& x2f;issues& x2f;1265

```
df.shape
```

```
(50000, 2)
```

Figure 2: gihub issue dataset head

3 Preprocessing

For the preprocessing part, a few changes were made to the dataset for cleaning purposes. The first thing done was to eliminate the contraction by substituting them with the formal form of the phrase. We also eliminated special characters and word issues from the various text with the text cleaner function.

```
1  stop_words = set(nltk.corpus.stopwords.words('english'))
2  
3  def text_cleaner(text,num):
4      newString = text.lower()
5      newString = BeautifulSoup(newString, "lxml").text
6      newString = re.sub(r'\([^)]*\)', '', newString)
7      newString = re.sub('"','', newString)
8      newString = ' '.join([contraction_mapping[t] if t in contraction_mapping else t
       for t in newString.split(" ")])
9      newString = re.sub(r"'s\b","",newString)
10     newString = re.sub("[^a-zA-Z]", " ", newString)
11     newString = re.sub('[m]{2,}', 'mm', newString)
12     newString = re.sub('issue',"",newString)
13     newString = re.sub('issu',"",newString)
14 
15     if(num==0):
16         tokens = [w for w in newString.split() if not w in stop_words]
17     else:
18         tokens=newString.split()
19     long_words=[]
20     for i in tokens:
21         if len(i)>1:
22             long_words.append(i)
23     return (" ".join(long_words)).strip()
24     return M
```

Besides this first cleaning, we also removed stop words. A stop word is a commonly used word (such as "the", "a", "an", or "in"). We would not want these words to take up space in our database, or take up the valuable processing time. For this, we can remove them easily, by storing a list of words that you consider stop words. NLTK (Natual Language Toolkit) in python has a list of stopwords stored in 16 different languages. Obviously, we have chosen the English list. The usage and removal of stop words are shown in the previous code on lines 1 and 16.

After this cleaning, we also remove all the rows that were not written in English. Before this, we had to remove empty rows to prevent checking on an empty row for the detection of the language.

```
data = pd.DataFrame()
data['cleaned_title']=cleaned_title
data['cleaned_body']=cleaned_body

## Drop empty rows
data.replace('', np.nan, inplace=True)
data.dropna(axis=0,inplace=True)

### Add language column
data['detect'] = data['cleaned_body'].apply(detect)

### Remove column where language is not english
data = data[data['detect'] == 'en']
data.drop(['detect'], axis=1, inplace=True)
data.shape
```

After cleaning the non-English phrase we end it up with a dataset of 42701 valid rows. But these rows could be formed by really long text once in a while so using the following histogram we calculated the percentage of title and body where the number of words was less than a threshold.

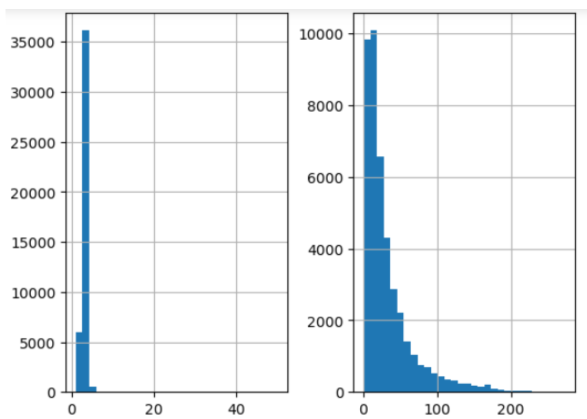

Figure 3: length of text Histogram

We found out that in our case a good compromise was:

- 8 Words for title taking 99.91 percent of the title group
- 120 Words for body taking 96.27 percent of the body group

We also calculated the inferior limit and we found out that 98.85 of title rows are formed by more than two words. For the body, the percentage of rows that are formed by more than 5 words is 97.16

After this, we removed the rows that were not inside the predefined range of words of the upper limit and we applied special tokens to the start and the end of the title for future usage to check if it was not empty. We used only the upper limit because the removal of the inferior limit will not have consequences on the results and the performance.

```
# Adding special tokens sostok and eostok as START and END tokens of title
df['title'] = df['title'].apply(lambda x : 'sostok '+ x + ' eostok')
```

Finally, we split the title and body into train and test collections with a test size of 0.1 of the entire collection.

```
x_tr,x_val,y_tr,y_val=train_test_split(np.array(df['body']),np.array(df['title']),
    test_size=0.1,random_state=0,shuffle=True)
```

4 Tokenization

We then proceeded to tokenize the text for both title and the body. Tokenization is one of the most common tasks when working with text data and consists of splitting a phrase, sentence, or paragraph into smaller units, such as individual words or terms. Each of these smaller units is called a token. Before processing a natural language, we need to identify the words that constitute a string of characters. That's why tokenization is the most basic step to proceed with NLP(Natual Language Processing). This is important because the meaning of the text could easily be interpreted by analyzing the words present in the text.

4.1 Body Tokenization

The code below was used to tokenize the body

```
x_tokenizer = Tokenizer()
x_tokenizer.fit_on_texts(list(x_tr))

thresh=3
cnt=0 ### Number of rare words -> words appearing less than the thresh
tot_cnt=0 ### Vocabulary size
freq=0
tot_freq=0

for key,value in x_tokenizer.word_counts.items():
    tot_cnt=tot_cnt+1
    tot_freq=tot_freq+value
    if(value<thresh):
        cnt=cnt+1
        freq=freq+value

#prepare a tokenizer for reviews on training data
x_tokenizer = Tokenizer(num_words=tot_cnt-cnt)
x_tokenizer.fit_on_texts(list(x_tr))

#convert text sequences into integer sequences
x_tr_seq    =   x_tokenizer.texts_to_sequences(x_tr)
x_val_seq   =   x_tokenizer.texts_to_sequences(x_val)

#padding zero upto maximum length
x_tr    =   pad_sequences(x_tr_seq,  maxlen=max_body_len, padding='post')
x_val   =   pad_sequences(x_val_seq, maxlen=max_body_len, padding='post')

#size of vocabulary ( +1 for padding token)
x_voc   =  x_tokenizer.num_words + 1
```

As a result of the code, we found out that the percentage of rare words, defined as the word which has appeared less than the thresh in this case 3, was 68.43. The total coverage of rare words was 5.31. In the end, we have collected a vocabulary with a size equal to 20923

4.2 Title Tokenization

The code belowe was used to tokenize the title

```
#prepare a tokenizer for reviews on training data
y_tokenizer = Tokenizer()
y_tokenizer.fit_on_texts(list(y_tr))

thresh=6
cnt=0
tot_cnt=0
freq=0
tot_freq=0

for key,value in y_tokenizer.word_counts.items():
    tot_cnt=tot_cnt+1
    tot_freq=tot_freq+value
    if(value<thresh):
        cnt=cnt+1
        freq=freq+value

#prepare a tokenizer for reviews on training data
y_tokenizer = Tokenizer(num_words=tot_cnt-cnt)
y_tokenizer.fit_on_texts(list(y_tr))

#convert text sequences into integer sequences
y_tr_seq    =   y_tokenizer.texts_to_sequences(y_tr)
y_val_seq   =   y_tokenizer.texts_to_sequences(y_val)

#padding zero upto maximum length
y_tr    =   pad_sequences(y_tr_seq, maxlen=max_title_len, padding='post')
y_val   =   pad_sequences(y_val_seq, maxlen=max_title_len, padding='post')

#size of vocabulary
y_voc  =   y_tokenizer.num_words +1

y_tokenizer.word_counts['sostok'],len(y_tr)
```

As a result of the code, we found out that the percentage of rare words, defined as the word which has appeared less than the thresh in this case 6, was 83.19. The total coverage of rare words was 9.55. In the end, we collected a vocabulary with a size equal to 2348.

After this, we deleted all the rows in which the body or title was formed only by the start and end tokens.

5 Model

We are finally at the model-building part. A machine learning model is a file that has been trained to recognize certain types of patterns. We train a model over a set of data, providing it with an algorithm that can use to reason over and learn from those data. Once the model is trained, we can use it to reason over data that it hasn't seen before, and make predictions about those data.

Before showing the code of our model we need to familiarize ourselves with a few terms which are required prior to building the model.

- Return Sequences = True: When the return sequences parameter is set to True, LSTM produces the hidden state and cell state for every timestep
- Return State = True: When return state = True, LSTM produces the hidden state and cell state of the last timestep only

- Initial State: This is used to initialize the internal states of the LSTM for the first timestep
- Stacked LSTM: Stacked LSTM has multiple layers of LSTM stacked on top of each other. This leads to a better representation of the sequence.

Here, we are building a 3-stacked LSTM for the encoder:

```
K.clear_session()

latent_dim = 300
embedding_dim=100

# Encoder
encoder_inputs = Input(shape=(max_body_len,))

#embedding layer
enc_emb =  Embedding(x_voc, embedding_dim,trainable=True)(encoder_inputs)

#encoder lstm 1
encoder_lstm1 = LSTM(latent_dim,return_sequences=True,return_state=True,dropout=0.4,
recurrent_dropout=0.4)
encoder_output1, state_h1, state_c1 = encoder_lstm1(enc_emb)

#encoder lstm 2
encoder_lstm2 = LSTM(latent_dim,return_sequences=True,return_state=True,
dropout=0.4,recurrent_dropout=0.4)
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)

#encoder lstm 3
encoder_lstm3=LSTM(latent_dim, return_state=True,
 return_sequences=True,dropout=0.4,recurrent_dropout=0.4)
encoder_outputs, state_h, state_c= encoder_lstm3(encoder_output2)

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None,))

#embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim,trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

decoder_lstm = LSTM(latent_dim, return_sequences=True,
return_state=True,dropout=0.4,recurrent_dropout=0.2)
decoder_outputs,decoder_fwd_state, decoder_back_state =
decoder_lstm(dec_emb,initial_state=[state_h, state_c])

# Attention layer
attn_layer = AttentionLayer(name='attention_layer')
attn_out, attn_states = attn_layer([encoder_outputs, decoder_outputs])

# Concat attention input and decoder LSTM output
decoder_concat_input = Concatenate(axis=-1, name='concat_layer')
([decoder_outputs, attn_out])

#dense layer
decoder_dense =  TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_concat_input)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```

After this, we set up the batch size to 128 and we set the number of epoch to 8.

```
history=model.fit([x_tr,y_tr[:,:-1]],
y_tr.reshape(y_tr.shape[0],y_tr.shape[1], 1)[:,1:] ,
epochs=8,callbacks=[es],batch_size=128,
validation_data=([x_val,y_val[:,:-1]],
y_val.reshape(y_val.shape[0],y_val.shape[1], 1)[:,1:]))
```

with the following results:

Epoch	Loss	Val Loss
1	2.3261	2.0039
2	1.9497	1.9345
3	1.8714	1.8975
4	1.8162	1.9302
5	1.7769	1.8365
6	1.7367	1.8292
7	1.6969	1.7967
8	1.6662	1.7855

Here we can see our model summary

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 120)]        0           []

 embedding (Embedding)          (None, 120, 100)     2092300     ['input_1[0][0]']

 lstm (LSTM)                    [(None, 120, 300),   481200      ['embedding[0][0]']
                                 (None, 300),
                                 (None, 300)]

 input_2 (InputLayer)           [(None, None)]       0           []

 lstm_1 (LSTM)                  [(None, 120, 300),   721200      ['lstm[0][0]']
                                 (None, 300),
                                 (None, 300)]

 embedding_1 (Embedding)        (None, None, 100)    234800      ['input_2[0][0]']

 lstm_2 (LSTM)                  [(None, 120, 300),   721200      ['lstm_1[0][0]']
                                 (None, 300),
                                 (None, 300)]

 lstm_3 (LSTM)                  [(None, None, 300),  481200      ['embedding_1[0][0]',
                                 (None, 300),                     'lstm_2[0][1]',
                                 (None, 300)]                     'lstm_2[0][2]']

 attention_layer (AttentionLaye  ((None, None, 300),  180300     ['lstm_2[0][0]',
 r)                              (None, None, 120))               'lstm_3[0][0]']

 concat_layer (Concatenate)     (None, None, 600)    0           ['lstm_3[0][0]',
                                                                  'attention_layer[0][0]']

 time_distributed (TimeDistribu  (None, None, 2348)  1411148     ['concat_layer[0][0]']
 ted)

==================================================================================================
Total params: 6,323,348
Trainable params: 6,323,348
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 4: Model

6 Diagnostic plots

We analyzed the diagnostic plots to understand the behavior of the model over time and we can infer that validation loss has increased after epoch 4 for 4 successive epochs. Although, it could be useful to train the model for other epochs to be sure about the increase would be maintained over time.

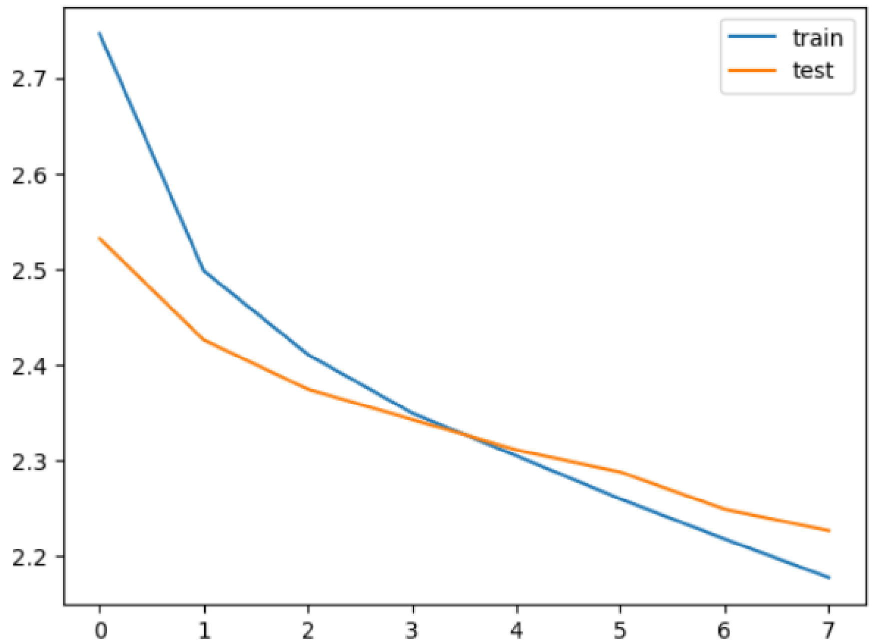

Figure 5: Diagnostic plots

7 Inference

We set up the inference of the encoder and decoder:

```
# Encode the input sequence to get the feature vector
encoder_model = Model(inputs=encoder_inputs,outputs=[encoder_outputs, state_h, state_c
    ])

# Decoder setup
# Below tensors will hold the states of the previous time step
decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_hidden_state_input = Input(shape=(max_body_len,latent_dim))

# Get the embeddings of the decoder sequence
dec_emb2= dec_emb_layer(decoder_inputs)
# To predict the next word in the sequence, set the initial states to the states from
    the previous time step
decoder_outputs2, state_h2, state_c2 = decoder_lstm(dec_emb2, initial_state=[
    decoder_state_input_h, decoder_state_input_c])

#attention inference
attn_out_inf, attn_states_inf = attn_layer([decoder_hidden_state_input,
    decoder_outputs2])
decoder_inf_concat = Concatenate(axis=-1, name='concat')([decoder_outputs2,
    attn_out_inf])
```

```

# A dense softmax layer to generate prob dist. over the target vocabulary
decoder_outputs2 = decoder_dense(decoder_inf_concat)

# Final decoder model
decoder_model = Model(
    [decoder_inputs] + [decoder_hidden_state_input,decoder_state_input_h,
    decoder_state_input_c],
    [decoder_outputs2] + [state_h2, state_c2])
```

We are defining a function below which is the implementation of the inference process:

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    e_out, e_h, e_c = encoder_model.predict(input_seq)
    # Generate an empty target sequence of length 1.
    target_seq = np.zeros((1,1))
    # Populate the first word of the target sequence with the start word.
    target_seq[0, 0] = target_word_index['sostok']
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + [e_out, e_h, e_c])
        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_token = reverse_target_word_index[sampled_token_index]
        if(sampled_token!='eostok'):
            decoded_sentence += ' '+sampled_token
        # Exit condition: either hit max length or find stop word.
        if (sampled_token == 'eostok'  or len(decoded_sentence.split()) >= (
    max_summary_len-1)):
            stop_condition = True
        # Update the target sequence (of length 1).
        target_seq = np.zeros((1,1))
        target_seq[0, 0] = sampled_token_index
        # Update internal states
        e_h, e_c = h, c
    return decoded_sentence
```

8 Evaluation

For our evaluation, we used ROGUE [4] which stands for Recall-Oriented Understudy for Gisting Evaluation that which is essentially a set of metrics for evaluating the automatic summarization of texts as well as machine translations. It works by comparing an automatically produced summary or translation against a set of reference summaries (typically human-produced).

If we consider just the number of overlapping words between the predicted summary and the reference summary we will get nothing as a metric. To get a good quantitative value, we can actually compute the precision and recall using the overlap.

Recall, in the context of Rogue, refers to how much of the reference summary the predicted summary is recovering or capturing.

A predicted summary can be also extremely long by capturing all words in the reference summary. But many of the words in the predicted summary may be useless, making the summary unnecessarily verbose. So in addition to recall we also use precision.

In terms of precision, what you are essentially measuring is, how much of the predicted summary was fact relevant or needed.

The precision aspect becomes really crucial when you are trying to generate summaries that are concise in nature. So it is always best to compute both the precision and recall and then report the F-score.

$$Recall = \frac{num\_overlapping\_words}{total\_words\_in\_reference\_summary} \tag{1}$$

$$Precision = \frac{num\_overlapping\_words}{total\_words\_in\_autogenerated\_summary} \tag{2}$$

$$F1 = 2 \times \frac{Recall \times Precision}{Recall - Precion} \tag{3}$$

ROUGE-N, ROUGE-S, and ROUGE-L can be thought of as the granularity of texts being compared between the system summaries and reference summaries.

- ROUGE-N — measures unigram, bigram, trigram, and higher order n-gram overlap
- ROUGE-L — measures the longest matching sequence of words using LCS. An advantage of using LCS is that it does not require consecutive matches but in-sequence matches that reflect sentence-level word order. Since it automatically includes the longest in-sequence common n-grams, you don't need a predefined n-gram length.
- ROUGE-S — Is any pair of words in a sentence in order, allowing for arbitrary gaps. This can also be called skip-gram concurrence. For example, skip-bigram measures the overlap of word pairs that can have a maximum of two gaps in between words. As an example, for the phrase "cat in the hat" the skip-bigrams would be "cat in, cat the cat hat, in the, in hat, the hat".

We evaluated the first 500 autogenerated titles in our case with the following mean results (value between 0 and 1):

	Rogue-1	Rogue-2	Rouge-l
Recall	0.101	0.02	0.101
Precision	0.150	0.039	0.150
F1-Score	0.115	0.030	0.115

There is numerous model available for the summarization of text. In this paper, [2] that we have studied the research is done on a similar topic using the BART[3], PRSummarizer[5], and iTAPE[1] models. We do highlight that the dataset is different and the models are better trained so the comparison is made to visually see the power of our model.

Approch	Rogue-1	Rogue-2	Rouge-l
BART	47.22	25.27	43.12
PRSummarizer	37.91	17.99	34.98
iTAPE	32.23	12.91	29.31

Our model result for again a different dataset

Approch	Rogue-1	Rogue-2	Rouge-l
GR	11.5	3.00	11.5

Analyzing the score we can see that our model works and is able to calculate unigrams with 11.5 % while the score is really low, 3 % for bigrams. In general, the Rouge-l F1 score is around % 11.5. We must say that this calculation where done on the first 500 predictions and not the entire dataset sample collected.

9 Conclusions

Below few good predictions are displayed to show the correct work of the tool.

```
Body: https github com prj rev bwfs dasmoto blob master dasmoto art resource css style css e
xcellent implementation dry principles setting font family elements one selector use set ent
ire page
Original title: great dry css
1/1 [==============================] - 2s 2s/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
Predicted title:  great dry css
```

Figure 6: Good Result 1

Following is an example of a bad prediction:

```
Body: make sure site deploys appropriate information db
Original title: migrate the db

1/1 [==============================] - 2s 2s/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
Predicted title:  add page
```

Figure 7: A bad result

We can say that deep learning could be improved by taking a bigger size of the dataset to work with. Also increasing the number of epochs will also result in better predictions.

References

[1] Songqiang Chen et al. "Stay professional and efficient: automatically generate titles for your bug reports". In: (Dec. 2020), pp. 385–397. DOI: 10.1145/3324884.3416538.

[2] Ivana Irsan et al. "AutoPRTitle: A Tool for Automatic Pull Request Title Generation". In: (June 2022).

[3] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: Jan. 2020, pp. 7871–7880. DOI: 10.18653/v1/2020.acl-main.703.

[4] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013.

[5] Zhongxin Liu et al. "Automatic Generation of Pull Request Descriptions". In: *CoRR* abs/1909.06987 (2019). arXiv: 1909.06987. URL: http://arxiv.org/abs/1909.06987.