Project Data Science - Text Summarization Technical Report

M. Lautaro Hickmann, Fabian Wurzberger, Megi Hoxhalli, Arne Lochner

April 28, 2021

Contents

1 Introduction

1.1 Motivation

This is the technical report to our research project, which focuses on analyzing a graph-transformer-based Multi-document Summarization (MDS) model with regard to the usage of different textual units as vertices in a graph structure guiding the summarization process. In addition our project provides insight in the possibility of utilizing attention weights to indicate source-origin information to improve explain-ability of abstractive MDS.

1.2 Structure

In Section 2, we introduce the different frameworks and tools that are relevant for the implementation of the project, as well as their extensions.

In Section 3, we describe our processing pipeline based on the existing GraphSum code provided by Li et al, which is used to generate the results reported in our scientific report.

1.3 Target audience

This documentation aims to provide further researchers information on how to extend our research to other attention based models and research questions.

1.4 Used hardware

- Ubuntu 18.04
- 3x P100 GPU (Tesla architecture), 16GB
- 16 cores (2x Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz)
- 128 GB RAM

1.5 Git repository

We provide a git repository containing all necessary code in order to reproduce our results. This git contains two branches, one for each research question (namely RQ1 and RQ2). The git repository contains the following structure:

- data: location for data sets and spm model
- env_local: bash scripts to set environment variables required by PaddlePaddle and ROUGE
- model_config: configuration files used for the GraphSum model
- pyrogue: python interface for the ROUGE perl script
- scripts: bash scripts to start process pipelines
- src: folder containing all required python modules and scripts

We further provide an `installer.sh` file to install all required dependencies to run the GraphSum model with anaconda. As a dependency we assume that anaconda is already installed.

2 Used Frameworks and Their Extension

2.1 PaddlePaddle

The research code by Li et al was implementend in PaddlePaddle. PaddlePaddle (PArallel Distributed Deep Learning) is a highly distributed deep learning framework developed by Baidu. PaddlePaddle's core is implemented in C++ with an API provided in python. The research code is implemented with PaddlePaddle 1.6.3, for our project we use the most recent compatible version of PaddlePaddle namely 1.8.1. We were not able to use the next major release of PaddlePaddle due to incompatabilities with the provided code. Further, the Nvidia Ampere architecture is only compatible with PaddlePaddle 2.0 due to missing backwards compatability of the CUDA drivers. Therefore we worked on an older GPU architecture (Tesla) and CUDA 10 [3].

2.2 GraphSum

We use the state-of-the-art abstractive graph-based transformer called GraphSum from Li et al. [1] as described in [**scientific report**]. We use the code provided by the authors, which is based on PaddlePaddle[1].

[1] `https://github.com/PaddlePaddle/Research/tree/master/NLP/ACL2020-GraphSum`

2.3 ROUGE 1.5.5

For Rouge calculation we use the perl script proposed by Lin [2] which was also used by Li et al. [1]. This perl script[2] creates many temporary folders and files for the calculation and can easily fragment the filesystem if used incorrectly. The script requires two files as input: a candidate summary file and a file containing the gold standard. The perl script reports an average ROUGE score over all examples provided in the files. For our research we installed perl with anaconda which required updating perl paths in certain scripts.

2.4 Jupyter Notebooks for Evaluation

For visualization purposes we provide jupyter notebooks in the `src/visualization` folder in the RQ2 branch of our repository.

3 Scripts for Executing the Evaluation Apparatus

3.1 Sentence- vs. paragraph-level representation

For our first research question we require a sentence-level representation of the MultiNews dataset. Therefore, we extend the provided pre-processing script of Li et al in order to transform the paragraph-level representation into sentence-level representation. For this purpose, additional command line parameters can be parsed to the `src/data_preprocess/graphsum/preprocess_graphsum_data.py` script. Namely the following new parameters are possible:

- num_examples (int): number of examples to pre-process, optional
- sentence_level (bool): flag indicating if sentence-level representation is desired

While researching we tested if pruning the sentence-level data by removing sentences which are too similar or un-similar to all other input sentences of a single example (multi document input). This pruning did not improve the performance of the model and therefore is not included in the scientific report. Nevertheless, the following parameters can be used in the pre-processing script:

- do_pruning (bool): do pruning
- pruning_low_threshold (float): minimum threshold for the average value of all incoming edges
- pruning_high_threshold (float): maximum threshold for the average value of all incoming edges
- num_token_sentence (int): minimum number of tokens per sentence; sentences with too less tokens are removed

In order to train the GraphSum model on sentence-level and paragraph-level respectively, we provide the following bash scripts in the script folder:
`scripts/run_graphsum_local_multinews_paragraphs.sh` and `scripts/run_graphsum_local_multinews_sentences.sh`

Note that the scripts must be started from the root folder. The bash scripts itself first run the bash scripts provided in the env_local folder and afterwards read the configuration from the model_config folder. The model_config itself defines the path to the data, therefore we provide a separate model_config file for sentence-level and paragraph-level. Within the bash scripts additional parameters for the PaddlePaddle framework are set. Afterwards, the python script `src/launch.py` is called, which invokes multiple instances of the `src/run.py` script, required for parallel training. The `src/run.py` script itself contains the code for the GraphSum model provided by Li et al. The pre-processed data is used as an input for the GraphSum model. The following parameters are used by the `src/run.py` script in order to pad and truncate the input data:

[2] `https://github.com/andersjo/pyrouge/tree/master/tools/ROUGE-1.5.5`

```
### Truncation / Padding parameters
python ./src/run.py \
                .... \
                --max_para_num 30 \ ### maximal number of textual units for a single
                ↪  example
                --max_para_len 60 \ ### maximal number of token for each textual unit
                ...
```

While the following parameters and configurations are used by PaddlePaddle, which do we did not explicitly change for our research:

```
### Used to configure the distributed learning for PaddlePaddle
distributed_args="--node_ips ${PADDLE_TRAINERS} \
                --node_id ${PADDLE_TRAINER_ID} \
                --current_node_ip ${POD_IP} \
                --selected_gpus 1,3,5 \
                --split_log_path log/rq1_paragraphs \
                --nproc_per_node 3"

./src/launch.py $(distributed_args) ...
```

```
### Command-Line arguments passed to src/run.py indicating if GPU training and if
↪  distributed Training should be done.
python ./src/run.py \
                .... \
                --use_cuda true \
                --is_distributed True \
```

As mentioned in our scientific report, we set the parameters to 30,60 for paragraph- and 60,30 for sentence-level respectively. An increase of these parameters results in higher memory consumption while training the model. Additionally, the batch size used by the model can be altered in the bash script by using the batch_size parameter.

The model itself generates summaries, which are saved to directory specified by the decode_path argument, which is parsed to the `src/run.py` script. The results of the ROUGE script are printed into the log files.

3.2 Source origin analysis

For our second research question we require a paragaph-level representation of the MultiNews and WikiSum dataset. For the MultiNews dataset we pre-process the dataset in the same way as done for the first research question. For the purpose of analyzing the presence of positional bias in generated summaries, we need to conserve the information, which input paragraph corresponds to which input document. Therefore we extended the `src/data_preprocess/graphsum/preprocess_graphsum_data.py` script. To pre-process the MultiNews dataset, we provide the `scripts/preprocess_multinews.sh` and `scripts/preprocess_wikisum.sh` scripts. The WikiSum dataset does not contain this meta information anymore. Therefore we are not able to anaylze the presence of positional bias for this dataset. In order to not run into an error when training the GraphSum model, dummy data is added to each input set, which does not have any side effect. This is done in `scripts/preprocess_wikisum.sh`.

Figure 1 displays the attention mechanism of the decoding layers of the GraphSum architecture. For our analysis we only consider the global graph attention as mentioned in our scientific report. Therefore, we needed to extract the global attention weights g_t at each step of the decoding process from the model. This is achieved by altering the paddle-paddle code to propagate the necessary information to the output of the model:

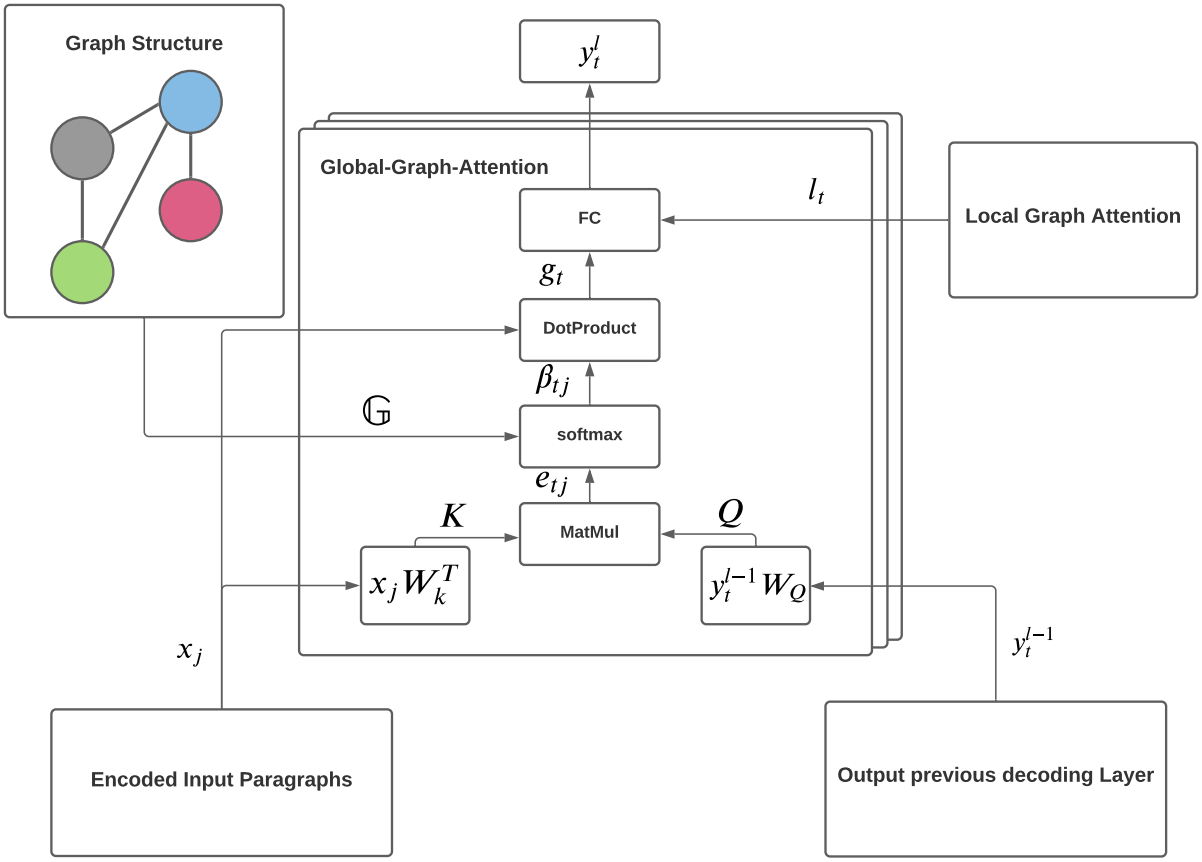

Figure 1: Global Graph Attention mechanism within a single decoding layer. Multi-Heads are indicated by stacked boxes.

Firstly we modified `src/models/attention.py` in order to save save global attention weights of each decoding layer to a persistent array, which can be accessed within `src/networks/graphsum/graphsum_model.py`, where the inference process of the GraphSum model takes place.

```
layers.array_write(weights, layer_id, attention_weights_array)
```

This enables the GraphSum model to access the global attention weights g_t within the inference process. The attention_weights_array does contain the attention weights of all decoding layers for a single step. Therefore we concatenate this array into a tensor, which is done in `src/networks/graphsum/graphsum_model.py`:

```
# Iterate over all Decoding Layers except the first one and concatinate them.
for i in range(1, self._dec_n_layer):

    # Read Attention Weights of Layer i from 'attention_weights_array'
    layer_i = layers.array_read(
        attention_weights_array, layers.fill_constant(shape=[1], value=i,
        ↪  dtype="int64"))

    input_i = layers.reshape(
        layer_i, shape=[-1, 1, 1, 1, self._n_head, self.max_para_num])

```

```
    # Check if first step_idx
    with ie.true_block():
        reshaped_layer_i = ie.input(input_i)
        # Expand 'reshaped_layer_i' for first step, becuase only 1 beam is active
        reshaped_layer_i = layers.expand(
            reshaped_layer_i, expand_times=[1, self.beam_size, 1, 1, 1, 1])

        layers.assign(reshaped_layer_i,
                      attention_weights_single_decoding)
        ie.output(reshaped_layer_i)
    with ie.false_block():
        reshaped_layer_i = ie.input(input_i)

        reshaped_layer_i = layers.reshape(
            reshaped_layer_i, shape=[-1, self.beam_size, 1, 1, self._n_head,
            ↪  self.max_para_num])

        layers.assign(reshaped_layer_i,
                      attention_weights_single_decoding)
        ie.output(reshaped_layer_i)

    # Concat Attention Weights of current decoding layer to the already existing tensor
    attention_weights_all_decoding = layers.concat(
        input=[attention_weights_all_decoding, attention_weights_single_decoding],
        ↪  axis=3)
```

This results in a single tensor for each decoding step. As the GraphSum model works as an auto-regressive decoder, we need to save those tensors temporarly into an array to finally concatenate into a tensor over all decoding steps.

```
# After current step_idx write 'attention_weights_all_decoding' to
↪  'attention_weights_array_all_steps'
                layers.array_write(
                    attention_weights_all_decoding, step_idx,
                    ↪  attention_weights_array_all_steps)
```

```
# Concatinate arrays for each step_idx
with while_oper.block():

    index = layers.elementwise_sub(number_steps, step_idx)

    attention_weights_i = layers.array_read(
        attention_weights_array_all_steps, index)
    if self.extract_local_attention:

        local_attention_weights_i = layers.array_read(
            local_attention_weights_array_all_steps, index)

    parent_idx_i = layers.array_read(
        parrent_idx_array, index)
    pre_id_i = layers.array_read(
        pre_ids_array, index
    )
    scores_i = layers.array_read(
```

```
        scores_array, index)

    layers.increment(step_idx, value=-1, in_place=True)

    padded_attention_weights = layers.pad_constant_like(
        weights_padding_helper, attention_weights_i, pad_value=-1)

    if self.extract_local_attention:
        padded_local_attention_weights = layers.pad_constant_like(
            local_weights_padding_helper, local_attention_weights_i, pad_value=-1)

    padded_parent_idx = layers.pad_constant_like(
        parent_padding_helper, parent_idx_i, pad_value=-1)

    padded_pre_ids = layers.pad_constant_like(
        parent_padding_helper, pre_id_i, pad_value=-1)

    padded_scores = layers.pad_constant_like(
        scores_padding_helper, scores_i, pad_value=-1)

    attention_weights_res = layers.concat(
        [attention_weight_tensor, padded_attention_weights], axis=2)
    if self.extract_local_attention:
        local_attention_weights_res = layers.concat(
            [local_attention_weight_tensor, padded_local_attention_weights], axis=1)

    parent_res = layers.concat(
        [parent_idx_tensor, padded_parent_idx], axis=2)
    pre_id_res = layers.concat(
        [pre_id_tensor, padded_pre_ids], axis=2)

    scores_res = layers.concat(
        [scores_tensor, padded_scores], axis=2)

    layers.assign(parent_res, parent_idx_tensor)
    layers.assign(pre_id_res, pre_id_tensor)
    layers.assign(scores_res, scores_tensor)
    layers.assign(attention_weights_res, attention_weight_tensor)
    if self.extract_local_attention:
        layers.assign(local_attention_weights_res,
                      local_attention_weight_tensor)

    layers.greater_than(step_idx, layers.fill_constant(
        shape=[1], value=0, dtype="int64"), cond)
```

The obtained tensors are the return values of the inference process of the GraphSum model and can be further processed in `src/networks/graphsum/run_graphsum.py`.

The code of `src/networks/graphsum/run_graphsum.py` is then modified to transform the outputs of the inference process into numpy arrays and dictionaries, which are later saved into files.

The same procedure is performed to gather the scores of each generated token sequence, the token sequence itself and a tensor storing parent information about the beams in order to apply a beam search decoder later on. Further information on the code can be extracted from the documentation within the python files.

In order to set the output directory for these files, an additional parameter is expected by the GraphSum model, namely ”attention_weights_path”, which has to be set in `scripts/rq2_multinews.sh` and `scripts/`

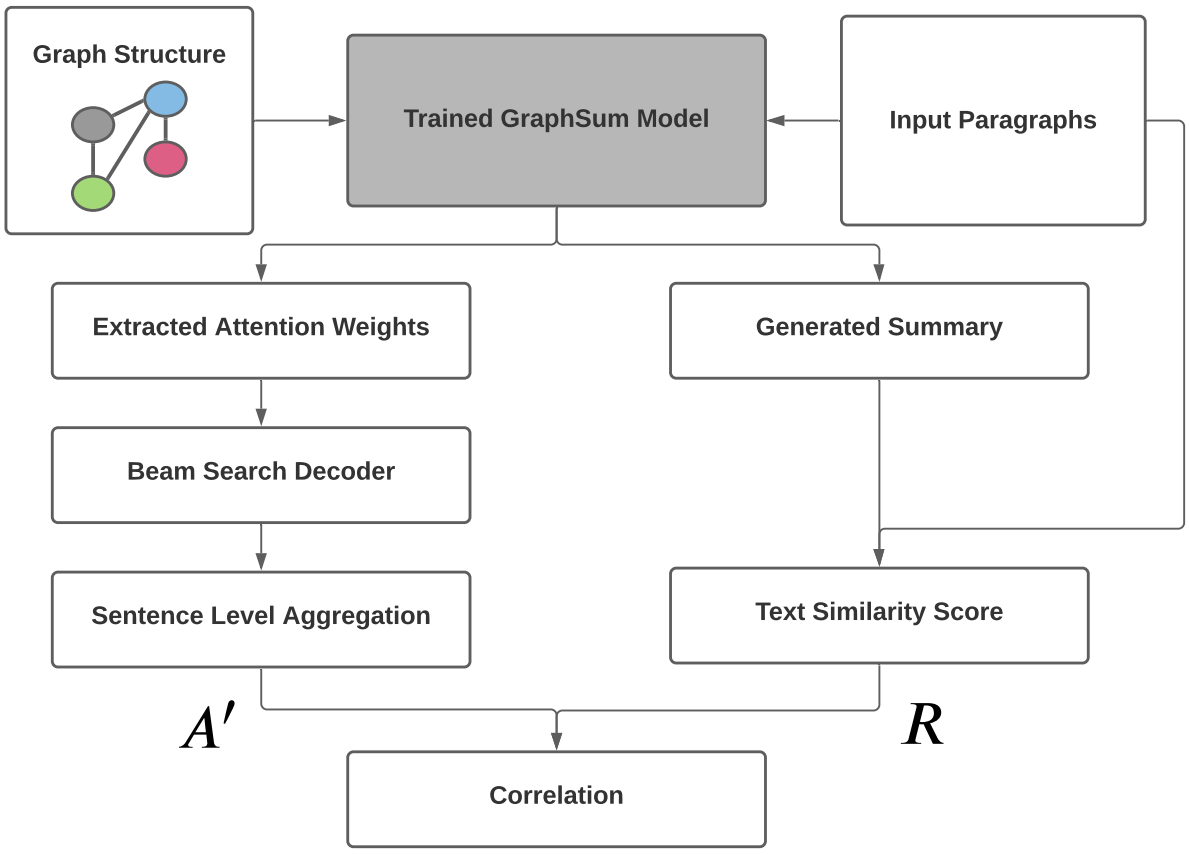

Figure 2: Processing pipeline to calculate correlation between aggregated attention weights distribution A' and reference text similarity metric R.

`rq2_wikisum.sh`.

Figure 2 shows our processing pipeline for our second research question as described in our paper. In the following we explain the individual parts separately. After running the GraphSum model and extracting the attention weights, it is necessary to reconstruct the individual beams by a beam search decoder. For this purpose we implemented the `src/transformation/transform_attention_weights.py` script. This script requires the following parameters:

- input_data (str): path to the directory containing the attention weights
- output_path (str): path to the directory where the transformed attention weights are stored
- max_beam_length (int): maximum beam length set by the GraphSum model
- only_highest_beam (bool): flag indicating if only the beam with the highest probability should be reconstructed (implemented to reduce resource consumption)
- cleanup_data (bool): flag indicating if the directory containing the attention weights should be deleted after the reconstruction is done (implemented to reduce resource consumption)

The script itself contains the following parts:

1. A function, which is used to counteract the issue that the generated beams of examples can be terminated prematurely, resulting in a reduced number of active examples at each decoding step.

```
attention_weights_matrix, scores_matrix, parent_idx_matrix =
↪ reconstruct_attention_weights(attention_weights, parent_idx, scores,
↪ result_dict)
```

2. A function to apply a beam search decoder to transform the attention weights and the scores of the beams. It requires the parent information of each beam.

```
decoded_weight_matrix, decoded_score_matrix =
↪ transform_attention_weights_decoder(attention_weights_matrix, scores_matrix,
↪ parent_idx_matrix, result_dict["longest_beam_array"]-1)
```

3. A function to sort the beams of each individual example by their probability. Afterwards, the first beam is the beam with the highest probability.

```
sorted_weight_matrix, sorted_score_matrix = sort_matrixes(decoded_weight_matrix,
↪ decoded_score_matrix, result_dict["longest_beam_array"]-1, max_beam_length)
```

4. A beam can be terminated prematurely by producing an EOS token. Nevertheless, the model still produces attention weights for this beam, till all other beams of the current example are terminated. Therefore, it is necessary to remove the attention weight information for beams after they have been terminated.

```
cleaned_weight_matrix, cleaned_score_matrix = cleanup_matrix(sorted_weight_matrix,
↪ sorted_score_matrix,result_dict["beam_length"])
```

5. As mentioned in our scientific report, the attention weight of a single generated token may not be useful. Therefore, we aggregate the weight information to sentence-level. This function serves the purpose to extract the information, where each generated sentence end and aggregate the token information accordingly.

```
sentence_level_aggregation =
↪ aggregate_weight_information_for_sentences(cleaned_weight_matrix, result_dict)
```

Finally, the transformed weights, scores and sentence-level aggregated weights are saved to files.

According to Figure 2, the sentence-level aggregated weights correspond to A'. For our analysis we require a reference metric which is based on the ROUGE score. The following steps are performed to obtain this reference metric:

For this purpose `src/rouge_calculation/rouge.py` is provided. The following parameters are necessary:

- can_path (str): path to directory where the file containng the generated summaries of the GraphSum model is stored
- input_data (str): path to directory where the input data of the GraphSum model is stored
- output_dir (str): directory, where the ROUGE scores are saved to
- spm_path (str): path to sentence piece model used as tokenizer in the GraphSum model

First the generated summaries are tokenized using the spm model. This is not required for the input paragraphs as the tokenized version is already stored in the files. This is required in order to split the input paragraphs and generated summaries into sentences.

```
summaries, paragraphs = tokenize(can_path=args.can_path, input_data=args.input_data)
```

Next, the ROUGE score between each sentence of each summary and each sentence of the corresponding input paragraph is calculated. Note that this function invokes the ROUGE perl script.

```
results = extract_rouge(summaries, paragraphs)
```

Additonally, necessary information about summary length with regard to the number of generated sentences is extracted:

```
meta = extract_meta(summaries, paragraphs)
```

Finally, the ROUGE scores and meta information are stored in files.

Following figure 2, the ROUGE scores correspond to R. As a final step we need to calculate the correlation between A' and R. Therefore, we provide the `src/correlation_calculation/correlation_calculation.py` script. The following parameters are necessary:

- rouge_information_path (str): path to directory where information R is stored
- transformed_attention_weights_path (str): path to directory where transformed attention weights are stored
- aggregation_metric (str): metric, which was used to aggregate attention weights from token- to sentence-level (mean/median)
- aggregate_function (str): aggregate function, which aggregated the ROUGE information from sentence- to paragraph-level (np.mean, np.median,...)
- result_output (str): path to directory where correlation results are stored

For each decoding layer we calculate the correlation between the attention weights and the ROUGE scores with the help of the following function:

```
corr, r1, r2, rl, attentions = attention_rouge_correlation(rouge_scores, rouge_meta,
↪  attention_weights, [idx], args.aggregation_metric, aggregate_function)
```

Where idx denotes the current decoding layer. Additionally the script calculates the correlation when considering the first decoding layers as a cluster and the last decoding layers as a cluster:

```
first_layers = np.arange(0,num_decoding_layers/2,1).astype("int") + 1
corr, r1, r2, rl, attentions = attention_rouge_correlation(rouge_scores, rouge_meta,
↪  attention_weights, first_layers, args.aggregation_metric, aggregate_function)

last_layers = np.arange(num_decoding_layers/2,num_decoding_layers,1).astype("int")
corr, r1, r2, rl, attentions = attention_rouge_correlation(rouge_scores, rouge_meta,
↪  attention_weights, last_layers, args.aggregation_metric, aggregate_function)
```

The obtained correlation information is printed to the console and saved to the output directory.

3.3 Visualization

We provide the following jupyter notebooks for the visualization of our results:

1. `src/visualization/analyze-correlation-between-attention-heads.ipynb`: notebook to calculate/visualize correlation between multi-heads and decoding layers with regard to the attention weights
2. `src/visualization/visualize_correlation.ipynb`: notebook used to visualize the correlation between A' and R for different decoding layers
3. `src/visualization/visualize_positional_bias.ipynb`: notebook used to visualize positional bias indicated by A' and R

4 Conclusion

4.1 Limitations

The provided code for the GraphSum model by Li et al.[1] was implemented with the PaddlePaddle framework. The paddlepaddle framework is not commonly used in the english speaking community. Therefore modifying the GraphSum code itself was challenging.

Due to resource limitations we were not able to train a sentence-level model with the batch size, which achieved the best results by Li et al.[1] for paragraph-level. In addition, we were not able to increase the length of the sequence of tokens as inputs for the GraphSum model over 1800.

For our source origin anaylsis we were only able to perform our analysis on a reduced test set with 500 examples.

4.2 Future Work

4.2.1 Sentence- vs. paragraph-level representation

As mentioned in subsection 3.1 we tested pruning techniques in order to achieve better results for sentence-level representation. We investigated this possibility, because truncating of the input data can result in sentences being used as input, which do not provide any useful information. Our pruning techniques resulted in similar average performance, but did not increase the average performance.

As mentioned in our [**scientific-report**] our analysis should be performend on a MDS dataset of a different domain, which uses paragraphs as a tool to split texts into different topics.

4.2.2 Source origin analysis

Our analysis for the source origin analysis was only based on the global attention weights within the transformer. This was the result of limited resources. Nevertheless our approach could also be applied to extract the local attention weights and combine this information with the global attention weights to achieve an even more sophisticated metric to indicate source origin. All of our research can also be applied to other graph-transformer-based models, which are implemented with different frameworks. For this task only the attention weights must be propagated to the output of the models. If this is achieved our further processing pipeline for source origin analysis can be utilized.

5 References

References

[1] Wei Li et al. *Leveraging Graph to Improve Abstractive Multi-Document Summarization*. 2020. arXiv: 2005.10043 [cs.CL].

[2] 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://www.aclweb.org/anthology/W04-1013.

[3] Ma Yanjun et al. "PaddlePaddle: An Open-Source Deep Learning Platform from Industrial Practice". In: *Frontiers of Data and Computing* 1.1, 105 (2019), p. 105. DOI: 10.11871/jfdc.issn.2096.742X.2019.01.011. URL: http://www.jfdc.cnic.cn/EN/abstract/article_2.shtml.