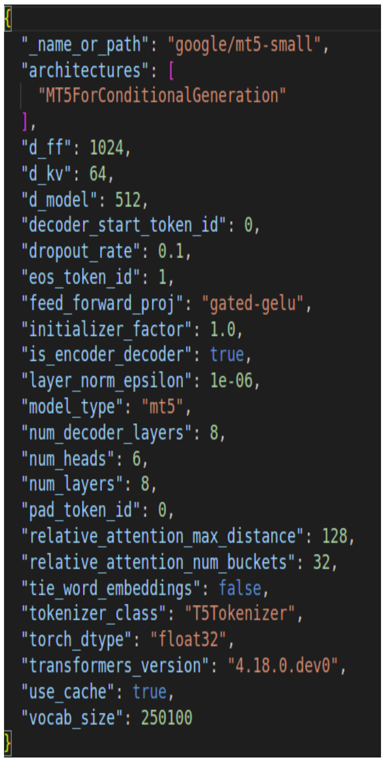
ADL 2022 Fall Hw3

Q1: Model

a. **Model**



```
{
  "_name_or_path": "google/mt5-small",
  "architectures": [
    "MT5ForConditionalGeneration"
  ],
  "d_ff": 1024,
  "d_kv": 64,
  "d_model": 512,
  "decoder_start_token_id": 0,
  "dropout_rate": 0.1,
  "eos_token_id": 1,
  "feed_forward_proj": "gated-gelu",
  "initializer_factor": 1.0,
  "is_encoder_decoder": true,
  "layer_norm_epsilon": 1e-06,
  "model_type": "mt5",
  "num_decoder_layers": 8,
  "num_heads": 6,
  "num_layers": 8,
  "pad_token_id": 0,
  "relative_attention_max_distance": 128,
  "relative_attention_num_buckets": 32,
  "tie_word_embeddings": false,
  "tokenizer_class": "T5Tokenizer",
  "torch_dtype": "float32",
  "transformers_version": "4.18.0.dev0",
  "use_cache": true,
  "vocab_size": 250100
}
```

There are 2 main stages for transformer to work on text summarization.

i. **forward:**
This stage will treat the output of source text tokenizer as the input of encoder which is multi-layer. Then, the output of the last encoder will be used as hidden states of decoder, and the input will be the target text tokenizer output. In this way, the result of the decoder is the probability of each subword.

ii. **generate:**
In this stage, we will take the previous result from decoding as input, and generate the summarization.

b. **Preprocessing**

T5 tokenizer is basically sentencepiece, which is composed of byte pair encoding and unigram language model. It will first divide the words into subwords(the smallest unit of chinese word is a character), select the path with highest probability based on viterbi algorithm, and eventually do tokenization.

Q2: Training

a. Hyperparameter

i. **optimizer:** To reduce the GPU storage usage, I use **Adafactor** instead of Adam.

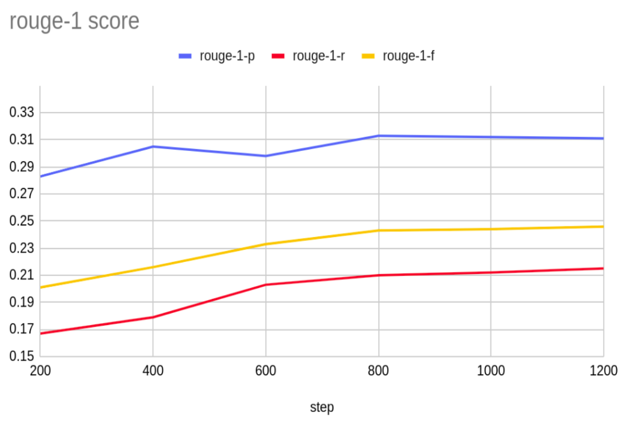
ii. **learning rate:** 0.001

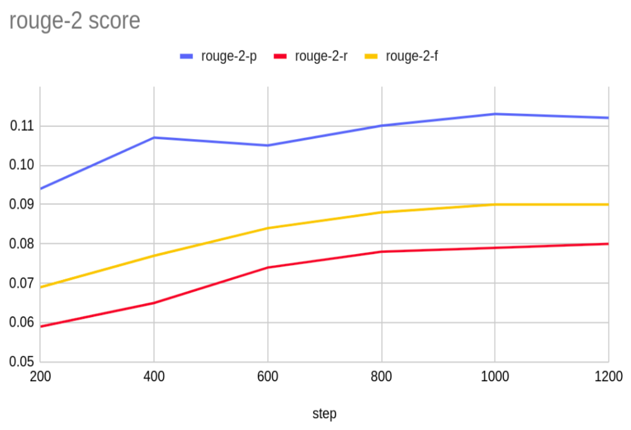
iii. **batch size:**
batch size = 4, gradient accumulation step = 4
To reduce the usage of GPU storage, we can't apply too large batch size but if we use a small one, it can lead to terrible result. Hence, I also apply gradient accumulation

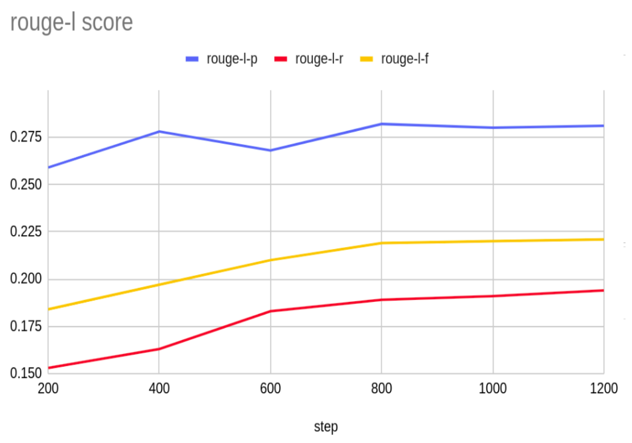
iv. **text size:**
256/16 for input/output
It helps to reduce the usage of GPU storage, which it's recommended by TA.

v. **fp16:** It helps to reduce the usage of GPU storage.

b. Learning curves







Q3: Generation Strategies

a. Strategies

i. **Greedy:**
Pick the one with max $P(w|w_1...w_{t-1})$ at each time stamp

ii. **Beam Search:**
Beam search will pick the most likely *num beam* of candidates at each time step, and choose the one with overall highest probability.

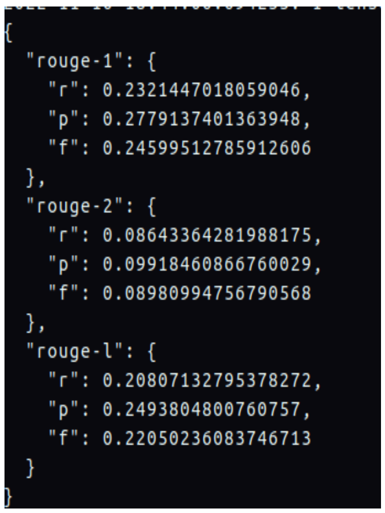
iii. **Top-k Sampling:**
In top-k sampling, only the *top-k* most likely next word are filtered and redistributed to sample the next word.

iv. **Top-p Sampling:**
Instead of choosing only *top-k* words, top-p sampling choose from the smallest possible set of words whose cumulative probability exceeds the probability *top-p* and redistributes the set of words.

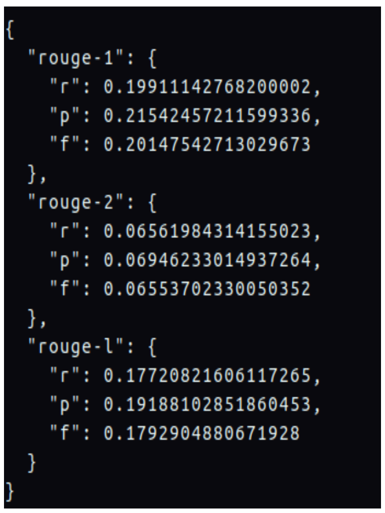
v. **Temperature:**
It changes the final probability distribution. The higher the value of temperature, the softer the distribution is. "Softer" means it the model will basically be less confident about it's prediction.

b. Hyperparameters

i. **Greedy vs not Greedy(do sample)**
Greedy strategy has better performance because sample strategy could pick worse choice and thus affects decoding.



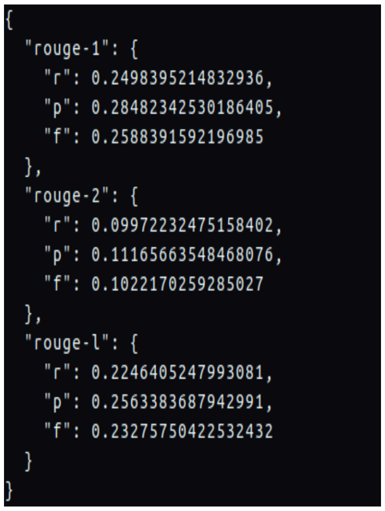
```
{
  "rouge-1": {
    "r": 0.2321447018059046,
    "p": 0.2779137401363948,
    "f": 0.24599512785912606
  },
  "rouge-2": {
    "r": 0.08643364281988175,
    "p": 0.09918460866760029,
    "f": 0.08980994756790568
  },
  "rouge-l": {
    "r": 0.20807132795378272,
    "p": 0.2493804800760757,
    "f": 0.22050236083746713
  }
}
```



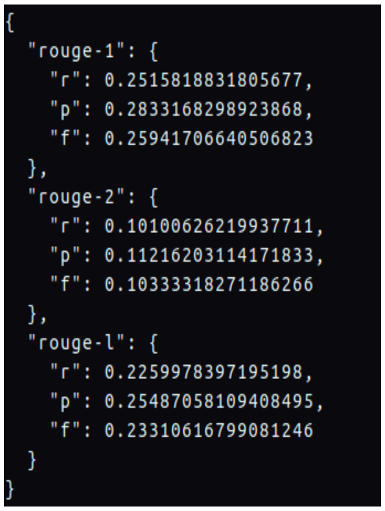
```
{
  "rouge-1": {
    "r": 0.19911142768200002,
    "p": 0.21542457211599336,
    "f": 0.20147542713029673
  },
  "rouge-2": {
    "r": 0.06561984314155023,
    "p": 0.06946233014937264,
    "f": 0.06553702330050352
  },
  "rouge-l": {
    "r": 0.17720821606117265,
    "p": 0.19188102851860453,
    "f": 0.1792904880671928
  }
}
```

ii. **num beam=5 vs num beam=10**

The result will be better than greedy strategy because beam search has the concept of candidates which is more tolerant than greedy. Moreover, if n=10, the result could be slighty better than n=5.



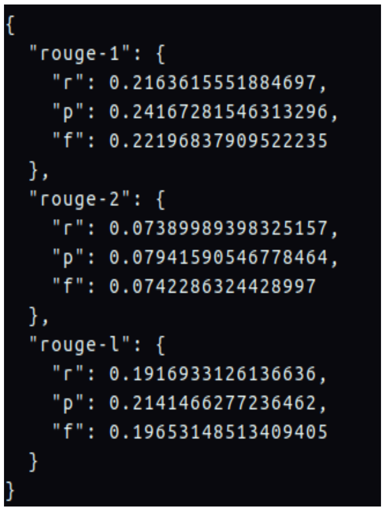
```
{
  "rouge-1": {
    "r": 0.2498395214832936,
    "p": 0.28482342530186405,
    "f": 0.2588391592196985
  },
  "rouge-2": {
    "r": 0.09972232475158402,
    "p": 0.11165663548468076,
    "f": 0.1022170259285027
  },
  "rouge-l": {
    "r": 0.2246405247993081,
    "p": 0.2563383687942991,
    "f": 0.23275750422532432
  }
}
```



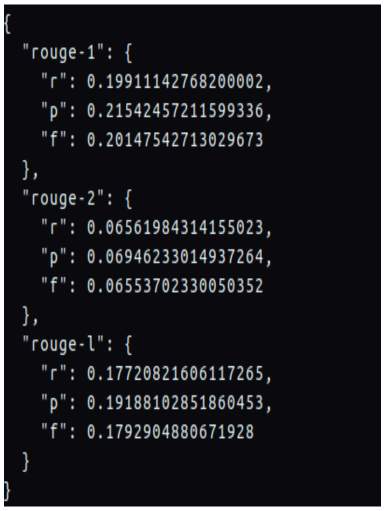
```
{
  "rouge-1": {
    "r": 0.2515818831805677,
    "p": 0.2833168298923868,
    "f": 0.25941706640506823
  },
  "rouge-2": {
    "r": 0.10100626219937711,
    "p": 0.11216203114171833,
    "f": 0.10333318271186266
  },
  "rouge-l": {
    "r": 0.2259978397195198,
    "p": 0.25487058109408495,
    "f": 0.23310616799081246
  }
}
```

iii. **top_k=10 vs top_k=50**

The experiment with top_k will be better than that of sample. However, if the value of top_k is too large, the result is almost the same.



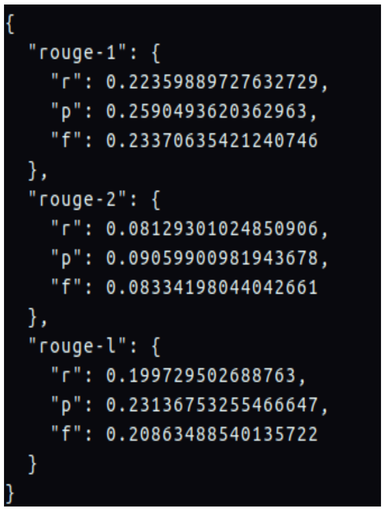
```
{
  "rouge-1": {
    "r": 0.2163615551884697,
    "p": 0.24167281546313296,
    "f": 0.22196837909522235
  },
  "rouge-2": {
    "r": 0.07389989398325157,
    "p": 0.07941590546778464,
    "f": 0.0742286324428997
  },
  "rouge-l": {
    "r": 0.1916933126136636,
    "p": 0.2141466277236462,
    "f": 0.19653148513409405
  }
}
```



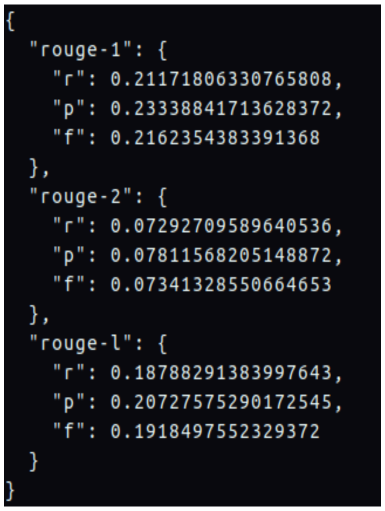
```
{
  "rouge-1": {
    "r": 0.19911142768200002,
    "p": 0.21542457211599336,
    "f": 0.20147542713029673
  },
  "rouge-2": {
    "r": 0.06561984314155023,
    "p": 0.06946233014937264,
    "f": 0.06553702330050352
  },
  "rouge-l": {
    "r": 0.17720821606117265,
    "p": 0.19188102851860453,
    "f": 0.1792904880671928
  }
}
```

iv. **top_p=0.6 vs top_p=0.9**

Top_p strategy has better performance than sample and top_k because the value of each output is seldom large. Thus, it will be effective to use top_p to select.



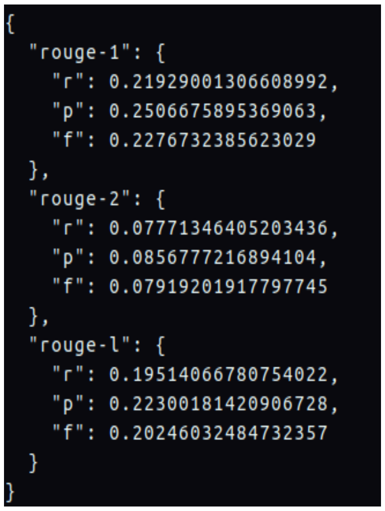
```
{
  "rouge-1": {
    "r": 0.22359889727632729,
    "p": 0.2590493620362963,
    "f": 0.23370635421240746
  },
  "rouge-2": {
    "r": 0.08129301024850906,
    "p": 0.09059900981943678,
    "f": 0.08334198044042661
  },
  "rouge-l": {
    "r": 0.19972950268876З,
    "p": 0.23136753255466647,
    "f": 0.20863488540135722
  }
}
```



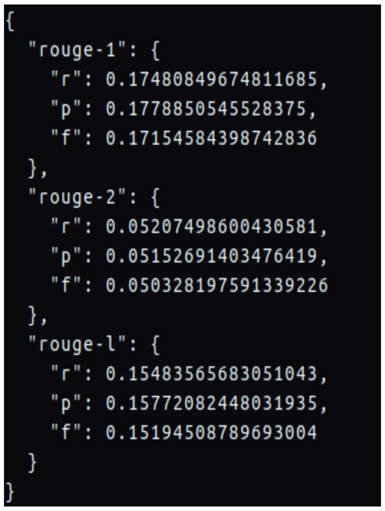
```
{
  "rouge-1": {
    "r": 0.21171806330765808,
    "p": 0.23338841713628372,
    "f": 0.2162354383391368
  },
  "rouge-2": {
    "r": 0.07292709589640536,
    "p": 0.07811568205148872,
    "f": 0.07341328550664653
  },
  "rouge-l": {
    "r": 0.18788291383997643,
    "p": 0.20727575290172545,
    "f": 0.1918497552329372
  }
}
```

v. **temperature=0.7 vs temperature=1.3**

If temperature¡1, the performance can be better than sample. In contrast, in the condition of temperature¿1, the performance is much worse. I think it's because the output is originally smooth enough. Therefore, if we use temperature ¿ 1, which makes it even smoother, it may make our selection ambiguous.



```
{
  "rouge-1": {
    "r": 0.21929001306608992,
    "p": 0.2506675895369063,
    "f": 0.2276732385623029
  },
  "rouge-2": {
    "r": 0.07771346405203436,
    "p": 0.0856777216894104,
    "f": 0.07919201917797745
  },
  "rouge-l": {
    "r": 0.19514066780754022,
    "p": 0.22300181420906728,
    "f": 0.20246032484732357
  }
}
```



```
{
  "rouge-1": {
    "r": 0.17480849674811685,
    "p": 0.1778850545528375,
    "f": 0.17154584398742836
  },
  "rouge-2": {
    "r": 0.05207498600430581,
    "p": 0.05152691403476419,
    "f": 0.050328197591339226
  },
  "rouge-l": {
    "r": 0.15483565683051043,
    "p": 0.15772082448031935,
    "f": 0.15194508789693004
  }
}
```

My final generation strategy is **beam search** with *num beam = 10*