Summarization Report

資管三 陳惟中 B06705014

I. Data Processing

1. Tokenize / Truncate

建立en_core_web_sm語言模型 --- spaCy模組化處理文字

Padding sample to the same length(max length) with ignore idx=-100

- spaCy將所有字轉換成lower case `Tokenizer(lower=config['lower_case'])`
- 對文字分詞(原始文字在空格處分割) `words = tokenizer.collect_words(documents)`
- 保留具有意義的字詞
- 匹配特殊規則 `disable=['tagger', 'ner', 'parser', 'textcat']`
- Tokenizer Definition:

```
class Tokenizer:
    def __init__(self, vocab=None, lower=False):
        self.nlp = spacy.load('en_core_web_sm',
                              disable=['tagger', 'ner', 'parser', 'textcat'])
        self.lower = lower
        if vocab is not None:
            self.set_vocab(vocab)
```

- Tokenizer is called from:

```
logging.info('Collecting words in documents...')
tokenizer = Tokenizer(lower=config['lower_case'])
words = tokenizer.collect_words(documents)
```

2. Embedding

使用GloVe library (./glove.840B.300d.txt) --- 根據語料庫(corpus)構建一個共現矩陣(Co-ocurrence Matrix),並計算衰減函數(decreasing weighting)

距離越遠的兩個單詞所佔總計數的權重越小

- 將300維glove檔存成embedding.pkl
- 將train/valid.jsonl的文字部分(text, summary)轉成300維度vector(embed_size=300)之後,存成三個.pkl,放到./seq_tag & ./seq2seq 資料夾中
- 在predict test的時候,讀入的file是jsonl檔,所以利用embedding.pkl / tokenizer.pkl 轉換成dataset讀進去給eval.py使用
- $embed = Embedding(embedding_path, words, rand_seed)$, where embedding_weight is from embedding.pkl generated by glove
- Embedding Definition:

```
class Embedding:
    def __init__(self, embedding_path, words=None, rand_seed=524,
                 special_tokens=['<pad>', '<s>', '</s>', '<unk>']):
        if words is not None:
            words = set(words)

        self.vocab = special_tokens
        vectors = [[] for _ in special_tokens]
        with open(embedding_path) as fp:
            ...
        random.seed(rand_seed)
        for i in range(len(special_tokens)):
            if len(vectors[i]) == 0:
                dim = len(vectors[-1])
                vectors[i] = [random.random() * 2 - 1 for _ in range(dim)]
        self.vectors = torch.tensor(vectors)
```

- Embedding is called from:

```
embedding = Embedding(config['embedding'], words=words)
```

II. Describe extractive summarization model

1. Model

```
class Encoder(nn.Module):
    def __init__(self,
                 embedding_path,
                 embed_size,
                 rnn_hidden_size) -> None:
        super(Encoder, self).__init__()
        with open(embedding_path, 'rb') as f:
            embedding = pickle.load(f)
        embedding_weight = embedding.vectors
        self.embedding = nn.Embedding.from_pretrained(embedding_weight)
        self.rnn = nn.LSTM(embed_size, rnn_hidden_size,batch_first=True)
        # init a LSTM/RNN
        self.embed_size = embed_size
        self.hidden_size = rnn_hidden_size

    def forward(self, idxs) -> Tuple[torch.tensor, torch.tensor]:
        embed = self.embedding(idxs) #idx is pretrained weight
        output, state = self.rnn(embed)
        return output, state


class SeqTagger(pl.LightningModule):
    def __init__(self, hparams) -> None:
        super(SeqTagger, self).__init__()
        self.hparams = hparams
        self.criterion = nn.BCEWithLogitsLoss(
            reduction='none',
            pos_weight=torch.tensor(hparams.pos_weight))
        self.encoder = Encoder(hparams.embedding_path, hparams.embed_size,
                               hparams.rnn_hidden_size)
        self.proj = nn.Linear(hparams.rnn_hidden_size, 1)

    def forward(self, idxs) -> torch.tensor:
        output, state = self.encoder(idxs)
        logit = self.proj(output).squeeze()
        return logit
```

Description:

- Encoder是利用LSTM的model：LSTM使用記憶來加強當前的決策，利用三個Gate來決定記憶的儲存與使用，performance通常比RNN好
- pos_weight=5 控制false negatives的數量
- Epoch size設為10，在實際train的過程中loss值有逐漸收斂，應為有效training
- $h_t, c_t = LSTM(w_t, h_{t-1}, c_{t-1})$, where w_t is the word embedding of the t-th token

2. Performance of the model

```
{
    "mean": {
        "rouge-1": 0.19137325805324057,
        "rouge-2": 0.02864821417481478,
        "rouge-l": 0.1309634257577266
    },
    "std": {
        "rouge-1": 0.07903362080513979,
        "rouge-2": 0.04089177076027065,
        "rouge-l": 0.0563171633859225
    }
}
```

3. Loss function

```
self.criterion = nn.BCEWithLogitsLoss(
    reduction='none',
    pos_weight=torch.tensor(hparams.pos_weight))
```

- Loss function is called from:

```
def _calculate_loss(self, y_hat, y) -> torch.tensor:
  loss = self.criterion(y_hat,y)
  mask = y.ne(-100)
  loss = torch.masked_select(loss,mask)
  return loss.mean()
```

> **Description:**
>
> - BCEWithLogitsLoss就是把Sigmoid-BCELoss合成
> - 計算loss值的時候，先透過 `masked_select()` 把padding的值（idx=-100）拿掉，以免錯估loss
> - 最後return的值是取loss.mean()，納入極端值影響
> - $loss = BCEWithLogitsLoss(W_{pos})$

4. Method

Optimization algorithm: adam

```
def configure_optimizers(self) -> torch.optim.Optimizer:
    return torch.optim.Adam(self.parameters())
```

> **Description:**
>
> - Momentum+RMSprop+各自做偏差的修正
> - 計算參數更新方向前會考慮前一次參數更新的方向
> - 在學習率上依據梯度的大小對learning rate進行加強或是衰減

Learning rate

adam default lr=0.0001, changing over time considering the gradient

Batch size

batch_size=８ 在記憶體可以負擔下，足夠的batch_size值

5. Post-processing strategy

計算完sentence_probability後，取機會最高的兩個sentences，作為extractive summary的結果

III. Describe seq2seq + attention summarization model

1. Model

```
class Encoder(nn.Module):
    def __init__(self,emb_dim,enc_hid_dim,dec_hid_dim,dropout):
        super().__init__()

        with open(EMBEDDING_PATH, 'rb') as f:
            embedding = pickle.load(f)
            self.input_dim=len(embedding.vocab)
        weight = embedding.vectors
        self.embedding = nn.Embedding.from_pretrained(weight)

        self.rnn = nn.GRU(emb_dim, enc_hid_dim,
                          bidirectional = True)

        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        outputs, hidden = self.rnn(embedded)
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:],
                          hidden[-1,:,:]), dim = 1)))
        return outputs, hidden
```

Description:

- Encoder是利用GRU的model：GRU利用設計改良後的神經單元，相較於LSTM，GRU加快執行速度及減少記憶體的耗用
- 雙向的GRU model: 讓我們可以從以前的表述中學習之外，也可以從未來的表述中學習
- Epoch size設為10，在實際train的過程中loss值有逐漸收斂，應為有效training
- $c_t, h_t = GRU(w_t)$, where w_t is the word embedding of the t-th token

2. Performance of the model

```
{
    "mean": {
        "rouge-1": 0.035960113597316504,
        "rouge-2": 0.0001530690503995818,
        "rouge-l": 0.03360296147219497
    },
    "std": {
        "rouge-1": 0.05205903212326727,
        "rouge-2": 0.0033338835059659572,
        "rouge-l": 0.04731458126013844
    }
}
```

3. Loss function

```
criterion = nn.CrossEntropyLoss(ignore_index = 0)
```

- Loss function is called from:

```
def train(model, iterator, optimizer, criterion, clip):

  ...

  epoch_loss = 0

  for i, batch in enumerate(tqdm(iterator)):

      loss = criterion(output, trg)

      loss.backward()

      torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

      optimizer.step()

      epoch_loss += loss.item()

  return epoch_loss / len(iterator)
```

> **Description:**
>
> - nn.CrossEntropyLoss()是nn.logSoftmax()和nn.NLLLoss()的合成
> - $loss(x) = weight_c(-x_c + log(\sum_j exp(x[j])))$, where x is input vector, c is class of the vector

4. Method

Optimization algorithm

```
optimizer = optim.Adam(model.parameters())
```

> **Description:**
>
> - Momentum+RMSprop+各自做偏差的修正
> - 計算參數更新方向前會考慮前一次參數更新的方向
> - 在學習率上依據梯度的大小對learning rate進行加強或是衰減

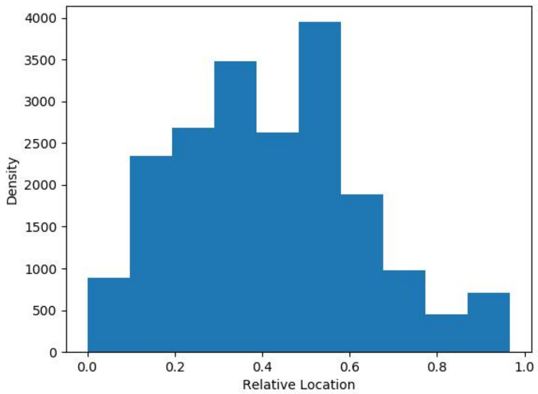
Learning rate

adam default lr=0.0001, changing over time considering the gradient

Batch size

batch_size=8 在記憶體可以負擔下, 足夠的batch_size值

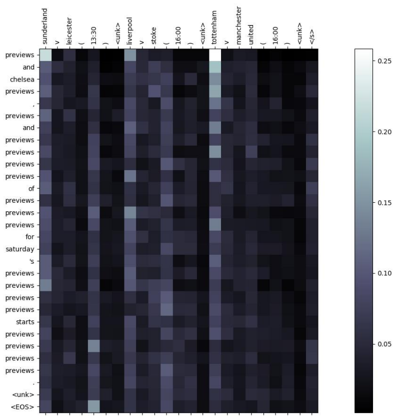
IV. Plot the distribution of relative location

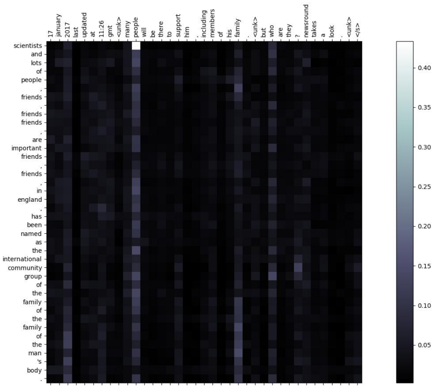


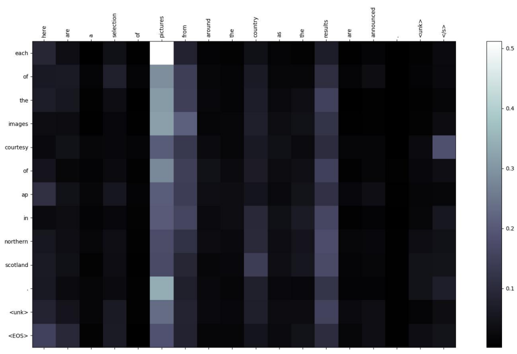
Description:

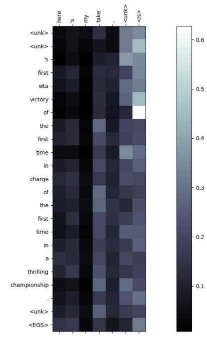
- 畫出的historgram有右尾的特徵, 標示文章前半部分的句子較容易成為summary的候選句子

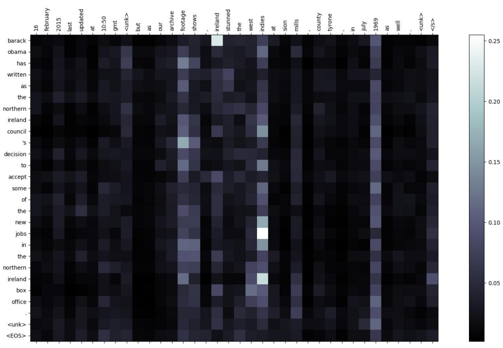
V. Visualize the attention weights

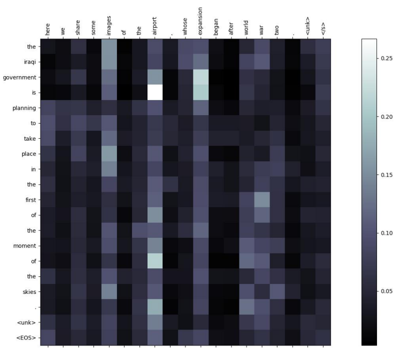


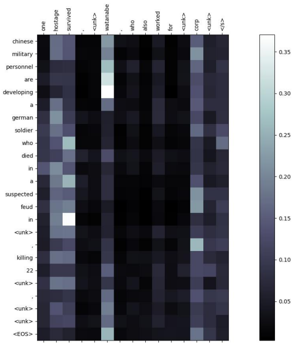


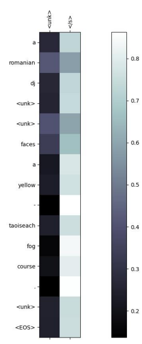












Description:

- 畫出的heatmap圖大致會有一條顏色較深的線，表示attention weight會隨著source被讀進去的順序，依序加重weight，讓output因時間順序性輸出結果

VI. Explain Rouge-L

Take the union LCS(longest common subsequence) matches between a reference summary sentence, r_i, and every candidate summary sentence, c_j

$R_{lcs} = \frac{\sum_{i=1}^{u} LCS_{\bigcup}(r_i, C)}{m}$ reference summary with u sentences containing a total of m words

$P_{lcs} = \frac{\sum_{i=1}^{u} LCS_{\bigcup}(r_i, C)}{m}$ candidate summary with v sentences containing a total of n words

$F_{lcs} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$

β is set to a very big number => only R_{lcs} is considered.

$LCS_{\bigcup}(r_i, C)$ is the LCS score of the union longest common subsequence between reference sentence r_i and candidate summary C