Assignment: Medical Keyword Extraction

TARUN KUMAR
IIIT HYDERABAD

The assignment focuses on creating a program to extract keywords from medical transcriptions. The extracted keywords serve the purpose of summarizing the

content, improving searchability, and facilitating efficient information retrieval within the medical domain. To achieve this, a substantial collection of transcribed medical reports covering diverse specialties will be employed for development and testing.

The essence of keyword extraction lies in its ability to distill crucial terms or phrases from the complex medical transcriptions. This functionality is particularly valuable for quickly grasping the main points of medical content, streamlining the search process, and aiding practitioners in accessing relevant information promptly. The underlying approach involves leveraging Natural Language Processing (NLP) techniques.

```
def tokenize(text):
    # text = text.strip()
    return word_tokenize(text)

def get_lower(text):
    return text.lower()

def remove_punctuations(text):
    return ''.join([char for char in text if char not in punct])

def remove_alpha_numeric(sentence):
    words = sentence.split()
    alphabetic_words = [word for word in words if word.isalpha()]
    return ' '.join(alphabetic_words)

def remove_tags(text):
    return BeautifulSoup(text, 'html.parser').get_text()

def remove_extra_gaps(text):
    return ' '.join(text.split())

def pipeline(text):
    text = get_lower(text)
#     text = remove_punctuations(text)
    text = remove_extra_gaps(text)
```

PREPROCESSING STEP

Let's break down the key steps in the pipeline:

Data Filtering:

- The DataFrame medical_df is filtered to exclude rows where either the 'transcription' or 'keywords' column has missing values (NaN).

Text Tokenization:

- A tokenization function (tokenize) is defined, likely to break down the text into individual words or tokens. This step is crucial for subsequent text processing.

Text Lowercasing:

- Another function (get_lower) is created to convert all text to lowercase. This normalization ensures uniformity and consistency in the text data.

Punctuation Removal:

- Although the code for removing punctuation is present but commented out, it seems that this step could be a part of the pipeline. Removing punctuation can be beneficial for text analysis, especially if the focus is on individual words.

Alphanumeric Removal:

- A function (remove_alpha_numeric) is defined to filter out non-alphabetic words from the text. This step might be useful to eliminate numerical or alphanumeric content, emphasizing purely linguistic information.

HTML Tag Removal:

- The remove_tags function, leveraging BeautifulSoup, is applied to eliminate HTML tags from the text. This step is essential when dealing with transcriptions that might contain HTML formatting.

Whitespace Normalization:

- The remove_extra_gaps function aims to normalize extra whitespaces in the text, reducing multiple consecutive spaces to a

single space. This helps in maintaining a clean and consistent text structure.

Overall Data Processing Pipeline:

- The pipeline function is designed to integrate these preprocessing steps by applying text lowercase conversion and whitespace normalization. The removal of punctuation and alphanumeric characters is commented out, suggesting potential customization based on specific requirements.

FINE-TUNING BART FOR KEYWORD EXTRACTION

BART (Bidirectional and Auto-Regressive Transformers) is a transformer-based neural network architecture designed for sequence-to-sequence tasks. Introduced by Facebook AI, BART excels in natural language processing applications, such as text summarization and language generation. Notably, it employs a denoising autoencoder objective during pre-training, where it learns to reconstruct a corrupted input sequence. BART has demonstrated state-of-the-art performance in various language generation tasks, making it a powerful model for tasks requiring sequence transduction and generation.

Fine-tuning BART for keyword extraction involves adapting the pre-trained BART model to specifically identify and extract keywords from text data. The process typically includes customizing the model's training on a dataset containing examples of medical transcriptions and associated keywords. By fine-tuning on this task-specific data, BART can learn to generate accurate and relevant keywords, providing a tailored solution for keyword extraction in the medical domain. The fine-tuned model can then be applied to new medical transcriptions to automatically extract essential terms, aiding in content summarization and information retrieval.

```
tokenizer = AutoTokenizer.from_pretrained("facebook/bart-base", padding_side="left",
                                          truncation_side='right')
model = BartForConditionalGeneration.from_pretrained("facebook/bart-base").to("cuda")
```

```python
class MedicalDataset(Dataset):
    def __init__(self, df,tokenizer):
        self.df = df
        self.tokenizer = tokenizer

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        transcript=self.df['transcription']
        keywords=self.df['keywords']

        transcript_tokens = self.tokenizer(transcript.iloc[idx],
                                           padding='max_length',
                                           truncation=True,
                                           max_length=1000,
                                           return_tensors='pt')
        transcript_tokens=transcript_tokens['input_ids']

        keyword_tokens = self.tokenizer(keywords.iloc[idx],
                                        padding="max_length", truncation=True,
                                        max_length=50,
                                        return_tensors='pt')
        keyword_tokens=keyword_tokens['input_ids']
```

- **Initialization:**
  - The class takes a DataFrame (df) and a tokenizer as input during initialization.
- **Length Method (__len__):**
  - Returns the total number of samples in the dataset, which corresponds to the length of the DataFrame.
- **Get Item Method (__getitem__):**
  - Retrieves a specific sample at the given index (idx) from the DataFrame.
  - Extracts the transcript and keywords for the selected index.
  - Tokenizes the transcript and keywords using the provided tokenizer.
  - Padding and truncation are applied to ensure consistent input sizes (max length of 1000 for transcripts and 50 for keywords).
  - The tokenized input is converted to PyTorch tensors.
  - The resulting tensors are moved to the GPU (cuda).
- **Returned Values:**
  - Returns a tuple containing the tokenized transcript and keyword tensors for the specified index.

This dataset class is designed to be used with PyTorch's DataLoader to facilitate efficient batching and training of a model, presumably for a task like keyword extraction from medical transcriptions using a BART model or similar

```
def train(data,num_batches, model, optimizer):
    model.train()
    model_loss = 0
    model_acc = 0
    i = 0
    for transcription, keyword in data:
        optimizer.zero_grad()
        out = model(transcription, labels=keyword)

        loss = out.loss
        model_loss += loss.item()

        logits = out.logits
        preds = torch.argmax(torch.softmax(logits,dim=2),dim=2)
        acc = torch.sum(keyword == preds).item()/(keyword.shape[0]*keyword.shape[1])
        model_acc += acc

        loss.backward()
        optimizer.step()

        i+=1

    print(f"loss={model_loss/i} accuracy={model_acc/i}",end="\n")
```

- **Function Parameters:**
    - data: Presumably a DataLoader or an iterable providing batches of training data.
    - num_batches: The total number of batches to process during training.
    - model: The neural network model being trained.
    - optimizer: The optimizer used for updating the model parameters during training.

- **Training Loop:**
    - The function sets the model in training mode using model.train().
    - Iterates through batches of data from data.

    - Zeroes out the gradients with optimizer.zero_grad() before each batch.
    - Passes the input (transcription) through the model, computing the loss with respect to the target (keyword) using the BART-like setup.
    - Backpropagates the loss and performs a parameter update using the optimizer.
    - Computes and accumulates the loss and accuracy metrics for reporting.

- **Accuracy Calculation:**
    - The accuracy is computed by comparing the predicted indices (preds) with the target indices (keyword). It calculates the accuracy by comparing the equality of corresponding elements and then summing and normalizing over the total number of elements.

- **Printing Metrics:**
    - Prints the average loss and accuracy over the training batches.

```
torch.save({
    'epoch': 3,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
}, '/kaggle/working/model.pth')
```

+ Code + Markdown

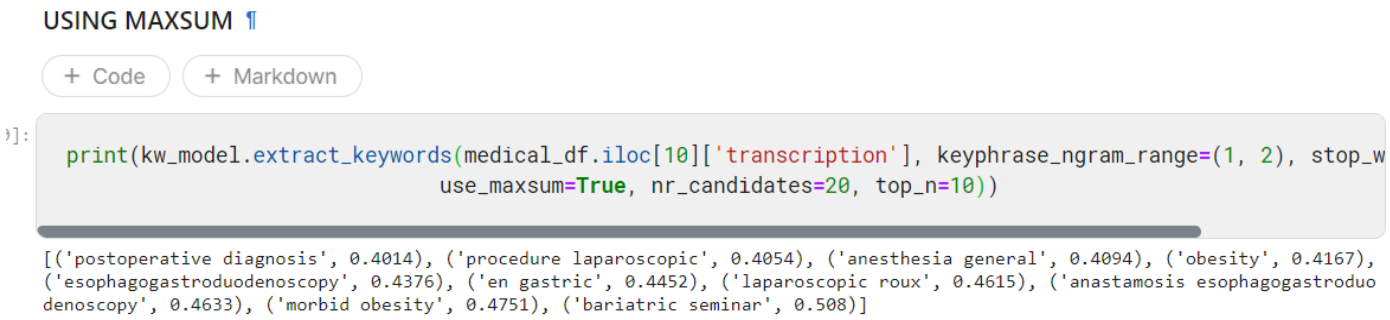
```
def summary(df, transcription, model, tokenizer):
    df['Result'] = df[transcription].apply(lambda x: text(x, model, tokenizer))
    return df

def text(text, model, tokenizer):
    input_ids = tokenizer(text, max_length=1000, padding='max_length', truncation=True, return_tensors='pt'
    generated_ids = model.generate(input_ids, min_length=20, max_length=50)
    decoded_keywords = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)
    return decoded_keywords
```

Key points about the code:

- **text Function:**
  - Takes a text input, a pre-trained model, and a tokenizer.
  - Tokenizing the input text using the provided tokenizer.
  - Passes the tokenized input through the model's generate method, obtaining generated keyword IDs.
  - Decodes the generated keyword IDs using the tokenizer, excluding special tokens.
  - Returns the decoded keywords.

- **summary Function:**
  - Applies the text function to each transcription in the DataFrame using the apply method.
  - Creates a new column 'Result' in the DataFrame, storing the extracted keywords.
  - Returns the modified DataFrame.

KEYWORD EXTRACTION USING KEYBERT



- **Input Text:**
  - The input text for keyword extraction is the transcription from the 10th row of the DataFrame: medical_df.iloc[10]['transcription'].
- **Keyword Extraction Parameters:**
  - keyphrase_ngram_range=(1, 2): Specifies the range of n-grams to consider during keyword extraction, here including unigrams and bigrams.
  - stop_words='english': Indicates that common English stop words should be excluded during keyword extraction.
  - use_maxsum=True: Suggests the utilization of the MaxSum algorithm for keyphrase selection.
  - nr_candidates=20: Specifies the number of candidate keyphrases to consider during the extraction process.
  - top_n=10: Specifies that the top 10 keyphrases should be returned as the final result.

USING MAXIMAL MARGINAL RELEVANCE

```
print(kw_model.extract_keywords(medical_df.iloc[10]['transcription'], keyphrase_ngram_range=(1, 2), stop_w
                                use_mmr=True, diversity=0.7))
```

```
[('obesity postoperative', 0.6884), ('eea anastamosis', 0.2951), ('comorbidities related', 0.1013), ('sequential compression',
0.0419), ('signing consent', 0.0164)]
```

- **Input Text:**
  - The input text for keyword extraction is the transcription from the 10th row of the DataFrame: medical_df.iloc[10]['transcription']
    .
- **Keyword Extraction Parameters:**
  - keyphrase_ngram_range=(1, 2): Specifies the range of n-grams to consider during keyword extraction, here including unigrams and bigrams.
  - stop_words='english': Indicates that common English stop words should be excluded during keyword extraction.
  - use_mmr=True: Suggests the utilization of Maximal Marginal Relevance (MMR) for keyphrase selection. MMR aims to balance between relevance and diversity in the extracted keywords.
  - diversity=0.7: Specifies the diversity parameter for MMR, controlling the trade-off between relevance and diversity. A higher value (e.g., 0.7) encourages more diverse keyword selection.

TEST RESULTS

USING BART

1

BART RESULTS

```
print(results['Result'].iloc[0])
```

```
['ophthalmology, intraocular lens implant, posterior chamber intraocular, phacoemulsification, posterior pole, anterior capsular, capsulorrhexis, max oil catara
```

GROUND TRUTH

```
print(results['keywords'].iloc[0])
```

```
surgery, phacoemulsification, intraocular lens implant, posterior chamber, chamber, eye, intraocular, lens,
```

BART RESULTS

```
print(results['Result'].iloc[1])
```

```
['gastroenterology, ablation of endometriosis, laparoscopy, uterosacral ligament, endometrialosis, endobronchial ligamentation, uteroacral, ligament']
```

Ground truth

```
print(results['keywords'].iloc[1])
```

```
obstetrics / gynecology, ablation of endometriosis, allen-masters window, uterosacral ligament, endometriosis, cul de sac, laparoscopy, lesions, ablati
```

Groundtruth

```
medical_df.iloc[10]['keywords']
```

```
cs, morbid obesity, roux-en-y, gastric bypass, antecolic, antegastric, anastamosis, esophagogastroduodenoscopy, eea, surgidac sutures, roux limb, port, sta
```

USING MAXSUM

```
print(kw_model.extract_keywords(medical_df.iloc[10]['transcription'], keyphrase_ngram_range=(1, 2), stop_words='english',
                                use_maxsum=True, nr_candidates=20, top_n=10))
```

```
[('postoperative diagnosis', 0.4014), ('procedure laparoscopic', 0.4054), ('anesthesia general', 0.4094), ('obesity', 0.4167), ('esophagogastroduodenosco
```

USING MAXIMAL MARGINAL RELEVANCE

```
print(kw_model.extract_keywords(medical_df.iloc[10]['transcription'], keyphrase_ngram_range=(1, 2), stop_words='english',
                                use_mmr=True, diversity=0.7))
```

```
[('obesity postoperative', 0.6884), ('eea anastamosis', 0.2951), ('comorbidities related', 0.1013), ('sequential compression', 0.0419), ('signing consent
```

USING BERT AND COUNT VECTORIZER

EXAMPLE 1 ¶

+ Code + Markdown

26]:

```
count_100 = CountVectorizer(ngram_range=n_gram_range, stop_words=stop_words).fit([medical_df.iloc[20]['tra
candidates_100 = count_100.get_feature_names_out()
doc_embedding_100 = model_sent.encode([medical_df.iloc[20]['transcription']])
candidate_embeddings_100 = model_sent.encode(candidates_100)
distances_100 = cosine_similarity(doc_embedding_100, candidate_embeddings_100)
keywords = [candidates_100[index] for index in distances_100.argsort()[0][-20:]]

print(keywords)
print(medical_df.iloc[20]['keywords'])
```

```
Batches: 100% 1/1 [00:00<00:00, 47.94it/s]

Batches: 100% 5/5 [00:00<00:00, 104.25it/s]

['final', 'wound', 'ounces', '30', 'med', 'hour', 'apnea', 'sleep', 'tomorrow', 'hospital', 'cardiopulmonary', 'laparoscopic',
'meds', 'old', 'bariatric', 'postoperative', 'hypertension', 'surgery', 'gastric', 'obesity']
bariatrics, medifast, laparoscopic roux-en-y gastric bypass, roux-en-y, bariatric clear liquids, gastric bypass, laparoscopic,
gastric, bariatric, bypass,
```

EXAMPLE 2

+ Code + Markdown

```
count_100 = CountVectorizer(ngram_range=n_gram_range, stop_words=stop_words).fit([medical_df.iloc[300]['tr
candidates_100 = count_100.get_feature_names_out()
doc_embedding_100 = model_sent.encode([medical_df.iloc[300]['transcription']])
candidate_embeddings_100 = model_sent.encode(candidates_100)
distances_100 = cosine_similarity(doc_embedding_100, candidate_embeddings_100)
keywords = [candidates_100[index] for index in distances_100.argsort()[0][-20:]]

print(keywords)
print(medical_df.iloc[20]['keywords'])
```

```
Batches: 100% 1/1 [00:00<00:00, 42.94it/s]

Batches: 100% 8/8 [00:00<00:00, 111.81it/s]

['bleeding', 'colotomy', 'endoscopically', 'postoperative', 'diaphragm', 'peritoneal', 'catheter', 'endotracheal', 'gastrotom
y', 'surgical', 'gastrostomy', 'endoscopic', 'biopsy', 'lysis', 'anesthesia', 'endoscope', 'hemostasis', 'chemotherapy', 'endo
scopy', 'carcinoma']
bariatrics, medifast, laparoscopic roux-en-y gastric bypass, roux-en-y, bariatric clear liquids, gastric bypass, laparoscopic,
gastric, bariatric, bypass,
```

ROUGE SCORE COMPARISON

KEYBERT ¶

+ Code + Markdown

```
]: count_100 = CountVectorizer(ngram_range=n_gram_range, stop_words=stop_words).fit([medical_df.iloc[10]['tra
   candidates_100 = count_100.get_feature_names_out()
   doc_embedding_100 = model_sent.encode([medical_df.iloc[10]['transcription']])
   candidate_embeddings_100 = model_sent.encode(candidates_100)
   distances_100 = cosine_similarity(doc_embedding_100, candidate_embeddings_100)
   keywords = [candidates_100[index] for index in distances_100.argsort()[0][-50:]]



   cal(medical_df.iloc[10]['keywords'],' '.join(keywords))
```

Batches: 100% 1/1 [00:00<00:00, 39.18it/s]

Batches: 100% 3/3 [00:00<00:00, 79.84it/s]

```
0.0008329157525953993
```

BART

```
cal(results['keywords'].iloc[1],' '.join(results['Result'].iloc[1]))
```

```
0.0026025713943907936
```