Part 0

Briefly explain the method you implemented and give an example in the report.

```python
def preprocessing_function(text: str) -> str:
    preprocessed_text = remove_stopwords(text)
    # TO-DO 0: Other preprocessing function attemption
    # Begin your code
    preprocessed_text = preprocessed_text.lower()
    preprocessed_text = preprocessed_text.replace('<br / >', ' ')
    preprocessed_text = ''.join([char for char in preprocessed_text if char.
        isalpha() or char.isspace()])
    words = nltk.word_tokenize(preprocessed_text)

    # Choose SnowballStemmer after comparison
    stemmed_words = [SnowballStemmer(language='english').stem(word) for word in
        words]
    preprocessed_text = ' '.join(stemmed_words)
    # End your code
    return preprocessed_text
```

Stemming:

```
Original:
This has to be the worst piece of garbage I've seen in a while.<br /><br />Heath
    Ledger is a heartthrob? He looked deformed. I wish I'd known that he and Naomi
     Watts are an item in real life because I spent 2 of the longest hours of my
    life wondering what she saw in him. <br /><br />Orlando Bloom is a heartthrob?
     With the scraggly beard and deer-in-the-headlights look about him, I can't
    say I agree.<br /><br />Rachel Griffiths was her usual fabulous self, but
    Geoffrey Rush looked as if he couldn't wait to get off the set. <br /><br />
Remove stopwords/symbols:
worst piece garbage seen while heath ledger heartthrob looked deformed wish known
    naomi watts item real life spent longest hours life wondering saw him orlando
    bloom heartthrob scraggly beard deerintheheadlights look say agree rachel
    griffiths usual fabulous self geoffrey rush looked wait get set
PorterStemmer:
worst piec garbag seen while heath ledger heartthrob look deform wish known naomi
    watt item real life spent longest hour life wonder saw him orlando bloom
    heartthrob scraggli beard deerintheheadlight look say agre rachel griffith
    usual fabul self geoffrey rush look wait get set
SnowballStemmer:
worst piec garbag seen while heath ledger heartthrob look deform wish known naomi
    watt item real life spent longest hour life wonder saw him orlando bloom
    heartthrob scragg beard deerintheheadlight look say agre rachel griffith usual
     fabul self geoffrey rush look wait get set
LancasterStemmer:
worst piec garb seen whil hea ledg heartthrob look deform wish known naom wat item
     real lif spent longest hour lif wond saw him orlando bloom heartthrob
    scraggly beard deerintheheadlight look say agr rachel griffith us fab self
    geoffrey rush look wait get set
```

1. Provided preprocessing method:
   (a) Removing stopwords

   This method filters out stopwords from the text. Stopwords are commonly used words that are often removed in the preprocessing step because they occur frequently but usually don't carry significant meaning for understanding the main content of the text.
   (b) Tokenization
      i. The process of splitting text into smaller parts, typically words or sentences, makes it easier for algorithms to analyze and process.
      ii. ToktokTokenizer is effective at handling punctuation and other special characters, making it a robust choice for tokenizing raw text data.
2. Other preprocessing method:
   (a) Remove redundant symbols

   The code converts text to lowercase, eliminates HTML line break tags, and filters out non-alphabetic characters, preparing it for a more effective stemming process.
   (b) Stemming
      i. Stemming is the process of reducing words to their word stem or root form. This helps in reducing the complexity of the text and consolidates different forms of the same word.
      ii. Comparison:
         A. PorterStemmer

         An older and relatively gentle stemming algorithm focusing on removing common morphological endings from words in English. It's known for its simplicity and speed but can sometimes be less aggressive in stemming.
         B. SnowballStemmer

         An extension of PorterStemmer, it's more comprehensive and supports multiple languages. In English, it's similar to Porter but slightly more aggressive and accurate, handling a wider range of irregular cases.
         C. LancasterStemmer

         A very aggressive stemming algorithm that can significantly alter words, sometimes leading to stems that don't resemble the original word closely. It's fast but can produce more over-stemming errors compared to Porter and Snowball.

Part 1

1. Briefly explain the concept of perplexity in report and discuss how it will be influenced.

   Perplexity is a measurement used in natural language processing to evaluate language models. Lower perplexity indicates a model is better at making predictions, as it's less surprised by the new data. Mathematically, for a model M, perplexity (PP) of a sequence of words $W = w_1,\ w_2, ...,\ w_N$ is given by

   $$PP(W) = P(w_1,\ w_2, ...,\ w_N)^{-\frac{1}{N}}$$

   where $P(w_1,\ w_2, ...,\ w_N)$ is the probability of the word sequence according to the model, and N is the length of the sequence.

   Several factors influence the perplexity of a language model, such as model complexity, training data size and quality, vocabulary size, etc.

2. Screenshot the outputs and tell your observations about the differences in the perplexity caused by the preprocessing methods.

   (a) Without preprocess

   ```
   Progress: 100%|                              | 10000/10000 [03:25<00:00, 48.75it/s]
   Perplexity of ngram: 116.26046015880357
   Converting train data to embeddings: 100%| | 40000/40000 [05:30<00:00, 120.90it/s]
   Converting test data to embeddings: 100%|  | 10000/10000 [01:23<00:00, 120.31it/s]
   F1 score: 0.7051, Precision: 0.7082, Recall: 0.7059
   ```

   (b) With remove stopwords

   ```
   Preprocessing train data: 100%|          | 40000/40000 [00:29<00:00, 1361.00it/s]
   Preprocessing test data: 100%|           | 10000/10000 [00:07<00:00, 1296.82it/s]
   Progress: 100%|                            | 10000/10000 [01:41<00:00, 98.65it/s]
   Perplexity of ngram: 195.43245350997685
   Converting train data to embeddings: 100%| | 40000/40000 [03:48<00:00, 175.32it/s]
   Converting test data to embeddings: 100%|  | 10000/10000 [00:56<00:00, 176.74it/s]
   F1 score: 0.6745, Precision: 0.6902, Recall: 0.6792
   ```

   (c) With your method

   ```
   Preprocessing train data: 100%|           | 40000/40000 [01:52<00:00, 354.58it/s]
   Preprocessing test data: 100%|            | 10000/10000 [00:24<00:00, 404.87it/s]
   Progress: 100%|                           | 10000/10000 [00:21<00:00, 471.84it/s]
   Perplexity of ngram: 309.5762439395389
   Converting train data to embeddings: 100%| | 40000/40000 [02:52<00:00, 231.58it/s]
   Converting test data to embeddings: 100%|  | 10000/10000 [00:42<00:00, 233.97it/s]
   F1 score: 0.7095, Precision: 0.7187, Recall: 0.7118
   ```

   Comparison:

Remove stopwords	Stemming	Perplexity	F1 score	Precision	Recall
		116.2605	0.7051	0.7082	0.7059
✓		195.4325	0.6745	0.6902	0.6792
✓	✓	309.5762	0.7095	0.7187	0.7118

   Summary:

   While removing stopwords increases complexity, it also enhances the effectiveness of the method in terms of F1 score, precision, and recall. There's no straightforward correlation between perplexity and F1 score, indicating that higher complexity does not necessarily mean lower accuracy.

Part 2

1. Briefly explain the two pre-training steps in BERT.
   (a) Masked language model (MLM)

   In this step, BERT randomly masks some percentage of the input tokens and then predicts these masked words based solely on their context. This forces BERT to develop a deep understanding of the sentence structure and context.

   (b) Next sentence prediction (NSP)

   BERT also learns to understand the relationship between sentences. During training, it is given pairs of sentences and has to predict whether the second sentence in the pair is the actual next sentence in the original document.
2. Briefly explain four different BERT application scenarios.
   (a) Sentiment analysis

   BERT can determine whether a piece of text has a positive, negative, or neutral sentiment. This is valuable for understanding customer feedback or public opinion on social media.

   (b) Question answering

   BERT is effective in question answering systems where it can find and extract answers from a text corpus given a specific question, a useful feature for chatbots and virtual assistants.

   (c) Text Summarization

   BERT can be used to generate concise summaries of large text documents while retaining the key information, which is beneficial for quickly grasping the essence of news articles, research papers, or long reports.

   (d) Named Entity Recognition (NER)

   BERT can identify and classify specific entities in text, which is crucial for information extraction from documents, enhancing search algorithms, and organizing content.
3. Discuss the difference between BERT and distilBERT?
   (a) Model size and complexity

   BERT is a larger model with more parameters, resulting in greater complexity. In contrast, DistilBERT is a streamlined version with about 40% fewer parameters, retaining much of BERT's performance with reduced complexity.

   (b) Training and Inference Speed

   BERT requires more computational power and time for both training and inference due to its size. DistilBERT, being smaller, offers faster training and inference, making it more efficient and cost-effective.

   (c) Performance

   BERT typically delivers top-tier performance across various NLP tasks. DistilBERT, while slightly less accurate due to its reduced size, still maintains a high level of effectiveness in most applications.

   (d) Use Cases

   BERT is ideal for resource-rich environments where maximum performance is crucial. DistilBERT is better suited for scenarios requiring efficiency and speed, especially in resource-constrained settings.

4. Screenshot the required test F1-score.

```
# BERT Version 1 (F1: 0.9355, non-preprocessed)
tokenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 107kB/s]
config.json: 100% 483/483 [00:00<00:00, 2.79MB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 479kB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 1.87MB/s]
model.safetensors: 100% 268M/268M [00:02<00:00, 112MB/s]
Epoch 1/1: 100% 5000/5000 [29:19<00:00,  2.84batch/s, loss=0.062]
Evaluating: 100% 10000/10000 [01:48<00:00, 92.04batch/s]
Epoch: 1, Loss: 0.2301, Precision: 0.9355, Recall: 0.9355, F1: 0.9355
```

5. (BONUS 5%) Explain the relation of the Transformer and BERT and the core of the Transformer.
   (a) BERT's relation to the Transformer

   BERT is an advanced model that adapts the Transformer's technique, which utilizes its encoder architecture for bidirectional context understanding in texts and is adaptable for various NLP tasks through pre-training and fine-tuning.

   (b) The core of the Transformer

   The Transformer is a model primarily based on self-attention mechanisms, which enables efficient parallel processing and better handling of sequence data.

Part 3

1. Briefly explain the difference between vanilla RNN and LSTM.
   (a) Vanilla RNN

   Vanilla RNNs are basic recurrent networks with a straightforward architecture, excelling in processing sequences with short-term dependencies but facing challenges with longer sequences due to the vanishing gradient problem.

   (b) LSTM

   LSTM is an advanced type of recurrent neural network with a specialized structure that includes input, output, and forget gates, designed to effectively capture and retain long-term dependencies in sequential data.
2. Please explain the meaning of each dimension of the input and output for each layer in the model.
   (a) Embedding layer (self.embedding)
      i. nn.Embedding()
         - Input Dimension: [batch_size, sequence_length]
         - Output Dimension: [batch_size, sequence_length, embedding_dim]

   (b) LSTM layer (self.encoder)
      i. nn.LSTM()
         - Input Dimension: [sequence_length, batch_size, input_size]
         - Output Dimension: Outputs a tuple (output, (h_n, c_n)):
           - output: [sequence_length, batch_size, hidden_size]. If the LSTM is bidirectional, hidden_size will be double the defined hidden size.
           - h_n (hidden state): [D * num_layers, batch_size, hidden_size] where D=2 for bidirectional LSTMs and D=1 for unidirectional LSTMs.
           - c_n (cell state): Same dimension as h_n.

   (c) Linear layer within classifier (self.classifier)
      i. nn.Dropout()
         - Input and Output Dimension: The dimensions for both input and output are identical. The dropout layer randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.
      ii. nn.Linear()
         - Input: [batch_size, additional_dimension, input_size]
         - Output: [batch_size, additional_dimension, output_size]
      iii. nn.Sigmoid()
         - Input and Output Dimension: The dimensions for both input and output are identical. The Sigmoid function applies element-wise and does not alter the shape of the input tensor.

3. Screenshot the required test F1-score.

```
# RNN (F1: 0.8561, preprocessed)
Preprocessing train data: 100% 40000/40000 [00:37<00:00, 1068.48it/s]
Preprocessing test data: 100% 10000/10000 [00:09<00:00, 1050.59it/s]
Epoch 1/10: 100% 5000/5000 [02:09<00:00, 38.63batch/s, loss=0.657]
Evaluating: 100% 10000/10000 [00:11<00:00, 851.08batch/s]
Epoch: 1, Loss: 0.6896, Precision: 0.7035, Recall: 0.7035, F1: 0.7035
Epoch 2/10: 100% 5000/5000 [02:08<00:00, 38.87batch/s, loss=0.626]
Evaluating: 100% 10000/10000 [00:11<00:00, 900.82batch/s]
Epoch: 2, Loss: 0.5438, Precision: 0.8116, Recall: 0.8098, F1: 0.8095
Epoch 3/10: 100% 5000/5000 [02:08<00:00, 38.90batch/s, loss=0.535]
Evaluating: 100% 10000/10000 [00:11<00:00, 889.44batch/s]
Epoch: 3, Loss: 0.4675, Precision: 0.8301, Recall: 0.8291, F1: 0.8290
Epoch 4/10: 100% 5000/5000 [02:08<00:00, 38.94batch/s, loss=0.314]
Evaluating: 100% 10000/10000 [00:10<00:00, 934.80batch/s]
Epoch: 4, Loss: 0.4392, Precision: 0.8352, Recall: 0.8352, F1: 0.8352
Epoch 5/10: 100% 5000/5000 [02:08<00:00, 38.94batch/s, loss=0.314]
Evaluating: 100% 10000/10000 [00:11<00:00, 880.51batch/s]
Epoch: 5, Loss: 0.4219, Precision: 0.8431, Recall: 0.8417, F1: 0.8415
Epoch 6/10: 100% 5000/5000 [02:09<00:00, 38.73batch/s, loss=0.437]
Evaluating: 100% 10000/10000 [00:11<00:00, 890.18batch/s]
Epoch: 6, Loss: 0.4067, Precision: 0.8499, Recall: 0.8496, F1: 0.8496
Epoch 7/10: 100% 5000/5000 [02:08<00:00, 38.88batch/s, loss=0.438]
Evaluating: 100% 10000/10000 [00:11<00:00, 902.70batch/s]
Epoch: 7, Loss: 0.3933, Precision: 0.8523, Recall: 0.8511, F1: 0.8510
Epoch 8/10: 100% 5000/5000 [02:09<00:00, 38.67batch/s, loss=0.313]
Evaluating: 100% 10000/10000 [00:11<00:00, 899.02batch/s]
Epoch: 8, Loss: 0.3841, Precision: 0.8565, Recall: 0.8561, F1: 0.8561
Epoch 9/10: 100% 5000/5000 [02:09<00:00, 38.71batch/s, loss=0.563]
Evaluating: 100% 10000/10000 [00:10<00:00, 912.11batch/s]
Epoch: 9, Loss: 0.3808, Precision: 0.8562, Recall: 0.8517, F1: 0.8512
Epoch 10/10: 100% 5000/5000 [02:08<00:00, 38.87batch/s, loss=0.313]
Evaluating: 100% 10000/10000 [00:11<00:00, 888.23batch/s]
Epoch: 10, Loss: 0.3760, Precision: 0.8552, Recall: 0.8488, F1: 0.8481
```

Discussion

1. Discuss the innovation of the NLP field and your thoughts of why the technique is evolving from n-gram -> LSTM -> BERT.

   The progression of NLP techniques from n-grams through LSTMs to BERT mirrors the field's ongoing quest for deeper and more efficient language understanding. Early n-gram models offered a basic, rule-based approach but struggled with contextual and long-term dependencies.

   The advent of RNNs and LSTMs marked a shift to more fluid, context-aware processing, better capturing sequence dependencies in language. However, limitations in handling very long-term dependencies and bidirectional context persisted.

   The introduction of the Transformer architecture and BERT revolutionized NLP by enabling models to process text non-sequentially, efficiently manage long-range dependencies, and understand deep bidirectional context. This evolution reflects both technological advancements and an enhanced understanding of linguistic complexities.
2. Describe problems you meet and how you solve them.
   (a) The preprocessing method is too slow due to the repeated reading of the stop_word_list for each setence.

   Solution: Move the stop_word_list outside of the remove_stopwords function for efficiency.

```python
stop_word_list = stopwords.words('english')

def remove_stopwords(text: str) -> str:
    '''
    E.g.,
        text: 'Here is a dog.'
        preprocessed_text: 'Here dog.'
    '''
    tokenizer = ToktokTokenizer()
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    filtered_tokens = [token for token in tokens if token.lower() not in
        stop_word_list]
    preprocessed_text = ' '.join(filtered_tokens)

    return preprocessed_text
```

(b) Test F1-score of a BERT model can't surpass 0.94.
Solution: Modify the architecture of the classifier, but the issue remains unresolved.

```
self.classifier = nn.Sequential(
    # Version 1 (F1: 0.9355)
    nn.Linear(self.pretrained_model.config.hidden_size, 512),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(512, num_labels)

    # Version 2 (F1: 0.9337)
    nn.Linear(self.pretrained_model.config.hidden_size, 1024),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, num_labels)

    # Version 3 (F1: 0.9344)
    nn.Linear(self.pretrained_model.config.hidden_size, 768),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(768, 384),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(384, num_labels)

    # Version 4 (F1: 0.9325)
    nn.Linear(self.pretrained_model.config.hidden_size, 1024),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(512, num_labels)

    # Version 5 (F1: 0.9227)
    nn.Linear(self.pretrained_model.config.hidden_size, 768),
    nn.BatchNorm1d(768),
    nn.LeakyReLU(),
    nn.Dropout(0.2),
    nn.Linear(768, 384),
    nn.BatchNorm1d(384),
    nn.LeakyReLU(),
    nn.Dropout(0.2),
    nn.Linear(384, num_labels)
)
```

In-context learning (BONUS 5%)

Try to use the LLMs (such as ChatGPT, Gemini, etc) with in-context learning to complete the task above. Please compare the results with different prompts and write down your findings in the report.

In this part, we focus on Part 0 of the homework assignment, which involves implementing a different text preprocessing method for sentiment classification.

1. Few-shot Learning Approach
    - Few-shot learning involves providing the LLM with a small number of examples that demonstrate the task at hand.
    - `https://chat.openai.com/share/631198c8-ac23-48b3-886b-fb26bd489cb1`
2. Chain of Thought Approach
    - The Chain of Thought approach prompts the model to articulate its reasoning process step-by-step to arrive at a conclusion.
    - `https://chat.openai.com/share/ee39e296-a607-4339-899d-cd5645ab4208`
3. Socratic Method Approach
    - The Socratic Method involves guiding the model to a conclusion through a series of questions and answers.
    - `https://chat.openai.com/share/5d005c86-a46b-48e8-ad10-d13831c6e5d2`

Summary:

- Few-shot Learning is best for straightforward, example-driven tasks.
- Chain of Thought is useful for complex problems requiring detailed reasoning.
- Socratic Method excels in exploratory learning and understanding underlying concepts.

Appendix

1. Reference
    (a) OpenAI. (2024). ChatGPT (4) [Large language model]. `https://chat.openai.com`