Neural Networks and NLP Assignment 1

Salva Karimisahari 240753803

Part A: Neural Machine Translation

(Implemented in Lab1_2_Neural_Machine_Translation_240753803.ipynb)

1. Task 1: Implementing the encoder:
   To implement the encoder embedding_source and embedding_target were defined as two nn.Embedding layers that convert token indices to dense vector representations. These embeddings are trainable and help the model understand the relationship between words. Setting padding_idx to source_dict.PAD and target_dict.PAD ensures that the paddings added to the embeddings won't affect the training process.
   An unidirectional LSTM layer was added for the encoder to process source sentences (source_words) into hidden states.

```
       """
       Task 1: Implementing the encoder 1/2

       Begin
       """

       # embeddings for source and target with padding_idx specified
          self.embedding_source = nn.Embedding(self.vocab_source_size,
self.embedding_size, padding_idx=source_dict.PAD)
          self.embedding_target = nn.Embedding(self.vocab_target_size,
self.embedding_size, padding_idx=target_dict.PAD)

       # encoder lstm layer
                                             self.encoder_lstm      =
nn.LSTM(input_size=self.embedding_size,hidden_size=self.hidden_size,
num_layers=1,batch_first=True,dropout=self.hidden_dropout_rate,
bidirectional=False)
       """
       End Task 1 1/2
       """
```

For the second part, first we do the embedding lookup step and end up with embeddings of shape [batch_size, max_source_len, embedding_size]. Then we use an LSTM to generate output sequences and use teacher forcing where the correct previous word is fed into the decoder. Finally, the decoder projects its outputs to the target vocabulary size.
Also if attention is used, it will be applied before the final prediction (added value errors in case it was None).

```python
    def forward(self, source_words, target_words):
        """
        Forward pass for training:
          1) Encode the source sentences using the encoder LSTM.
          2) Use the final encoder state to initialize the decoder's hidden state.
          3) Feed all target words into the decoder LSTM in one go (teacher forcing).
          4) (Optional) apply the attention layer between the decoder outputs and the encoder outputs.
          5) Project the decoder outputs to vocabulary logits with self.proj.
        """

        """
        Task 1: Implementing the encoder 2/2

        Begin
        """

        #embedding lookup
        source_words_embeddings = self.embedding_source(source_words)  # [batch_size, max_source_len, embedding_size]
        target_words_embeddings = self.embedding_target(target_words)  # [batch_size, max_target_len, embedding_size]



        #encoding source words
        encoder_outputs, (enc_h, enc_c) = self.encoder_lstm(source_words_embeddings)

        #teacher forcing
        decoder_outputs, _ = self.decoder_lstm(target_words_embeddings, (enc_h, enc_c))

        # if attention is used
        if self.use_attention:
            print("applying attention")
```

```
                                                decoder_outputs    =
self.decoder_attention(encoder_outputs, decoder_outputs)
            if decoder_outputs is None:
                raise ValueError("decoder_outputs is none after
attention!")


        """
        End Task 1 2/2
        """

        # 5) Projection
          decoder_outputs = self.decoder_dense(decoder_outputs)   #
[batch, max_tgt_len, vocab_target_size]
        return decoder_outputs
```

2. Task 2: Implementing the decoder and inference loop
   With the embedding lookup, the current target word generated at the previous step gets converted into its embedding representation. The LSTM decoder processes the embedding to update the decoder's hidden state. If attention is enabled, contextual attention scores get computed and decoder outputs get modified. Finally, the decoder output is converted into vocabulary logits for predicting the next word.

```
          def   decode_step(self,   target_words,   decoder_states,
encoder_outputs):
        """
        A single step of decoder inference:
          - Embedding for the current token
          - One-step LSTM forward
          - (Optional) attention over encoder outputs
          - Project to vocab
        Inputs:
          tgt_input: shape [batch_size, 1]
            decoder_states: (dec_h, dec_c) each is [1, batch_size,
hidden_size]
          encoder_outputs: [batch_size, max_src_len, hidden_size]
        Returns:
          logits for the next token, and the new decoder states
        """



        """

        Begin
        """
```

```
                #embedding lookup for the current target token [batch_size, 1, embedding_size]
                                        target_words_embeddings = self.embedding_target(target_words)

        # LSTM step using previous decoder states
                            decoder_outputs, (dec_h, dec_c) = self.decoder_lstm(target_words_embeddings, decoder_states)

        #if attention is used
        if self.use_attention:
                                                  decoder_outputs = self.decoder_attention(encoder_outputs, decoder_outputs)
            if decoder_outputs is None:
                    raise ValueError("decoder_outputs is None after attention!")

                   #projecting to vocabulary space [batch, 1, vocab_target_size]
        decoder_outputs = self.decoder_dense(decoder_outputs)

        """
        End Task 2
        """

        return decoder_outputs, (dec_h, dec_c)
```

The Model BLEU score ended up being as follows:

```
Model BLEU score: 4.46
```

And some samples of predicted translation and reference translation can be seen as follows:

```
predicted Translation: the <unk> <unk> <unk> <unk> <unk> <unk> <unk> , and it 's <unk> .
reference Translation: the second quote is from the head of the u.k. financial services <unk> .

predicted Translation: it 's <unk> .
reference Translation: it gets worse .

predicted Translation: what 's the <unk> of the <unk> that we can do is <unk> ?
reference Translation: what 's happening here ? how can this be possible ?

predicted Translation: well , it 's not a <unk> .
```

```
reference Translation: unfortunately , the answer is yes .

predicted Translation: but <unk> , <unk> <unk> <unk> <unk> <unk>
<unk> <unk> <unk> <unk> <unk> <unk> <unk> .
reference Translation: but there 's an <unk> solution which is
coming from what is known as the science of <unk> .
```

3. Task 3: Adding attention
   First, it is checked that the encoder output isn't empty and then decoder outputs need to be reshaped to match the encoder's shape. Attention scores are computed via performing a dot product between encoder and decoder outputs. Softmax is applied to obtain attention weights and then the context vector is computed as a weighted sum of encoder outputs and the context vector concatenated with decoded outputs is returned as a result.

```
class AttentionLayer(nn.Module):
    """
    Custom layer implementing Luong attention.
    """
    def __init__(self):
        super(AttentionLayer, self).__init__()

    def forward(self, encoder_outputs, decoder_outputs):
        """
        encoder_outputs : [batch_size, max_source_length, hidden_size]
        decoder_outputs : [batch_size, max_target_length, hidden_size]
        """

        if encoder_outputs is None or decoder_outputs is None:
                raise ValueError("encoder_outputs or decoder_outputs is
None.")

        batch_size, max_source_len, hidden_size = encoder_outputs.shape
        _, max_target_len, _ = decoder_outputs.shape

        #transposing decoder outputs to match encoder shape
             decoder_outputs_t = decoder_outputs.permute(0, 2, 1)    #
[batch_size, hidden_size, max_target_length]

        #computing attention scores with dot product
          luong_score = torch.bmm(encoder_outputs, decoder_outputs_t)   #
[batch_size, max_source_length, max_target_length]

        #applying a softmax to obtain attention weights
          attention_weights = F.softmax(luong_score, dim=1)   # Normalize
over source sequence
```

```
        #computing thecontext vector as a weighted sum of encoder
outputs
        attention_weights = attention_weights.permute(0, 2,
1).unsqueeze(-1)  # [batch, max_target_length, max_source_length, 1]
        encoder_outputs_exp = encoder_outputs.unsqueeze(1)  # [batch, 1,
max_source_length, hidden_size]

        encoder_vector = torch.sum(attention_weights *
encoder_outputs_exp, dim=2)  # [batch, max_target_length, hidden_size]

        #ensuring decoder_outputs and encoder_vector have the same
length
        min_len = min(decoder_outputs.shape[1], encoder_vector.shape[1])
        decoder_outputs = decoder_outputs[:, :min_len, :]
        encoder_vector = encoder_vector[:, :min_len, :]

        #concating context vector with decoder outputs
        new_decoder_outputs = torch.cat([decoder_outputs,
encoder_vector], dim=-1)

        print("attention decoder outputs shape:",
new_decoder_outputs.shape)
        return new_decoder_outputs
```

The model’s BLEU score has risen to 14 as adding attention allows the model to focus on different words dynamically.

```
Model BLEU score: 14.00
```

Some sample predicted and reference translation can be seen below:

```
predicted Translation: in the first few of the <unk> first <unk>
comes from the first business comes from the way .
reference Translation: the second quote is from the head of the
u.k. financial services <unk> .

predicted Translation: so , it 's more <unk> .
reference Translation: it gets worse .

predicted Translation: what 's happening in here ? why are you ?
"
reference Translation: what 's happening here ? how can this be
possible ?

predicted Translation: unfortunately , unfortunately , the answer
is yes .
reference Translation: unfortunately , the answer is yes .
```

```
predicted Translation: but in fact , there 's a very interesting solution from the age of doing science .
reference Translation: but there 's an <unk> solution which is coming from what is known as the science of <unk> .
```

Part B: Using the Pre-trained BERT models

(Implemented in Lab3_ABSA_with_BERT_240753803.ipynb)

1. Task 1: Data pre-processing

Using a preprocessing function, sentences and aspect words are combined with a [SEP] token. So for example, train[0] which originally is ['the decor is not special at all but their food and amazing prices make up for it.', 'decor', 'negative', '4', '9'] will turn into “the decor is not special at all but their food and amazing prices make up for it. [SEP] decor”. Afterwards, this sentence will be tokenized using the DistilBERT tokenizer that was previously defined and the function will return the input ids and attention masks (with max_length padding). In conclusion, x_train_int[0] and x_train_mask[0] will be as follows:

```
[   101  1996 25545  2003  2025  2569  2012  2035  2021  2037  2833  1998
  6429  7597  2191  2039  2005  2009  1012   102 25545   102     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

The 101 indicating the start and 102 indicating the [SEP] that we added before the aspect and in the ending. The implementation of the preprocess function and its usage can be seen below:

```
"""
Task 1: Data pre-processing

Begin
"""

# Please write your code to combine sentences and aspect words into the following varibles

# x_train_int
# x_train_masks
# x_dev_int
# x_dev_masks
```

```
# x_test_int
# x_test_masks
# Tips:
# 1) Use the special token [SEP] to concatenate sentences and aspect
words
# 2) Make sure they are paded/truncated to a max length
# Function to tokenize review + aspect
def preprocess(dataset, tokenizer, pad_length=128):
   input_ids, attention_masks = [], []

   for row in dataset:
       review = row[0]
       aspect = row[1]

       #combine review and aspect using the [SEP] token
       combined_row = review + " [SEP] " + aspect
       #print(combined_row)
       #tokenizing using DistilBERT
                  tokenized_row = tokenizer.encode_plus(combined_row,
add_special_tokens=True,
                                                  max_length=pad_length,
padding='max_length',
                                                        truncation=True,
return_attention_mask=True)

       input_ids.append(tokenized_row['input_ids'])
       attention_masks.append(tokenized_row['attention_mask'])

   return np.array(input_ids), np.array(attention_masks)

#applying preprocessing and tokenization to train, dev, and test sets

x_train_int, x_train_masks = preprocess(train, tokenizer)
x_dev_int, x_dev_masks = preprocess(dev, tokenizer)
x_test_int, x_test_masks = preprocess(test, tokenizer)



"""
End Task 1
"""

# Don't forget the to use the np.array function to wrap the outputs
```

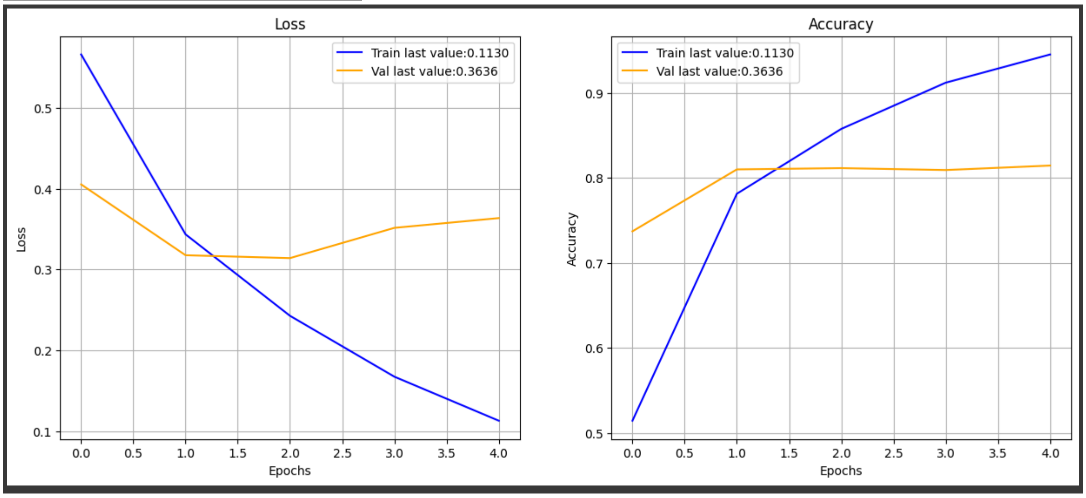
```
x_train_int_np = np.array(x_train_int)
x_train_masks_np = np.array(x_train_masks)
x_dev_int_np = np.array(x_dev_int)
x_dev_masks_np = np.array(x_dev_masks)
x_test_int_np = np.array(x_test_int)
x_test_masks_np = np.array(x_test_masks)


print(x_dev_int[0])
print(x_dev_masks[0],'\n')
print(x_dev_int_np[0])
print(x_dev_masks_np[0]) # sentence + aspect
```

The requested prints had these outputs:

```
[   101   2044   1037   3232   1997   8974   1010   1996 18726   1011   1011   1045
   2066   1996 27940   1013 24792   2621   4897   1998   1996 13675 11514   6508
  26852   1011   1011   2175   2091   2307   1012    102   8974    102      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[   101   2044   1037   3232   1997   8974   1010   1996 18726   1011   1011   1045
   2066   1996 27940   1013 24792   2621   4897   1998   1996 13675 11514   6508
  26852   1011   1011   2175   2091   2307   1012    102   8974    102      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

2. Task 2: Basic classifiers using BERT
    - Results for model 1:
      Running the example code resulted in the following loss and accuracy values:
      ```
      Epoch 1/5, Train Loss: 0.5660, Train Acc: 0.5145, Val Loss: 0.4053, Val Acc: 0.7372
      ```

```
Epoch 2/5, Train Loss: 0.3435, Train Acc: 0.7814, Val Loss: 0.3177, Val Acc: 0.8101
Epoch 3/5, Train Loss: 0.2428, Train Acc: 0.8577, Val Loss: 0.3141, Val Acc: 0.8116
Epoch 4/5, Train Loss: 0.1674, Train Acc: 0.9120, Val Loss: 0.3515, Val Acc: 0.8093
Epoch 5/5, Train Loss: 0.1130, Train Acc: 0.9452, Val Loss: 0.3636, Val Acc: 0.8146
```



Test seat results:

```
Test Loss: 0.3269, Test Acc: 0.8263
```

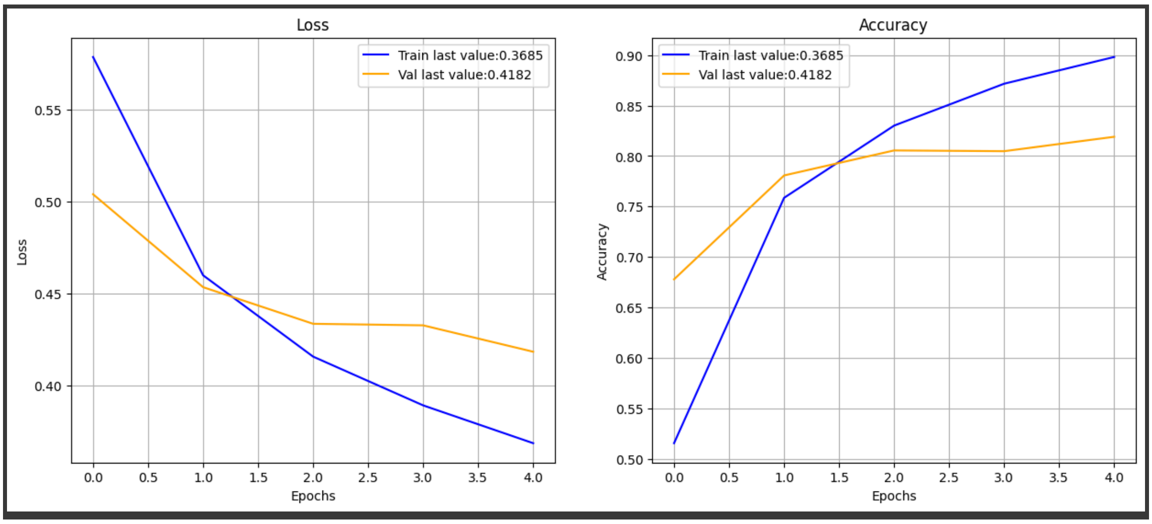
- Results for model 2:
  The code for model 2 is completed with loading the BERT embeddings, adding a pooling layer to average token embeddings (ignoring padding), setting hidden layer size to hdepth, defining the first hidden layer with 16 units,and an output layer for 3 classes.

```
lass BagOfWordsBERT(nn.Module):
   def __init__(self, hdepth=16):
       super(BagOfWordsBERT, self).__init__()

       """
       Task 2: Basic classifiers using BERT

       Begin
       """

                                     self.bert      =
DistilBertModel.from_pretrained('distilbert-base-uncased',
config=config)
       self.pooling = GlobalAveragePooling1DMasked()
       self.hidden_size = hdepth
       self.hidden = nn.Linear(768, self.hidden_size)
```

```
        self.output = nn.Linear(self.hidden_size, 3) #because
we have 3 classes

        """
        End Task 2
        """

    def forward(self, input_ids, attention_mask):
                bert_output  =  self.bert(input_ids=input_ids,
attention_mask=attention_mask)[0]
        pooled = self.pooling(bert_output, attention_mask)
        hidden = torch.sigmoid(self.hidden(pooled))
        output = self.output(hidden)
        return torch.softmax(output, dim=1)
```

The training and validation results were as follows:

```
Epoch 1/5, Train Loss: 0.5785, Train Acc: 0.5156, Val Loss:
0.5038, Val Acc: 0.6779
Epoch 2/5, Train Loss: 0.4598, Train Acc: 0.7586, Val Loss:
0.4533, Val Acc: 0.7808
Epoch 3/5, Train Loss: 0.4156, Train Acc: 0.8301, Val Loss:
0.4334, Val Acc: 0.8056
Epoch 4/5, Train Loss: 0.3890, Train Acc: 0.8716, Val Loss:
0.4326, Val Acc: 0.8048
Epoch 5/5, Train Loss: 0.3685, Train Acc: 0.8981, Val Loss:
0.4182, Val Acc: 0.8191
Epoch 1/5, Train Loss: 0.5501, Train Acc: 0.5769, Val Loss:
0.4762, Val Acc: 0.7440
Epoch 2/5, Train Loss: 0.4560, Train Acc: 0.7821, Val Loss:
0.4487, Val Acc: 0.7928
Epoch 3/5, Train Loss: 0.4204, Train Acc: 0.8470, Val Loss:
0.4358, Val Acc: 0.8138
Epoch 4/5, Train Loss: 0.3961, Train Acc: 0.8874, Val Loss:
0.4287, Val Acc: 0.8176
Epoch 5/5, Train Loss: 0.3839, Train Acc: 0.9009, Val Loss:
0.4254, Val Acc: 0.8191
```



The test result:

`Test Loss: 0.4083, Test Acc: 0.8301`

- Comparison

  In model 2, we use DistilBERT model instead of DistilBERTForSequenceClassification where the BERT parameters are not fine-tuned and instead of using the CLS, an average of word embeddings is passed to a layer with 16 neurons. We can see that model 1 and model 2 have quite similar performance because fine tuning BERT captures more details, but for a dataset like ABSA, a simple embedding and averaging approach can be effective. Looking at the loss plot, Model 1's validation loss starts slightly increasing after the second epoch, indicating overfitting but model 2's validation is continuously decreasing meaning it did not overfit. Also model 2's final training loss is higher than that of model one's so it also shows less overfitting. Also since the difference between model one's training and validation accuracy is a lot more, it indicates that model 1 memorized the training data because the model was too complex and didn't generalize well. It can be concluded that the simpler model 2 approach is less prone to overfitting.

	Model 1	Model 2
Final Train Loss	0.1130	0.3685
Final Train Accuracy	0.9452	0.8981
Final Validation Loss	0.3636	0.4182
Final Validation Accuracy	0.8146	0.8191
Test Loss	0.3269	0.4083
Test Accuracy	0.8263	0.8301

3. Task 3: Advanced classifier using BERT : Model 3 (using LSTM)

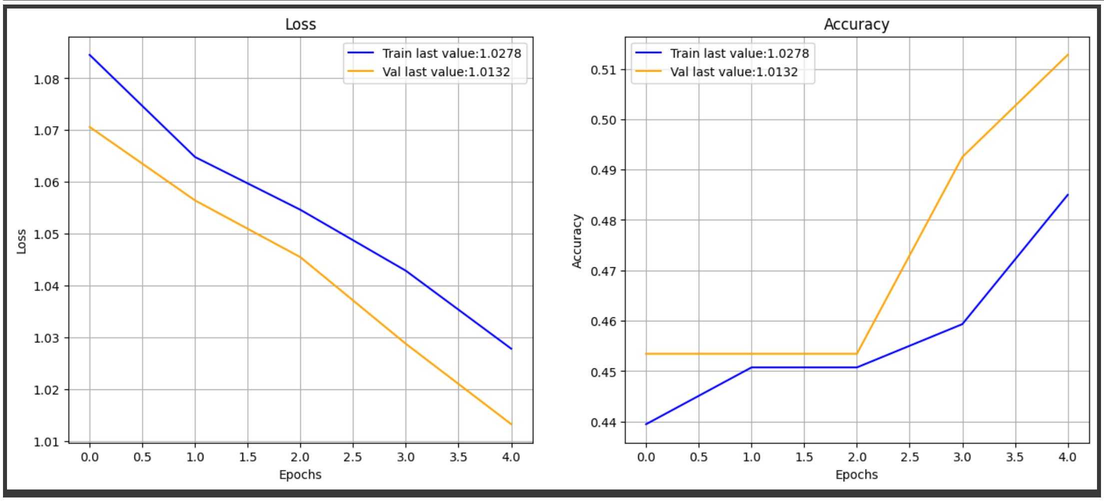
The global average pooling layer gets replaced with an LSTM layer. BERT parameters are still not being fine-tuned. The LSTM layer will capture sequential dependencies unlike the BoW approach. The code can be seen below:

```
class LSTM_BERT(nn.Module):
    """
    Task 3: Advanced classifier using BERT:

    Begin
    """

    def __init__(self, lstm_hidden_size=100):
        super(LSTM_BERT, self).__init__()
                                                        self.bert        =
DistilBertModel.from_pretrained('distilbert-base-uncased')
        #freezing BERT parameters so they dont get fine-tuned
        for param in self.bert.parameters():
            param.requires_grad = False
        #LSTM
        self.lstm_hidden_size = lstm_hidden_size
                             self.lstm    =    nn.LSTM(input_size=768,
hidden_size=self.lstm_hidden_size,batch_first=True,
bidirectional=False)
        #outout for 3 classes
        self.fc = nn.Linear(self.lstm_hidden_size, 3)


    def forward(self, input_ids, attention_mask):
        #extract BERT embeddings (last hidden states)
        with torch.no_grad():
                     bert_output  =  self.bert(input_ids=input_ids,
attention_mask=attention_mask)

        embeddings = bert_output.last_hidden_state

        # LSTM
        lstm_output, _ = self.lstm(embeddings)

        #classification from output
        lstm_final = lstm_output[:, -1, :]
        #return class probabilities
        return torch.softmax(self.fc(lstm_final), dim=1)
```

```
"""
End Task 3
"""
```

The following results were obtained on training and validation data:

```
Epoch 1/5, Train Loss: 1.0845, Train Acc: 0.4395, Val Loss: 1.0706, Val Acc: 0.4535
Epoch 2/5, Train Loss: 1.0648, Train Acc: 0.4507, Val Loss: 1.0564, Val Acc: 0.4535
Epoch 3/5, Train Loss: 1.0546, Train Acc: 0.4507, Val Loss: 1.0455, Val Acc: 0.4535
Epoch 4/5, Train Loss: 1.0428, Train Acc: 0.4593, Val Loss: 1.0287, Val Acc: 0.4925
Epoch 5/5, Train Loss: 1.0278, Train Acc: 0.4850, Val Loss: 1.0132, Val Acc: 0.5128
```



Test results:

```
Test Loss: 1.0156, Test Acc: 0.5105
```

- Comparison to model 2:

	Model 2	Model 3
Final Train Loss	0.3685	1.0278
Final Train Accuracy	0.8981	0.4850
Final Validation Loss	0.4182	1.0132
Final Validation Accuracy	0.8191	0.5128
Test Loss	0.4083	1.0156
Test Accuracy	0.8301	0.5105

In model 2, the validation loss stabilized early and remained around 0.4182, which suggests that it is learning well. However in model 3, The validation loss gradually decreased but remained high , meaning the model is not converging. The training accuracy also improves slightly in the last epoch but stays very low. In general model 2 outperforms model 3 probably because of the fact that BERT embeddings already encapsulate contextuality so LSTM might be a better approach for raw embeddings. Adding an LSTM just added to the parameters without improving the performance.

Part C: Natural Language Generation

(Implemented in Lab4_Natural_Language_Generation_240753803.ipynb)

1. Task 1 and 2: Build and evaluate greedy and beam search algorithms from scratch
    - Greedy algorithm:
      We compute the log probabilities using softmax and log, take the most likely token with argmax and the log probabilities of the next token and append the generated token to the input sequence until we reach the max length.
      The greedy algorithm we implemented generates this output:

```
Output:
----------------------------------------------------------------------------------------------------
The cat slept on the floor, and the cat was still asleep.

"I was just trying to get her to sleep," she said. "I was trying to get her to sleep, but she was still asleep."

The cat

 Its log-probability:
----------------------------------------------------------------------------------------------------
-62.58576202392578
```

      Huggingface greedy output:

```
The cat slept on the floor, and the cat was still asleep.

"I was just trying to get her to sleep," she said. "I was trying to get her to sleep, but she was still asleep."

The cat
```

      It can be seen that they both produced the same result.
    - Beam search algorithm:
      We compute the probability using softmax and log and instead of the argmax, we keep the top k (beam size) probabilities and we maintain the best sequences based on their cumulative log probabilities.
      Our implementation's output:

```
Output:
----------------------------------------------------------------------------------------------------
The cat slept on the floor of the house.

"It was like a nightmare," she said.
```

```
"It was like a nightmare. It was like a nightmare. It was like a nightmare. It was like a nightmare. It

 Its log-probability:
----------------------------------------------------------------------------------------------------
-46.066965823905775
```

Official implementation’s output:

```
Output:
----------------------------------------------------------------------------------------------------
The cat slept on the floor of the house.

"It was like a nightmare," she said.

"It was like a nightmare. It was like a nightmare. It was like a nightmare. It was like a nightmare. It
```

We can see that they produce the same output, however the output is problematic because of repetition.

- Comparison:
  The log-probability for greedy search is -62.59, whereas the log-probability for beam search is -46.07. The log probability is higher (less negative) for beam search, meaning it found a more probable sequence than the greedy search.
  Greedy search only makes the best decision at each step without looking at the overall structure whereas Bea msearch explores multiple high probability paths. That;s why Beam search results look more fluent and having the higher log probability means that the result obtained with beam search is more probable than the one obtained with greedy search. Greedy search repeats the sequences “The cat slept on the floor, and the cat was still asleep “ and “I was trying to get her to sleep, but she was still asleep". Beam search repeats the phrase “I was like a nightmare” multiple times. So, they both got stuck in repetition but in different ways, beam search is repeating a highly probable sentence and greedy search gets stuck in a loop of repeating similar words.
  The log probability difference suggests beam search still found a more likely sequence under the model, but both methods struggled with repetition.

2. Task 3: Build and evaluate the beam search algorithm with a repetition penalty
   - Official implementation’s output:

```
Output:
----------------------------------------------------------------------------------------------------
The cat slept on the floor of the house.

"I'm not going to tell you what happened," she said. "I don't want to talk about it. I'm just trying to make it clear that I didn't do anything
```

   - Our implementation’s output:

```
Output:
```

```
----------------------------------------------------------------------------------------------------
The cat slept on the floor of her room, and when she woke up to find that it had been eaten by a large black cat in the middle of her room, she immediately ran to help. She had no idea what was going on with the
Its log-probability:
----------------------------------------------------------------------------------------------------
-77.42537941224873
```

- Comparison between hypothesis generated in task 2 (Beam Search) and task 3 and Huggingface's no repetition:
  The three outputs obtained in task 2 and task 3 are very different.
  Huggingface's sentence avoids repetition and follows a more natural conversational flow. The structure is coherent, and there is no repetitive looping. However our implementation of no repetition prevents word for word repetition but doesn't generate coherent or meaningful sentences.
  The Huggingface implementation puts constraints on n-grams, preventing any bigram from appearing twice, resulting in more coherent sentences.
  Our approach penalizes the probabilities of the last 20 tokens, which is causing unusual generalizations.
  The hypothesis in Task 2 has a higher log probability (-46.07) than the one in task 3 (-77.42) meaning it is more likely to appear and it has been generated with more confidence. The sentence in task 2 was more meaningful but it was stuck repeating the phrase "It was like a nightmare" multiple times. The sentence in Task 3 has no repetition but has unusual parts like "eaten by a large black cat". The repetition penalty in task 3 forces the model to choose lower-probability words to avoid reusing the same n-grams so the model starts producing sentences that are less likely in general, resulting in a worse log probability. Huggingface's own implemented repetition penalty created the perfect balance of abiding repetition while generating probable sentences.

3. Task 4: Experiment and assess sampling results in terms of coherence and diversity
   a. Random sampling:

```
Random Sampling output:
----------------------------------------------------------------------------------------------------
The cat slept on the floor of her room, while she held it to her chest.

When Kiel went to check on him, a nurse found that Ejima had taken her cat.

Kiel told the nurse that she
```

   This approach chooses the next word without hard constraints so the tokens are very diverse but not very coherent or meaningful text.
   b. High Temperature output:

```
High temperature output:
```

```
------------------------------------------------------------
----------------------------------------
The cat slept on the walls. Two kids lay here — the boys
went here with two very quiet girls and one with one pretty
girl (for two months they hadn't got a boy since that day's
story): one was not a parent and three
```
This approach increases the randomness in choice of words leading to a higher diversity but lower coherency. The resulting sentence has no grammatical or contextual structure.

c. Low Temperature output

```
Low temperature output:
------------------------------------------------------------
----------------------------------------
The cat slept on the floor, and the cat was still asleep.

The cat was not the only one who had been affected.

The cat was also found to have been in a coma.

The cat had been found to
```
This approach is less random so generated words are more coherent but it keeps repeating the same phrases starting with “The cat…” so we have less diversity.

d. Top P output:

```
Top p output:
------------------------------------------------------------
----------------------------------------
The cat slept on the door to his basement as a friend found
it. That wasn't to say it didn't hurt, as the couple grew to
love each other; in fact, while they probably already knew
it had been broken, and that you
```
This method is a controlled way of increasing randomness as it has the constraint of an accumulated probability equal or higher than a predefined value. While having diversity, it has also created the most coherent and readable text among others.

Part D: Social Media Processing

(Implemented in Lab5_Social_Media_Processing_240753803.ipynb)

1. Task 1 and 2: Build and evaluate two regression models for humour rating prediction
   a. Task 1:
   The output for print(model):

```
BERTRegressionModel(
  (bert): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768,
padding_idx=0)
```

```
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): DistilBertSdpaAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
  )
  (global_avg_pool): GlobalAveragePooling1DMasked()
  (hidden_layer): Linear(in_features=768, out_features=16, bias=True)
  (activation): Sigmoid()
  (output_reg): Linear(in_features=16, out_features=1, bias=True)
)
```

Train and Val loss:

```
Epoch 1/5: 100%|██████████| 20/20 [00:43<00:00,  2.18s/it]
Train Loss: 0.1714, Val Loss: 0.0135
Epoch 2/5: 100%|██████████| 20/20 [00:46<00:00,  2.30s/it]
Train Loss: 0.0131, Val Loss: 0.0130
Epoch 3/5: 100%|██████████| 20/20 [00:48<00:00,  2.44s/it]
Train Loss: 0.0120, Val Loss: 0.0121
Epoch 4/5: 100%|██████████| 20/20 [00:48<00:00,  2.40s/it]
Train Loss: 0.0114, Val Loss: 0.0118
```

```
Epoch 5/5: 100%|          | 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0112, Val Loss: 0.0115
```

Test results:

```
Test loss: 0.0116
Test Mean Squared Error: 0.0119
```

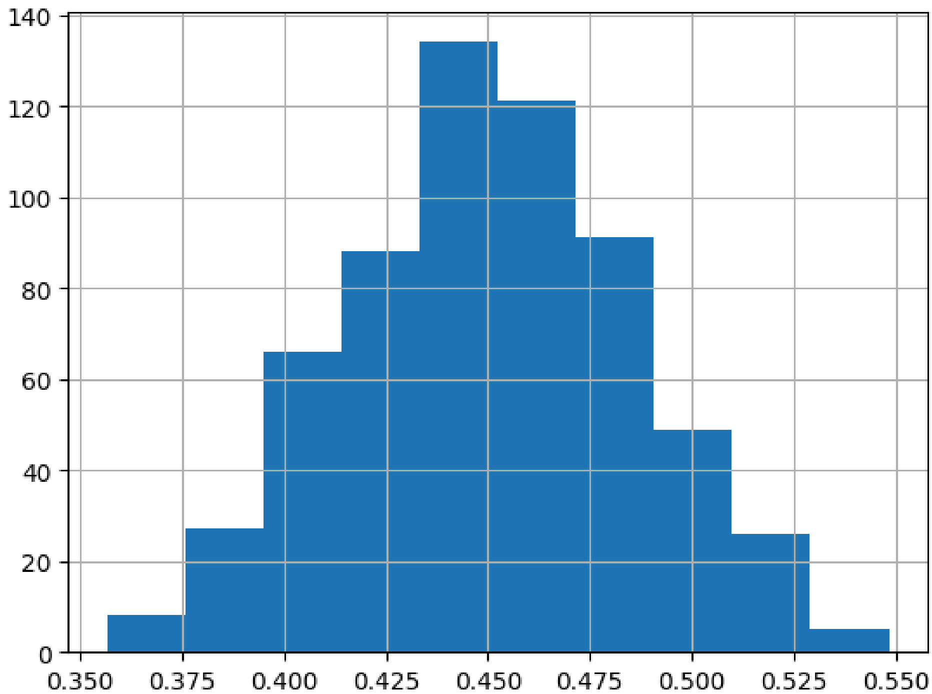
Min, max and mean of preds:

(0.35658523, 0.5482287, 0.45032236)

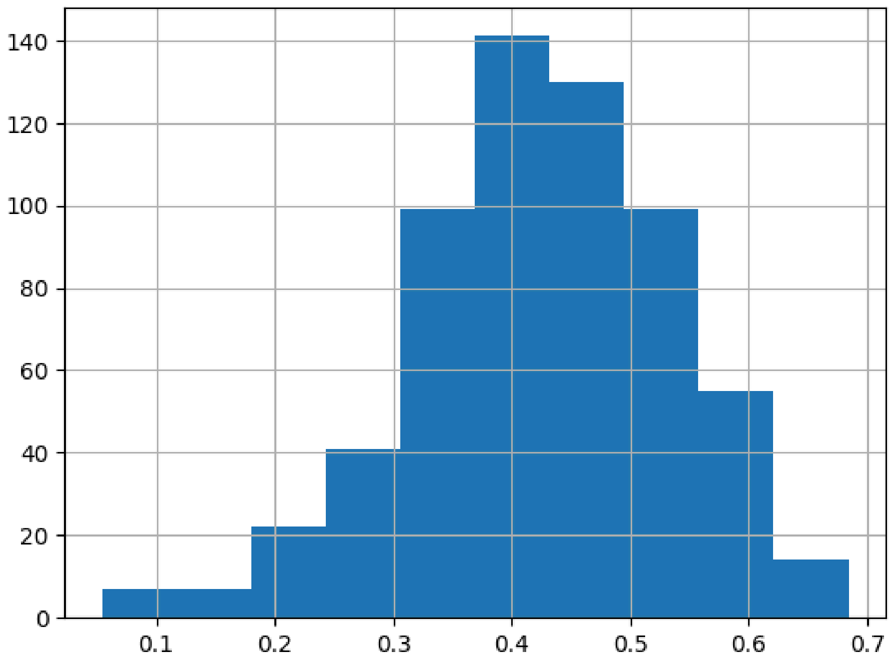
min(test_targets), max(test_targets), test_targets.mean()

(0.054, 0.684, 0.42383412)

Predictions histogram:

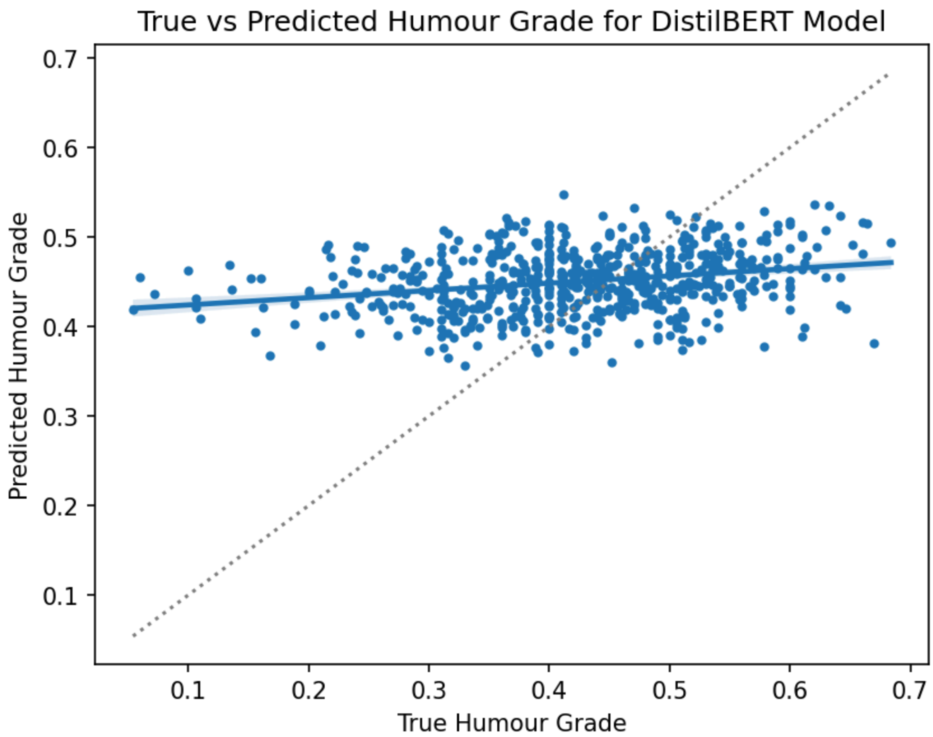


Test targets histogram:



Comparing the two histograms we can see that both follow a normal distribution.
Test targets span approximately from 0.1 to 0.7 while predicted ratings span from 0.35 to 0.55, indicating that the model's predictions are less spread out than the ground truth. As mentioned in the notebook, our regressor tends to smooth down the extreme rating values to make them closer to the mean because we can observe a bias towards the mean value in predicted values

True vs predicted humor glade:



b. Task 2:

Train and Val loss:

```
Epoch 1/5: 100%|██████████| 20/20 [00:49<00:00,  2.47s/it]
Train Loss: 0.0494, Val Loss: 0.0186
Epoch 2/5: 100%|██████████| 20/20 [00:48<00:00,  2.40s/it]
Train Loss: 0.0144, Val Loss: 0.0124
Epoch 3/5: 100%|██████████| 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0119, Val Loss: 0.0120
Epoch 4/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0114, Val Loss: 0.0116
Epoch 5/5: 100%|██████████| 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0110, Val Loss: 0.0114
```

Test results:

```
Test loss: 0.0121
Test Mean Squared Error: 0.0124
```

Min, max and mean of preds:

```
(0.36339033, 0.54666567, 0.45857787)
```

min(test_targets), max(test_targets), test_targets.mean()

```
(0.054, 0.684, 0.42383412)
```

c. Difference between model A and B:
Model A (without preprocessing) achieves a lower test MSE (0.0119) than Model B (with preprocessing) (0.0124). Both models end up with similar loss values meaning that preprocessing didn’t significantly improve the performance or the training. These could be because some aspects that indicate humor (like a miss-spelling or typing ‘heeeey’ instead of ‘hey’) got removed during preprocessing and since the tokens are more normal (similar), we lost the diversity we had in the dataset and it is hard to distinguish humorous content.

2. Task 3: Augment the training data twice for the regression model
   a. Synonym augmentation:

```
Training examples before augmentation:
4932
Training examples after augmentation:
9864
```

```
Epoch 1/5: 100%|██████████| 39/39 [01:36<00:00,  2.48s/it]
Train Loss: 0.1186, Val Loss: 0.0130
Epoch 2/5: 100%|██████████| 39/39 [01:35<00:00,  2.46s/it]
Train Loss: 0.0129, Val Loss: 0.0126
Epoch 3/5: 100%|██████████| 39/39 [01:35<00:00,  2.45s/it]
Train Loss: 0.0127, Val Loss: 0.0122
Epoch 4/5: 100%|██████████| 39/39 [01:35<00:00,  2.45s/it]
Train Loss: 0.0124, Val Loss: 0.0118
Epoch 5/5: 100%|██████████| 39/39 [01:35<00:00,  2.45s/it]
```

```
Train Loss: 0.0121, Val Loss: 0.0116

Test loss: 0.0122
Test Mean Squared Error: 0.0125
```

b. Deletion Augmentation

```
Training examples before augmentation:
4932
Training examples after augmentation:
9864

Epoch 1/5: 100%|          | 39/39 [01:36<00:00,  2.46s/it]
Train Loss: 0.0222, Val Loss: 0.0127
Epoch 2/5: 100%|          | 39/39 [01:35<00:00,  2.46s/it]
Train Loss: 0.0124, Val Loss: 0.0117
Epoch 3/5: 100%|          | 39/39 [01:35<00:00,  2.45s/it]
Train Loss: 0.0121, Val Loss: 0.0115
Epoch 4/5: 100%|          | 39/39 [01:35<00:00,  2.46s/it]
Train Loss: 0.0118, Val Loss: 0.0112
Epoch 5/5: 100%|          | 39/39 [01:35<00:00,  2.46s/it]
Train Loss: 0.0116, Val Loss: 0.0110

Test loss: 0.0115
Test Mean Squared Error: 0.0118
```

c. Comparison:
The results are close to each other but in synonym augmentation training loss started high (0.1186) but gradually improved. Final test MSE: 0.0125, which is slightly worse than the baseline. In deletion augmentation, Training loss was lower from the start (0.0222) and improved more consistently. Final test MSE: 0.0118, which is slightly better than the synonym augmentation

3. Task 4: Perform ensembling of three regressors
Results for each regressor are shown below:

```
Epoch 1/5: 100%|| 20/20 [00:49<00:00,  2.48s/it]
Train Loss: 0.0324, Val Loss: 0.0140
Epoch 2/5: 100%|| 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0124, Val Loss: 0.0122
Epoch 3/5: 100%|| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0115, Val Loss: 0.0118
Epoch 4/5: 100%|| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0111, Val Loss: 0.0116
Epoch 5/5: 100%|| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0107, Val Loss: 0.0114
Test loss: 0.0124
Test Mean Squared Error: 0.0127
/usr/local/lib/python3.11/dist-packages/transformers/optimization.py:591:
FutureWarning: This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
```

```
  warnings.warn(
Epoch 1/5: 100%|██████████| 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0165, Val Loss: 0.0125
Epoch 2/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0122, Val Loss: 0.0118
Epoch 3/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0115, Val Loss: 0.0115
Epoch 4/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0111, Val Loss: 0.0111
Epoch 5/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0105, Val Loss: 0.0108
Test loss: 0.0115
Test Mean Squared Error: 0.0118
/usr/local/lib/python3.11/dist-packages/transformers/optimization.py:591:
FutureWarning: This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
Epoch 1/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0477, Val Loss: 0.0191
Epoch 2/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0138, Val Loss: 0.0123
Epoch 3/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0116, Val Loss: 0.0118
Epoch 4/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0111, Val Loss: 0.0116
Epoch 5/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0110, Val Loss: 0.0113
Test loss: 0.0123
Test Mean Squared Error: 0.0126
```

Test MSE:

```
Ensemble Test MSE : 0.0122
```

Ensembling multiple models typically improves the performance. In this case, the ensemble model achieves a Test MSE of 0.0122, compared to the Task 1 model's Test MSE of 0.0119. This means that the ensemble slightly worsened the performance. Normally, ensembling helps stabilize predictions and reduce overfitting to random noise but since all three regressors were trained on the same data, they probably learned representations that were too similar. The model from Task 1 had slightly lower test loss, which probably caused less overfitting.

Task 5: Build and evaluate the multi-task learning regressor

Train and validation results:

```
Epoch 1/5: 100%|██████████| 20/20 [00:49<00:00,  2.47s/it]
Train Loss: 0.0488, Val Loss: 0.0454
Epoch 2/5: 100%|██████████| 20/20 [00:48<00:00,  2.41s/it]
Train Loss: 0.0303, Val Loss: 0.0370
Epoch 3/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
```

```
Train Loss: 0.0250, Val Loss: 0.0282
Epoch 4/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0191, Val Loss: 0.0217
Epoch 5/5: 100%|██████████| 20/20 [00:48<00:00,  2.42s/it]
Train Loss: 0.0160, Val Loss: 0.0201
```

Test results:

```
Test loss: 0.0130
Test Mean Squared Error: 0.0133
```

The MTL model performed slightly worse on the humor prediction task, with a Test MSE of 0.0133, compared to 0.0119 for the single-task model. Adding the offense might have distracted the model from predicting the humor because the training and validation loose are higher throughout the process (probably because it was learning two outcomes). Even though the test loss was higher, the train loss in MTL was also higher than task 1, meaning it generalized better and avoided overfitting. (task 1 train loss was very low). Humor and offense might have been less correlated than we thought so learning them both at the same time probably caused some conflicts.

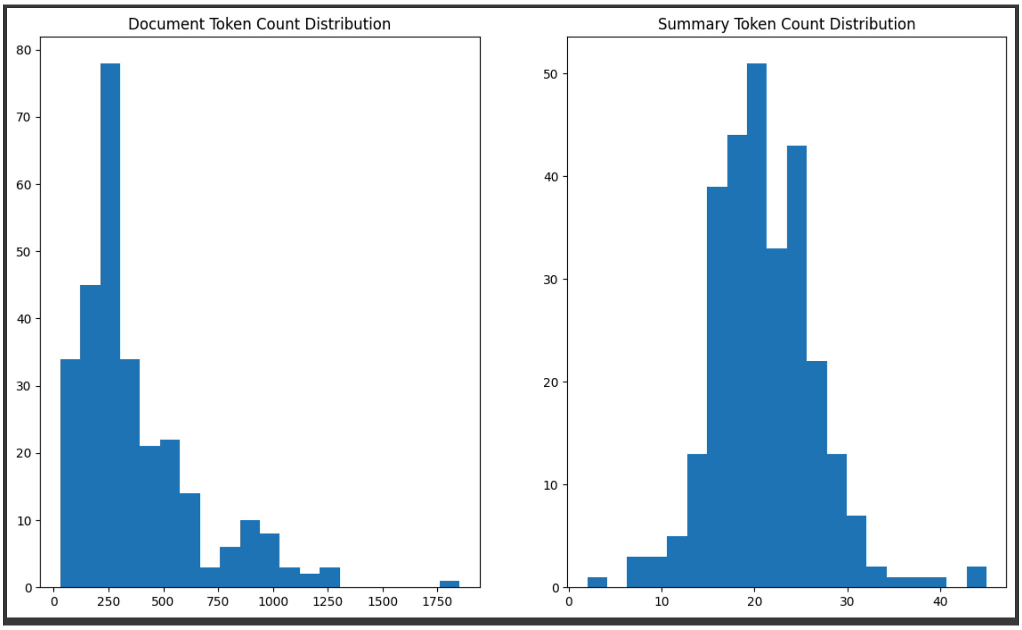
Part E: Summarisation and Data Generation

(implemented in Lab6_Summarisation_and_Data_Generation_240753803.ipynb)

1. Task 1: Analyse the XSum summarisation dataset

```
Statistics for train set:
  Source Documents: Mean = 371.50, Std = 315.83
  Target Summaries: Mean = 21.05, Std = 5.17

Statistics for validation set:
  Source Documents: Mean = 375.52, Std = 277.19
```

```
Target Summaries: Mean = 21.16, Std = 5.53
```



The average length of source documents in booth training and validation set is 371-375 tokens per document but the std is quite high, showing that document lengths in train and validation data are quite high.The left-skewed distribution in the histogram shows that most documents are relatively short, but there are some much longer articles (1000+ tokens), which cause a long tail. The summaries have an average of 21 tokens with a low std of approximately 5 meaning all summaries are in similar range and there aren't any very long or very short summaries.

2. Task 2: Fine-tune and evaluate T5 for the XSum summarisation task
   Training loss in 5 epochs:

```
Epoch 1: Training loss: 3.098968982696533
Epoch 2: Training loss: 3.5333352088928223
Epoch 3: Training loss: 2.9322493076324463
Epoch 4: Training loss: 2.785102605819702
Epoch 5: Training loss: 2.7536585330963135
```

- Fine tuned results:

```
A flood warning has been issued in the area after the floods in Newton Stewart caused a flood warning and a flood warning.
```

```
{'rouge1': 0.22580645161290322,
 'rouge2': 0.06666666666666667,
 'rougeL': 0.1935483870967742,
 'rougeLsum': 0.1935483870967742}
```

- Generic output:

```
the flood protection plan was right but backed calls to speed up the process . the full cost of damage in Newton Stewart is still being assessed . many roads in Peeblesshire remain badly affected by standing water .
```

```
{'rouge1': 0.18666666666666665,
```

```
 'rouge2': 0.05479452054794521,
 'rougeL': 0.08,
 'rougeLsum': 0.08}
```

- Comparison:
  The fine tuned model captured the flood warning aspect but the term “flood warning” keeps being repeated and the answer isn’t coherent. The non fine tuned model has a more generic result with a broader overview to the document.
  ROUGE-1 and ROUGE-2 improved slightly with fine-tuning.
  ROUGE-L and ROUGE-Lsum showed significant improvement, meaning the fine-tuned model is better at maintaining meaningful long-span structures.
  Overall, The fine-tuned model performs better but its output is not very coherent because of the repetitions. Since this model also improved accuracy, further training or different hyperparameters could enhance the outputs.

3. Build and evaluate a T5-based model for data generation using keywords
   Training loss:
   ```
   Epoch 1: Training loss: 3.7324516773223877
   Epoch 2: Training loss: 3.576667308807373
   Epoch 3: Training loss: 4.1585798263549805
   Epoch 4: Training loss: 4.1764235496521
   Epoch 5: Training loss: 3.782414197921753
   ```

   Output:
   ```
   disruption Lamington viaduct commercial thoroughfare multi-agency
   neglected
   ```

The training loss ended up decreasing but there was an increase in the second epoch that could indicate overfitting. Also the output isn’t generating meaningful text, it just looks like it has memorized the documents and repeated it. The model might need different hyperparameters, more training or different preprocessing techniques.