Neural Networks and NLP (ECS7001P)

Assignment 2

Salva Karimisahari 240753803

Table of Contents

PART A: Information Extraction 1: Training a Named Entity Resolver

Task 1:

As mentioned in the assignment, a dropout layer has been applied to the word embeddings to reduce overfitting. A 2 layer bidirectional GRU with the input size of embedding sizes and specified hidden size has been defined to encode contextual information from previous and next tokens. The output of the GRU (word_output) which has the shape [batch_size, max_sen_length, 2*hidden size (bidirectional)]] is passed through a 2 layer feed forward neural network, Therefore the first layer of the FFNN has the input size of hidden_size*2, ReLU activation function and dropout. The second FFNN layer has input size of hidden layer, ReLU activation function and dropout. The output layer has an input of size hidden_layer and output of size num_labels (the 5 NER tags). The output ner_scores has shape [batch_size, max_sent_length,num_labels] and includes per-token class scores.
The code has been implemented as:

```
class BiGRUFFNN(nn.Module):
   """
   Task 1

   Begin
   """

   def __init__(self,
                embedding_size,
                embedding_dropout_rate,
                hidden_size,
                ffnn_layer,
                hidden_dropout_rate,
                num_labels):
       super(BiGRUFFNN, self).__init__()
       #applying dropout to word embeddings
       self.embedding_dropout = nn.Dropout(embedding_dropout_rate)
       #defining bi-directional GRU
       self.bigru=nn.GRU(input_size=embedding_size,
hidden_size=hidden_size, num_layers=2, bias=True, batch_first=True,
dropout=hidden_dropout_rate, bidirectional=True)
       #FFNN
       self.ffnn = nn.Sequential(
           #since input is bidirrectional
           nn.Linear(2*hidden_size, hidden_size),
           nn.ReLU(),
           nn.Dropout(hidden_dropout_rate),
```

```
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(hidden_dropout_rate),
            #output layer, no dropout
            nn.Linear(hidden_size, num_labels)
        )

    def forward(self, word_embeddings):
        """
        word_embeddings: Tensor of shape [batch_size, max_sent_length,
embedding_size]
        """
        word_embeddings = self.embedding_dropout(word_embeddings)
        word_output, _ = self.bigru(word_embeddings)
        ner_scores = self.ffnn(word_output)

        return ner_scores

    """
    End Task 1
    """
```

Which resulted in this model architecture:

```
BiGRUFFNN(
  (embedding_dropout): Dropout(p=0.5, inplace=False)
  (bigru): GRU(100, 50, num_layers=2, batch_first=True, dropout=0.2,
bidirectional=True)
  (ffnn): Sequential(
    (0): Linear(in_features=100, out_features=50, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=50, out_features=50, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=50, out_features=5, bias=True)
  )
)
```

Task 2:

In order to define the output as requested in the assignment, first we take the prediction with the highest probability as the label, Then we loop through each sentence and divide the possibilities into two sections. Either we have a current label and reach to an ‘o’ label which indicates the ending of an NER span, we add (sid,start,idx-1, label_id) to the list, or we have a current label and we reach a new label which indicates a different NER sequence beginning so we add the previous span to the list ad set the current index to be the new starting point for an NER sequence.

After computing the spans saved in pred_entites, we use the gold sentence to calculate our evaluations. Since the f1 score for evaluating NER tasks is strict, a TP would be when the prediction sequence completely matches the cold sequence. A FN would be sth that exists in the gold sentence but not in our predictions and a FP would be sth that has been predicted but doesn't exist in the gold sequence.

```
def eval(eval_fd_list, device):
tp, fn, fp = 0,0,0
 with torch.no_grad():
     for word_embeddings, _, gold,sent_lens in eval_fd_list:
         word_embeddings = torch.Tensor(word_embeddings).to(device)
         predictions = ner_model(word_embeddings)
         predictions = predictions.detach().cpu().numpy()
         """
         Task 2 create the predictions of NER from the IO label
         e.g.
         0 I         O
         1 met       O
         2 John      PER
         3 this      O
         4 afternoon O
         should give you a person NE John (x,2,2,1)
         where x is the sentence id in the batch, and 2,2 are the start
and end indices of the NE,
         1 is the id for 'PER'

         Begin
         """
         pred_entities = set()
         labels = np.argmax(predictions, axis=-1)

         for sid, (sent_pred, sent_len) in enumerate(zip(labels,
sent_lens)):
             curr_label = None
             start = None
             #only evaluating the idx within sentence length to ignore
paddings
             for idx in range(sent_len):
                 label = sent_pred[idx]
                 # getting to an 'o' label
                 if label == 0:
                     if curr_label is not None:
```

```
                         pred_entities.add((sid, start, idx - 1,
curr_label))
                         curr_label = None
                         start = None
                 else:
                     #getting to a different label and starting a new
start
                     if curr_label != label:
                         if curr_label is not None:
                             pred_entities.add((sid, start, idx - 1,
curr_label))
                         curr_label = label
                         start = idx
             # add final entity if it ends at the last token
             if curr_label is not None:
                 pred_entities.add((sid, start, sent_len - 1,
curr_label))

         # computing TP, FN, FP
         tp += len(pred_entities & gold)
         fn += len(gold - pred_entities)
         fp += len(pred_entities - gold)




         """
         End Task 2
         """

 p = 0.0 if tp == 0 else tp*1.0/(tp+fp)
 r = 0.0 if tp == 0 else tp*1.0/(tp+fn)
 f = 0.0 if tp == 0 else 2*p*r/(p+r)
 print("F1 : {:.2f}%".format(f * 100))
 print("Precision: {:.2f}%".format(p * 100))
 print("Recall: {:.2f}%".format(r * 100))
```

Training and test set results:

```
Load 141 training batches from train.conll03.json
Load 33 dev batches from dev.conll03.json
Load 35 test batches from test.conll03.json

Starting training epoch 1/25
Time used for epoch 1: 0 m 1 s
```

```
Evaluating on dev set after epoch 1/25:
F1 : 9.29%
Precision: 20.32%
Recall: 6.02%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 2/25
Time used for epoch 2: 0 m 0 s
Evaluating on dev set after epoch 2/25:
F1 : 30.08%
Precision: 48.88%
Recall: 21.73%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 3/25
Time used for epoch 3: 0 m 0 s
Evaluating on dev set after epoch 3/25:
F1 : 47.66%
Precision: 56.62%
Recall: 41.15%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 4/25
Time used for epoch 4: 0 m 0 s
Evaluating on dev set after epoch 4/25:
F1 : 57.69%
Precision: 64.69%
Recall: 52.05%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 5/25
Time used for epoch 5: 0 m 0 s
Evaluating on dev set after epoch 5/25:
F1 : 62.94%
Precision: 68.14%
Recall: 58.48%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 6/25
Time used for epoch 6: 0 m 0 s
Evaluating on dev set after epoch 6/25:
F1 : 64.52%
Precision: 70.39%
Recall: 59.56%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 7/25
Time used for epoch 7: 0 m 0 s
Evaluating on dev set after epoch 7/25:
F1 : 65.67%
Precision: 68.88%
```

```
Recall: 62.76%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 8/25
Time used for epoch 8: 0 m 0 s
Evaluating on dev set after epoch 8/25:
F1 : 67.10%
Precision: 70.90%
Recall: 63.68%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 9/25
Time used for epoch 9: 0 m 0 s
Evaluating on dev set after epoch 9/25:
F1 : 68.39%
Precision: 74.14%
Recall: 63.46%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 10/25
Time used for epoch 10: 0 m 0 s
Evaluating on dev set after epoch 10/25:
F1 : 69.80%
Precision: 74.17%
Recall: 65.92%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 11/25
Time used for epoch 11: 0 m 0 s
Evaluating on dev set after epoch 11/25:
F1 : 70.37%
Precision: 72.63%
Recall: 68.24%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 12/25
Time used for epoch 12: 0 m 0 s
Evaluating on dev set after epoch 12/25:
F1 : 70.96%
Precision: 71.86%
Recall: 70.08%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 13/25
Time used for epoch 13: 0 m 0 s
Evaluating on dev set after epoch 13/25:
F1 : 71.75%
Precision: 74.46%
Recall: 69.24%
Time used for evaluate on dev set: 0 m 0 s
```

```
Starting training epoch 14/25
Time used for epoch 14: 0 m 0 s
Evaluating on dev set after epoch 14/25:
F1 : 72.89%
Precision: 74.32%
Recall: 71.51%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 15/25
Time used for epoch 15: 0 m 0 s
Evaluating on dev set after epoch 15/25:
F1 : 71.90%
Precision: 74.15%
Recall: 69.77%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 16/25
Time used for epoch 16: 0 m 0 s
Evaluating on dev set after epoch 16/25:
F1 : 72.87%
Precision: 73.54%
Recall: 72.21%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 17/25
Time used for epoch 17: 0 m 0 s
Evaluating on dev set after epoch 17/25:
F1 : 73.97%
Precision: 77.07%
Recall: 71.10%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 18/25
Time used for epoch 18: 0 m 0 s
Evaluating on dev set after epoch 18/25:
F1 : 74.11%
Precision: 75.73%
Recall: 72.57%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 19/25
Time used for epoch 19: 0 m 0 s
Evaluating on dev set after epoch 19/25:
F1 : 74.91%
Precision: 76.61%
Recall: 73.29%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 20/25
Time used for epoch 20: 0 m 0 s
Evaluating on dev set after epoch 20/25:
```

```
F1 : 74.11%
Precision: 77.59%
Recall: 70.92%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 21/25
Time used for epoch 21: 0 m 0 s
Evaluating on dev set after epoch 21/25:
F1 : 74.98%
Precision: 75.81%
Recall: 74.17%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 22/25
Time used for epoch 22: 0 m 0 s
Evaluating on dev set after epoch 22/25:
F1 : 75.56%
Precision: 77.70%
Recall: 73.53%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 23/25
Time used for epoch 23: 0 m 0 s
Evaluating on dev set after epoch 23/25:
F1 : 75.49%
Precision: 78.01%
Recall: 73.12%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 24/25
Time used for epoch 24: 0 m 0 s
Evaluating on dev set after epoch 24/25:
F1 : 75.37%
Precision: 76.29%
Recall: 74.47%
Time used for evaluate on dev set: 0 m 0 s

Starting training epoch 25/25
Time used for epoch 25: 0 m 0 s
Evaluating on dev set after epoch 25/25:
F1 : 76.62%
Precision: 77.81%
Recall: 75.46%
Time used for evaluate on dev set: 0 m 0 s

Training finished!
Time used for training: 0 m 17 s

Evaluating on test set:
F1 : 73.45%
Precision: 74.38%
```

```
Recall: 72.54%
Time used for evaluate on test set: 0 m 0 s
```

F1 score on the test set:

```
F1 : 73.45%
```

PART B: Information Extraction 2: Coreference Resolution

Task 1:

In order to complete the generate_data() function, previously implemented functions have been used as seen below:

```
import re, json

def preprocess_arabic_text(text):
 #diacrtic unicodes are found using regular expressions
 diacritics_unicode = re.compile(r'[\u0617-\u061A\u064B-\u0652]')
 #the diacrtics are then removed
 text = re.sub(diacritics_unicode, "", text)
 return text

def get_data(json_file, is_training, preprocess_text):
   processed_docs = []

   for line in open(json_file):

     # read the document in
     doc = json.loads(line)

     # check that there are coreferent mentions in this document
     clusters = doc['clusters']

     sentences = doc['sentences']

     if(preprocess_text==True):
         preprocessed_sents = [[preprocess_arabic_text(t) for t in
sent] for sent in sentences]
         doc['sentences'] = preprocessed_sents

     if len(clusters) == 0:
         continue

     """
```

```
    Task 1

    Begin
    """

    #  get the mentions and their cluster information.
    gold_mentions, gold_mention_map, cluster_ids, num_mentions =
get_mentions(clusters)

    # splits the mentions into two arrays, one representing the start
indices,
    # and the other for the end indices
    raw_starts, raw_ends = zip(*gold_mentions)

    # pad sentences, create glove sentence embeddings, create mention
starts and ends for padded document
    word_emb, starts, ends = tensorize_doc_sentences(doc['sentences'],
gold_mentions)

    # generate (anaphor, antecedent) pairs and their labels
    mention_pairs, mention_pair_labels, raw_mention_pairs =
generate_pairs(num_mentions, cluster_ids, starts, ends, raw_starts,
raw_ends, is_training)
    mention_pairs, mention_pair_labels =
np.array(mention_pairs),np.array(mention_pair_labels)

    # add the processed document to the list
    processed_docs.append((word_emb[0], mention_pairs[0],
mention_pair_labels[0], clusters, raw_mention_pairs))
    """
    End Task 1
    """
  return processed_docs
```

The function get_mentions(clusters) returns a list of gold mentions, a mapping from mention to index, cluster ID per mention and number of mentions. In order to convert sentences to padded embeddings, the function tensorize_doc_sentence is called on the sentences and gold mentions and returns the word_emb ready to be used in the model and (starts, ends) which are mention positions after the padding has happened. To generate mention_pairs, the function geenrate_pairs is called on the number of mentions, cluster_ids, starts, ends, raw_starts, raw_ends and acts accordingly to the fact that we are on training or evaluation/test mode.

Task 2:

In this task the code for init and forward functions have been completed according to the description. First, 3 dropout layers are defined, embedding_dropout which is applied to the input word embeddings, flatten_dropout which is applied after the BiLSTM output is flattened, ffnn_dropout which is applied after each hidden layer in the FFNN. After that, 2 LSTM layers were defined so that the first one would take the embeddings as input and the second one refined the output of the first one. In the FFNN, we first apply a linear projection to reduce the input size to HIDDEN_SIZE, after that we have two hidden layers with ReLU and dropout, the final output layer outputs a single logit as a probability score.

```
import torch.nn.functional as F

# Model Definition
class CoreferenceModel(nn.Module):

   """
   Task 2

   Begin
   """
   def __init__(self):
       super(CoreferenceModel, self).__init__()

       # Your code goes here

       # Dropout layers
       self.embedding_dropout = nn.Dropout(EMBEDDING_DROPOUT_RATE)
       self.flatten_dropout = nn.Dropout(HIDDEN_DROPOUT_RATE)
       self.ffnn_dropout = nn.Dropout(HIDDEN_DROPOUT_RATE)

       # Bidirectional LSTM layers
       self.bi_lstm1 = nn.LSTM(EMBEDDING_SIZE, HIDDEN_SIZE,
batch_first=True, bidirectional=True,dropout= HIDDEN_DROPOUT_RATE)
       self.bi_lstm2 = nn.LSTM(2 * HIDDEN_SIZE, HIDDEN_SIZE,
batch_first=True, bidirectional=True,dropout=HIDDEN_DROPOUT_RATE)

       self.linear_layer = nn.Linear(8 * HIDDEN_SIZE, HIDDEN_SIZE)
       # Fully Connected Layers
       self.ffnn_layers = nn.ModuleList()
       for _ in range(NUM_FFN_LAYER):

         self.ffnn_layers.append(nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE))

       # Output layer
       self.output_layer = nn.Linear(HIDDEN_SIZE, 1)
```

```
        self.sigmoid = nn.Sigmoid()

    def forward(self, word_embeddings, mention_pairs):
        """
        word_embeddings: Tensor of shape (batch_size, num_sents,
num_words, embedding_size)
        mention_pairs: Tensor of shape (batch_size, num_pairs, 4)
        """
        # Reshape for LSTM: merge first two dimensions if necessary
        # origin shape is (batch_size, num_sents, num_words,
embedding_size)
        # We'll treat it as (batch_size*num_sents, num_words,
embedding_size)
        word_embeddings = word_embeddings.view(-1,
word_embeddings.size(2), word_embeddings.size(3))

        # Your code goes here
        # Apply dropout
        word_embeddings = self.embedding_dropout(word_embeddings)

        # Apply Bidirectional LSTM
        word_output, _ = self.bi_lstm1(word_embeddings)
        word_output, _ = self.bi_lstm2(word_output)

        # Flatten word_output: reshape to
(batch_size*num_sents*num_words, 2*hidden_size)
        flatten_word_output = word_output.contiguous().view(-1, 2 *
HIDDEN_SIZE)  # (batch_size*num_sents*num_words, 2*hidden_size)
        flatten_word_output = self.flatten_dropout(flatten_word_output)
#  (batch_size*num_sents*num_words, 2*hidden_size)

        # Gather mention pair embeddings
        flatten_mention_pairs = mention_pairs.contiguous().view(-1)  #
Shape: batch_size*num_pairs*4
        flatten_mention_pair_emb =
torch.index_select(flatten_word_output,0,flatten_mention_pairs)  #
Shape: (batch_size*num_pairs*4, 2*hidden_size)
        mention_pair_emb =
flatten_mention_pair_emb.contiguous().view(-1, 8 * HIDDEN_SIZE)  #
Shape: (batch_size*num_pairs, 8*hidden_size)

        # Apply FFNN layers
        #first laeyr
```

```
        x = self.linear_layer(mention_pair_emb)
        x = F.relu(x)
        #other layers
        for layer in self.ffnn_layers:
            x = layer(x)
            x = F.relu(x)
            x = self.ffnn_dropout(x)

        # Output layer
        mention_pair_scores = self.output_layer(x)  # (batch_size * 
num_pairs, 1)
        mention_pair_scores = self.sigmoid(mention_pair_scores)  # 
(batch_size * num_pairs, 1)

        # Squeeze the last dimension
        mention_pair_scores = mention_pair_scores.squeeze(1)  # 
(batch_size * num_pairs)

        return mention_pair_scores
    """
    End Task 2
    """
```

Task 3:

In this part, the evaluate_coref() function has been completed so that it converts each cluster in gold_clusters into a tuple. Each mention inside those clusters is also converted into a tuple to keep the consistency. After that each mention gets mapped to its corresponding gold cluster in the mention_to_gold dictionary. The get_predicted_clusters function is used to group the predicted mention pairs into full clusters and use them for evaluation.

```
def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):
    """
    Task 3

    Begin
    """
    gold_clusters = [tuple([tuple(m) for m in cluster]) for cluster in 
gold_clusters]


    # mention to gold is a {mention: cluster of mentions it belongs, 
including the present mention} map
    mention_to_gold = {}
    for cluster in gold_clusters:
        for mention in cluster:
```

```
        mention_to_gold[mention] = cluster


  # get the predicted clusters and the map of mention to predicted
cluster
  predicted_clusters, mention_to_predicted =
get_predicted_clusters(predicted_mention_pairs)
  evaluator.update(predicted_clusters, gold_clusters,
mention_to_predicted, mention_to_gold)

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

The results on the train set was as following:

```
Epoch 1/10, Loss: 205.9651
Average F1 (py): 35.88%
Average precision (py): 49.13%
Average recall (py): 47.05%
Time used: 0 m 7 s
Epoch 2/10, Loss: 189.5476
Average F1 (py): 38.41%
Average precision (py): 57.88%
Average recall (py): 39.67%
Time used: 0 m 9 s
Epoch 3/10, Loss: 179.1496
Average F1 (py): 35.64%
Average precision (py): 47.98%
Average recall (py): 51.57%
Time used: 0 m 8 s
Epoch 4/10, Loss: 171.4944
Average F1 (py): 33.42%
Average precision (py): 38.12%
Average recall (py): 54.09%
Time used: 0 m 8 s
Epoch 5/10, Loss: 159.9734
Average F1 (py): 36.57%
Average precision (py): 44.54%
Average recall (py): 53.39%
Time used: 0 m 8 s
Epoch 6/10, Loss: 151.6012
Average F1 (py): 32.80%
Average precision (py): 36.52%
Average recall (py): 58.32%
Time used: 0 m 8 s
Epoch 7/10, Loss: 145.9409
Average F1 (py): 31.84%
Average precision (py): 36.72%
```

```
Average recall (py): 61.39%
Time used: 0 m 9 s
Epoch 8/10, Loss: 137.5887
Average F1 (py): 31.54%
Average precision (py): 35.04%
Average recall (py): 59.43%
Time used: 0 m 8 s
Epoch 9/10, Loss: 129.8688
Average F1 (py): 31.91%
Average precision (py): 36.80%
Average recall (py): 61.42%
Time used: 0 m 8 s
Epoch 10/10, Loss: 128.4432
Average F1 (py): 31.02%
Average precision (py): 35.75%
Average recall (py): 62.29%
Time used: 0 m 9 s
```

Task 4:

- Would the performance decrease if we do not preprocess the text? If yes (or no), then why?
  Yes, the performance is more likely to increase because without preprocessing the model may treat two same words with mild differences as different tokens and cause embedding mismatches and worse generalization. Also in Arabic, most words come from a 3-letter root and words sharing the same root will encapsulate the same meaning in different grammatical spaces so preprocessing is crucial and needs to be more advanced.
- Experiment with different values for max antecedent (MAX_ANT) and negative ratio (NEG_RATIO), what do you observe?
  - Smaller MAX_ANT (50): resulted in faster training, higher precision and lower recall as we have fewer noisy pairs

```
Epoch 1/10, Loss: 228.8525
Average F1 (py): 22.13%
Average precision (py): 54.57%
Average recall (py): 15.37%
Time used: 0 m 5 s
Epoch 2/10, Loss: 203.3845
Average F1 (py): 36.15%
Average precision (py): 50.61%
Average recall (py): 44.89%
Time used: 0 m 6 s
Epoch 3/10, Loss: 186.3974
Average F1 (py): 38.04%
Average precision (py): 54.38%
Average recall (py): 44.25%
Time used: 0 m 5 s
Epoch 4/10, Loss: 178.4972
Average F1 (py): 35.40%
Average precision (py): 46.85%
Average recall (py): 51.06%
```

```
Time used: 0 m 6 s
Epoch 5/10, Loss: 171.6600
Average F1 (py): 35.89%
Average precision (py): 47.74%
Average recall (py): 49.89%
Time used: 0 m 6 s
Epoch 6/10, Loss: 160.5845
Average F1 (py): 40.41%
Average precision (py): 56.50%
Average recall (py): 44.70%
Time used: 0 m 6 s
Epoch 7/10, Loss: 152.7946
Average F1 (py): 33.06%
Average precision (py): 37.02%
Average recall (py): 57.48%
Time used: 0 m 5 s
Epoch 8/10, Loss: 146.7445
Average F1 (py): 33.85%
Average precision (py): 40.17%
Average recall (py): 56.12%
Time used: 0 m 6 s
Epoch 9/10, Loss: 139.6094
Average F1 (py): 30.51%
Average precision (py): 33.83%
Average recall (py): 60.63%
Time used: 0 m 5 s
Epoch 10/10, Loss: 132.8594
Average F1 (py): 34.65%
Average precision (py): 40.36%
Average recall (py): 59.27%
Time used: 0 m 6 s
```

- Larger MAX_ANT (350): resulted in slower training as there is more context, higher precision and lower recall.

```
Epoch 1/10, Loss: 235.3640
Average F1 (py): 10.31%
Average precision (py): 38.53%
Average recall (py): 6.37%
Time used: 0 m 8 s
Epoch 2/10, Loss: 214.5494
Average F1 (py): 35.03%
Average precision (py): 47.01%
Average recall (py): 48.09%
Time used: 0 m 8 s
Epoch 3/10, Loss: 194.9652
Average F1 (py): 33.58%
Average precision (py): 42.16%
Average recall (py): 55.73%
Time used: 0 m 9 s
```

```
Epoch 4/10, Loss: 180.1448
Average F1 (py): 34.15%
Average precision (py): 43.16%
Average recall (py): 54.22%
Time used: 0 m 7 s
Epoch 5/10, Loss: 165.8279
Average F1 (py): 35.75%
Average precision (py): 45.75%
Average recall (py): 52.35%
Time used: 0 m 9 s
Epoch 6/10, Loss: 157.4027
Average F1 (py): 34.87%
Average precision (py): 42.21%
Average recall (py): 54.56%
Time used: 0 m 8 s
Epoch 7/10, Loss: 150.1718
Average F1 (py): 34.78%
Average precision (py): 41.98%
Average recall (py): 55.58%
Time used: 0 m 9 s
Epoch 8/10, Loss: 141.2130
Average F1 (py): 34.90%
Average precision (py): 42.69%
Average recall (py): 56.07%
Time used: 0 m 7 s
Epoch 9/10, Loss: 134.1164
Average F1 (py): 34.78%
Average precision (py): 40.39%
Average recall (py): 58.32%
Time used: 0 m 9 s
Epoch 10/10, Loss: 131.7181
Average F1 (py): 35.80%
Average precision (py): 43.29%
Average recall (py): 56.67%
Time used: 0 m 8 s
```

- Smaller NEG_RATIO (1): meaning for every positive pair, we include one negative pair in training and because it has seen more positive samples compared to the previous model, it has a high recall cause it's more optimistic in prediction conferences and considers more things as coreferences. F1 score is lower because the model is more simplified now.

```
Epoch 1/10, Loss: 236.6063
Average F1 (py): 32.52%
Average precision (py): 40.99%
Average recall (py): 53.54%
Time used: 0 m 8 s
Epoch 2/10, Loss: 205.9544
Average F1 (py): 29.79%
Average precision (py): 34.12%
Average recall (py): 63.25%
```

```
Time used: 0 m 8 s
Epoch 3/10, Loss: 188.7671
Average F1 (py): 29.89%
Average precision (py): 34.98%
Average recall (py): 61.96%
Time used: 0 m 9 s
Epoch 4/10, Loss: 179.1391
Average F1 (py): 29.77%
Average precision (py): 33.78%
Average recall (py): 63.18%
Time used: 0 m 8 s
Epoch 5/10, Loss: 168.4817
Average F1 (py): 30.24%
Average precision (py): 33.89%
Average recall (py): 63.23%
Time used: 0 m 9 s
Epoch 6/10, Loss: 158.8553
Average F1 (py): 30.41%
Average precision (py): 36.38%
Average recall (py): 63.99%
Time used: 0 m 8 s
Epoch 7/10, Loss: 150.0259
Average F1 (py): 31.54%
Average precision (py): 35.51%
Average recall (py): 63.93%
Time used: 0 m 9 s
Epoch 8/10, Loss: 142.0127
Average F1 (py): 30.31%
Average precision (py): 35.66%
Average recall (py): 64.18%
Time used: 0 m 8 s
Epoch 9/10, Loss: 135.5323
Average F1 (py): 30.52%
Average precision (py): 36.66%
Average recall (py): 65.16%
Time used: 0 m 8 s
Epoch 10/10, Loss: 126.9871
Average F1 (py): 31.20%
Average precision (py): 36.27%
Average recall (py): 64.54%
Time used: 0 m 9 s
```

- Larger NEG_RATIO(3): this means more negative samples compared to positive ones and it will cause the model to favor non-coreference labels which would harm the recall.

```
Epoch 1/10, Loss: 220.5668
Average F1 (py): 18.61%
Average precision (py): 42.48%
Average recall (py): 13.57%
Time used: 0 m 9 s
Epoch 2/10, Loss: 195.1079
```

```
Average F1 (py): 34.82%
Average precision (py): 45.58%
Average recall (py): 51.25%
Time used: 0 m 7 s
Epoch 3/10, Loss: 180.5288
Average F1 (py): 39.71%
Average precision (py): 57.80%
Average recall (py): 42.87%
Time used: 0 m 9 s
Epoch 4/10, Loss: 172.6751
Average F1 (py): 36.16%
Average precision (py): 48.41%
Average recall (py): 49.20%
Time used: 0 m 8 s
Epoch 5/10, Loss: 162.9317
Average F1 (py): 37.07%
Average precision (py): 50.46%
Average recall (py): 49.18%
Time used: 0 m 10 s
Epoch 6/10, Loss: 154.2203
Average F1 (py): 38.73%
Average precision (py): 49.05%
Average recall (py): 48.60%
Time used: 0 m 7 s
Epoch 7/10, Loss: 145.9693
Average F1 (py): 39.28%
Average precision (py): 48.93%
Average recall (py): 48.45%
Time used: 0 m 9 s
Epoch 8/10, Loss: 137.9027
Average F1 (py): 39.34%
Average precision (py): 50.56%
Average recall (py): 48.96%
Time used: 0 m 8 s
Epoch 9/10, Loss: 132.9960
Average F1 (py): 37.35%
Average precision (py): 44.39%
Average recall (py): 52.37%
Time used: 0 m 9 s
Epoch 10/10, Loss: 125.9684
Average F1 (py): 37.28%
Average precision (py): 45.80%
Average recall (py): 53.34%
Time used: 0 m 7 s
```

- How would you improve the accuracy?
  - We could use pre-trained BERT embeddings instead of GLoVE which capture more contextual meanings.
  - We could do more enhanced and language specific preprocessing
  - We could use the attention mechanism to have dynamic mention weights

- We could explore different hyperparameters that were set for this task on the validation set.

PART C: Dialogue 1: Dialogue Act Tagging

Task 1:

The model uses an embedding layer to map vocab_size inputs into embed_sized ones. 2 BiLSTM layers to process the input embeddings and a dense layer to obtain the final representation. Through many trials, it was discovered that the paddings affected the training in a negative way and even the weighted cross entropy loss didn't help raising the accuracy, In order to avoid this problem, we find the index of the last non-padding token of a sequence in a batch and pass the last valid hidden state with of a meaningful sequence t the final dene layer.

```
class BiLSTMModel(nn.Module):
"""
  Task 1 1/4

  Begin
  """
  def __init__(self,vocab_size, embed_size, hidden_size, max_length):
      super(BiLSTMModel, self).__init__()

      # Include 2 BLSTM layers, in order to capture both the forward and backward
hidden states

      # Embedding layer
      # Bidirectional 1
      # Bidirectional 2
      # Dense layer

      self.embedder= nn.Embedding(vocab_size, embed_size)

      self.lstm1 = nn.LSTM(input_size=embed_size, hidden_size=hidden_size,
                      batch_first=True, bidirectional=True)

      self.lstm2 = nn.LSTM(input_size=2*hidden_size, hidden_size=hidden_size,
                      batch_first=True, bidirectional=True)

      self.dense = nn.Linear(2 * hidden_size, hidden_size)
```

```
    def forward(self, x):
        """
        x shape: (batch_size, seq_length)
        """
        out2 = self.embedder(x)
        out2, _ = self.lstm1(out2)
        out2, _ = self.lstm2(out2)

        # sentence_representation = x[:, -1, :]
        # probs = self.dense_layer(sentence_representation)

        # return probs

        idx = (x != 0).sum(1).clamp(min=1) - 1

        out = out2[torch.arange(out2.size(0)), idx]

        return self.dense(out)
    """
    End Task 1 1/4
    """

# Instantiate the model
model = BiLSTMModel(
    vocab_size=VOCAB_SIZE,
    embed_size=EMBED_SIZE,
    hidden_size=HIDDEN_SIZE,
    max_length=MAX_LENGTH
)
# Print model summary (PyTorch-style)
print(model)
```

```
BiLSTMModel(
  (embedder): Embedding(43732, 100)
  (lstm1): LSTM(100, 43, batch_first=True, bidirectional=True)
  (lstm2): LSTM(86, 43, batch_first=True, bidirectional=True)
  (dense): Linear(in_features=86, out_features=43, bias=True)
)
```

For the next part, the forward pass returns the probabilities over each class, we use the tag with the highest probability and compare it to actual tags to compute the accuracy.

```
def train(model, train_dataset, val_dataset, epoch_num, lr,
batch_size=1, device="cpu", weight=None):
    model.train()
    model = model.to(device)
    if weight:
```

```
      weight = torch.Tensor(weight).to(device)
   criterion = nn.CrossEntropyLoss(weight=weight).to(device)
   optimizer = optim.Adam(model.parameters(), lr=lr)
   train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
   val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)

   for epoch in range(epoch_num):
     model.train()
     total_loss = 0.0
     correct = 0
     """
     Task 1 2/3

     Begin
     """
     for batch_x, batch_y in train_loader:

         batch_x = batch_x.to(torch.int32).to(device)
         batch_y = batch_y.type(torch.LongTensor).to(device)


         # calculate the loss
         # backpropagation
         # compute the training accuracy

         #perform fwd pass
         outputs = model(batch_x)
         loss = criterion(outputs, batch_y.long())
         optimizer.zero_grad()
         loss.backward()
         optimizer.step()
         total_loss += loss.item()
         predicted = torch.argmax(outputs,dim = 1)
         correct += (predicted == batch_y).sum().item()





     avg_loss = total_loss / len(train_dataset)
     accuracy = correct / len(train_dataset)
```

```
    # Validation
    model.eval()
    val_loss = 0.0
    val_correct = 0
    with torch.no_grad():
        for val_x, val_y in val_loader:
            val_x = val_x.to(torch.int32).to(device)
            val_y = val_y.type(torch.LongTensor).to(device)

            # calculate the validation loss
            # calculate the validation accuracy

            outputs = model(val_x)
            loss = criterion(outputs, val_y.long())
            val_loss += loss.item()
            predicted = torch.argmax(outputs,dim = 1)
            val_correct += (predicted == val_y).sum().item()



    """
    End Task 1 2/3
    """
    val_loss /= len(val_dataset)
    val_accuracy = val_correct / len(val_dataset)

    print(f"Epoch [{epoch+1}/{epoch_num}] "
          f"Train Loss: {avg_loss:.4f}, Train Acc: {accuracy:.4f} | "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_accuracy:.4f}")
```

The training result can be seen as below:

```
Epoch [1/5] Train Loss: 0.0057, Train Acc: 0.5966 | Val Loss: 0.0047, Val Acc:
0.6360
Epoch [2/5] Train Loss: 0.0040, Train Acc: 0.6932 | Val Loss: 0.0042, Val Acc:
0.6701
Epoch [3/5] Train Loss: 0.0036, Train Acc: 0.7198 | Val Loss: 0.0041, Val Acc:
0.6820
Epoch [4/5] Train Loss: 0.0033, Train Acc: 0.7380 | Val Loss: 0.0040, Val Acc:
0.6868
Epoch [5/5] Train Loss: 0.0032, Train Acc: 0.7508 | Val Loss: 0.0040, Val Acc:
0.6890
```

Next we complete the eval() function by computing the loss and accuracy on the test set

```
def eval(model, test_dataset, batch_size=1, device="cpu"):
  model.eval()
```

```python
    model.to(device)
    test_loss = 0.0
    test_correct = 0

    criterion = nn.CrossEntropyLoss()
    test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

    with torch.no_grad():
        for test_x, test_y in test_loader:
            test_x = test_x.to(torch.int32).to(device)
            test_y = test_y.type(torch.LongTensor).to(device)

            """
            Task 1 3/3

            Begin
            """

            # calculate the test loss
            # calculate the test accuracy
            outputs = model(test_x)
            loss = criterion(outputs, test_y)
            test_loss += loss.item()
            predicted = torch.argmax(outputs,dim = 1)
            test_correct += (predicted == test_y).sum().item()

            """
            End Task 1 3/3
            """


    test_loss /= len(test_dataset)
    test_accuracy = test_correct / len(test_dataset)

    print(f"Overall Loss: {test_loss:.4f}, Acc: {test_accuracy:.4f}")
```

```
Overall Loss: 0.0096, Acc: 0.7028
```

Since multiclass classification is a challenging task, 70% is quite acceptable but minority classes should be delved into inorder to see enhancements a confusion matrix will aid in finding the challenging tags that drop the accuracy.

Task 2:

The predict() function is completed to calculate the raw probabilities on the test set.

```
"""
Task 2 1/2

Begin
"""
# Generate predictions for the test data
def predict(model, test_dataset, batch_size=100, device="cpu"):
   model.eval()
   model.to(device)

   test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

   all_predictions = []
   with torch.no_grad():
       for test_x, test_y in test_loader:
           test_x = test_x.to(torch.int32).to(device)

           outputs = model(test_x)
           all_predictions.append(outputs)


   #stack all batch outputs into a tensor
   output = torch.cat(all_predictions, dim=0)
   return output.detach().cpu().numpy()

label_pred = predict(model, test_dataset, 100, device)
```

From creating the confusion matrix, it can be seen that the model is performing well in predicting more frequent tags but it has a very poor accuracy for less frequent ones

```
Class '%' → Accuracy: 94.86% (830/875)
Class '^2' → Accuracy: 53.58% (299/558)
Class '^q' → Accuracy: 29.55% (65/220)
Class 'aa' → Accuracy: 3.70% (5/135)
Class 'aap_am' → Accuracy: 6.80% (14/206)
Class 'ad' → Accuracy: 27.10% (781/2882)
Class 'b' → Accuracy: 30.20% (45/149)
Class 'ba' → Accuracy: 76.35% (397/520)
Class 'bd' → Accuracy: 19.05% (4/21)
Class 'bh' → Accuracy: 5.56% (1/18)
Class 'bk' → Accuracy: 73.68% (767/1041)
Class 'fa' → Accuracy: 93.35% (323/346)
```

```
Class 'fc' → Accuracy: 71.30% (3352/4701)
Class 'fo_o_fw_"_by_bc' → Accuracy: 22.31% (56/251)
Class 'ft' → Accuracy: 64.67% (227/351)
Class 'h' → Accuracy: 85.08% (15380/18078)
Class 'ng' → Accuracy: 1.94% (3/155)
Class 'nn' → Accuracy: 1.19% (4/335)
Class 'no' → Accuracy: 28.30% (15/53)
Class 'ny' → Accuracy: 1.10% (2/181)
Class 'qh' → Accuracy: 76.94% (2823/3669)
Class 'qo' → Accuracy: 70.42% (862/1224)
Class 'qw^d' → Accuracy: 45.45% (30/66)
Class 'qy' → Accuracy: 2.48% (20/805)
Class 'qy^d' → Accuracy: 23.32% (87/373)
Class 'sv' → Accuracy: 36.36% (32/88)
Class 't1' → Accuracy: 68.73% (178/259)
Class 't3' → Accuracy: 94.81% (8843/9327)
Class 'x' → Accuracy: 48.82% (3841/7867)
```

The accuracy for the ‘br’ class is about 36% and ‘bf’ tag is 0:

```
br accuracy: 0.36363636363636365
bf accuracy: 0.0
```

To solve this problem we are training the model with balanced weights (10 epoch were not enough to finetune the new weights a the accuracy was drifting quite low, so the number of epochs were increased to 20 which was the optimal point of increasing validation accuracy without overfitting on the training set, after that point the validation accuracy wasn’t improving significantly):

```
model_balanced = BiLSTMModel(
    vocab_size=VOCAB_SIZE,
    embed_size=EMBED_SIZE,
    hidden_size=HIDDEN_SIZE,
    max_length=MAX_LENGTH
)
train(model_balanced, train_dataset, val_dataset, 20, 1e-4, 256,
device, d_class_weights)
```

```
Epoch [1/20] Train Loss: 0.0116, Train Acc: 0.2899 | Val Loss: 0.0102, Val Acc:
0.3741
Epoch [2/20] Train Loss: 0.0084, Train Acc: 0.3655 | Val Loss: 0.0090, Val Acc:
0.3420
Epoch [3/20] Train Loss: 0.0069, Train Acc: 0.3861 | Val Loss: 0.0088, Val Acc:
0.3624
Epoch [4/20] Train Loss: 0.0061, Train Acc: 0.4078 | Val Loss: 0.0088, Val Acc:
0.3588
Epoch [5/20] Train Loss: 0.0054, Train Acc: 0.4169 | Val Loss: 0.0087, Val Acc:
0.4040
Epoch [6/20] Train Loss: 0.0048, Train Acc: 0.4361 | Val Loss: 0.0089, Val Acc:
0.3817
Epoch [7/20] Train Loss: 0.0043, Train Acc: 0.4468 | Val Loss: 0.0091, Val Acc:
0.3300
```

```
Epoch [8/20] Train Loss: 0.0040, Train Acc: 0.4667 | Val Loss: 0.0093, Val Acc:
0.4125
Epoch [9/20] Train Loss: 0.0036, Train Acc: 0.4870 | Val Loss: 0.0098, Val Acc:
0.3921
Epoch [10/20] Train Loss: 0.0033, Train Acc: 0.5011 | Val Loss: 0.0104, Val Acc:
0.4315
Epoch [11/20] Train Loss: 0.0031, Train Acc: 0.5142 | Val Loss: 0.0104, Val Acc:
0.3902
Epoch [12/20] Train Loss: 0.0028, Train Acc: 0.5303 | Val Loss: 0.0108, Val Acc:
0.4650
Epoch [13/20] Train Loss: 0.0027, Train Acc: 0.5493 | Val Loss: 0.0110, Val Acc:
0.4629
Epoch [14/20] Train Loss: 0.0025, Train Acc: 0.5680 | Val Loss: 0.0114, Val Acc:
0.4446
Epoch [15/20] Train Loss: 0.0023, Train Acc: 0.5785 | Val Loss: 0.0115, Val Acc:
0.4364
Epoch [16/20] Train Loss: 0.0022, Train Acc: 0.5935 | Val Loss: 0.0119, Val Acc:
0.4739
Epoch [17/20] Train Loss: 0.0021, Train Acc: 0.6041 | Val Loss: 0.0125, Val Acc:
0.4926
Epoch [18/20] Train Loss: 0.0021, Train Acc: 0.6126 | Val Loss: 0.0126, Val Acc:
0.4992
Epoch [19/20] Train Loss: 0.0020, Train Acc: 0.6213 | Val Loss: 0.0125, Val Acc:
0.4949
Epoch [20/20] Train Loss: 0.0019, Train Acc: 0.6389 | Val Loss: 0.0133, Val Acc:
0.4738
```

This model takes longer to reach good accuracy because frequent tags dominate other infrequent ones like ‘br’ or ’bf’ and the model needs more time to learn meaningful representations for less frequent tags because it has seen less examples.

The overall accuracy is:

```
Overall Loss: 0.0159, Acc: 0.4953
```

This accuracy is lower than the initial model but performs better than the unweighted model in predicting infrequent tags. It is true that the unweighted model got better results just by predicting frequent tags more often but this model is more generalizable and we can be sure that it is not avoiding infrequent tags in total. This can be concluded by looking at the accuracy of ‘br’ and ‘bf’ tags prediction which has significantly increased.

```
br accuracy: 0.4318181818181818
bf accuracy: 0.1520912547528517
```

Task 3:

The forward function has been completed based on the description. The CNN and maxpool extracts local features from each utterance, Dense_1 projects CNN features to embedding size, BiLSTM models context from previous and future utterances, Dense_2 compresses the final context vector and the final layer pedicts the dialogue act label.

```
    def forward(self, x):
     """
       x shape: [batch_size, MAX_LENGTH]
       """
       # 1) Embedding
       x = self.embedding(x)                 # [batch_size, MAX_LENGTH, EMBED_SIZE]
       x = x.unsqueeze(1)                    # [batch_size, 1, MAX_LENGTH,
EMBED_SIZE] so we can apply Conv2D
```

```
        # 2) Parallel Convolutions -> BN -> ReLU -> MaxPool
        # Each conv_block expects input of shape [batch_size, 1, MAX_LENGTH,
EMBED_SIZE].
        pooled_outputs = []
        for conv in self.conv_blocks:
            conv_out = conv(x)  # shape: [batch_size, num_filters, (MAX_LENGTH - fs
+ 1), 1]
            # Perform 2D max pooling across the variable spatial dimension
conv_out.shape[2] and width=1
            # This collapses each feature map to a single value per filter.
            pooled = F.max_pool2d(conv_out, kernel_size=(conv_out.shape[2],
conv_out.shape[3]))
            # Now shape: [batch_size, num_filters, 1, 1]
            pooled_outputs.append(pooled)
        """
        Task 3 1/2

        Begin
        """
        # Concatenate along the channel dimension (num_filters dimension)
        # After pooling, each block has shape [batch_size, num_filters, 1, 1].
        # concatenation => [batch_size, len(filter_sizes)*num_filters, 1, 1]
        x_cat = torch.cat(pooled_outputs, dim=1)
        # 3) Flatten to [batch_size, 1, len(filter_sizes)*num_filters]
        x_cat = x_cat.view(x_cat.size(0), 1, -1)
        # 4) Dense_1 + Dropout
        cnn_repr = self.dropout_1(self.dense_1(x_cat))
        # 5) BiLSTM_1 (return_sequences=True)
        #    Input shape: [batch_size, sequence_len, EMBED_SIZE]
        #    Output shape: [batch_size, sequence_len, 2*EMBED_SIZE]
        out, _ = self.blstm1(cnn_repr)
        # 6) BiLSTM_2 (no return_sequences => final hidden state)
        #    Input shape: [batch_size, sequence_len, 2*EMBED_SIZE]
        #    final_timestep shape: [2, batch_size, EMBED_SIZE]
        out2, (h_n, c_n) = self.blstm2(out)

        #getting final hidden states and concatenating them
        blstm_repr = torch.cat((h_n[0], h_n[1]), dim=1)
        # 7) Dense_2 + Dropout

        blstm_out = self.dropout_2(self.dense_2(blstm_repr))

        # 8) Concatenate Flatten(dropout_1) and dropout_2
        #    Flatten(dropout_1) => shape [batch_size, EMBED_SIZE]
        #    out => [batch_size, EMBED_SIZE]
```

```
        # Note: dropout_1 was [batch_size, 1, EMBED_SIZE], so flatten the second
dimension:

        cnn_flat = cnn_repr.squeeze(1)
        combined = torch.cat((cnn_flat, blstm_out), dim=1)
        # 9) Output layer
        logits = self.output(combined)

        """
        End Task 3 1/2
        """
        return logits
```

Resulting in this model:

```
CNN_BiLSTM(
  (embedding): Embedding(43732, 100)
  (conv_blocks): ModuleList(
    (0): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 100), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): Conv2d(1, 64, kernel_size=(4, 100), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): Conv2d(1, 64, kernel_size=(5, 100), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
  )
  (dense_1): Linear(in_features=192, out_features=100, bias=True)
  (dropout_1): Dropout(p=0.2, inplace=False)
  (blstm1): LSTM(100, 100, batch_first=True, bidirectional=True)
  (blstm2): LSTM(200, 100, batch_first=True, bidirectional=True)
  (dense_2): Linear(in_features=200, out_features=100, bias=True)
  (dropout_2): Dropout(p=0.2, inplace=False)
  (output): Linear(in_features=200, out_features=43, bias=True)
)
```

The training results were as follows:

```
Epoch [1/5] Train Loss: 0.0043, Train Acc: 0.6677 | Val Loss: 0.0043,
Val Acc: 0.6584
Epoch [2/5] Train Loss: 0.0038, Train Acc: 0.7019 | Val Loss: 0.0041,
Val Acc: 0.6699
Epoch [3/5] Train Loss: 0.0035, Train Acc: 0.7199 | Val Loss: 0.0040,
Val Acc: 0.6752
Epoch [4/5] Train Loss: 0.0032, Train Acc: 0.7354 | Val Loss: 0.0041,
Val Acc: 0.6731
```

```
Epoch [5/5] Train Loss: 0.0031, Train Acc: 0.7486 | Val Loss: 0.0040, Val Acc: 0.6822
```

The overall accuracy seems to better than the weighted model while taking the contest into consideration

```
Overall Loss: 0.0159, Acc: 0.4953
```

To inspect the affect on minority classes, we again see the accuracy in predicting the tags ‘br’ and ‘bf’, ‘br’ has increased significantly but ‘bf’ got lower suggesting that modelling sequences with this tag is still hard even with context

```
br accuracy: 0.38636363636363635
bf accuracy: 0.0
```

The accuracy in all classes can be seen below. The model is now not just dependant on the frequency and can generalize well in some infrequent cases while keeping a god prediction with the frequent ones.

```
Class 'na' → Accuracy: 0.00% (0/197)
Class 'qy^d' → Accuracy: 6.27% (21/335)
Class '%' → Accuracy: 73.21% (2686/3669)
Class 'bd' → Accuracy: 28.57% (6/21)
Class 'ng' → Accuracy: 7.87% (7/89)
Class 'oo_co_cc' → Accuracy: 21.74% (5/23)
Class 'bk' → Accuracy: 42.09% (157/373)
Class 'ad' → Accuracy: 14.56% (30/206)
Class '^q' → Accuracy: 1.23% (3/244)
Class 't3' → Accuracy: 12.50% (2/16)
Class 't1' → Accuracy: 33.33% (6/18)
Class 'bf' → Accuracy: 0.00% (0/263)
Class 'b^m' → Accuracy: 1.10% (2/181)
Class 'qw' → Accuracy: 74.42% (387/520)
Class 'arp_nd' → Accuracy: 0.00% (0/47)
Class 'qrr' → Accuracy: 71.70% (38/53)
Class 'fp' → Accuracy: 40.91% (27/66)
Class 'no' → Accuracy: 1.32% (1/76)
Class 'ft' → Accuracy: 21.74% (5/23)
Class 'qy' → Accuracy: 69.61% (852/1224)
Class 'x' → Accuracy: 93.94% (822/875)
Class '^2' → Accuracy: 6.45% (10/155)
Class 'aap_am' → Accuracy: 0.00% (0/29)
Class 'b' → Accuracy: 93.66% (8736/9327)
Class 'fc' → Accuracy: 51.08% (285/558)
Class 'fo_o_fw_"_by_bc' → Accuracy: 7.17% (18/251)
Class 'qh' → Accuracy: 8.15% (11/135)
Class 'qo' → Accuracy: 43.18% (95/220)
Class 'h' → Accuracy: 62.96% (221/351)
Class '^g' → Accuracy: 28.57% (6/21)
Class 'nn' → Accuracy: 94.80% (328/346)
Class '^h' → Accuracy: 33.56% (50/149)
Class 'sd' → Accuracy: 88.01% (15910/18078)
Class '+' → Accuracy: 66.65% (3133/4701)
Class 'fa' → Accuracy: 23.08% (3/13)
```

```
Class 'ny' → Accuracy: 16.65% (134/805)
Class 'sv' → Accuracy: 40.37% (3176/7867)
Class 'br' → Accuracy: 38.64% (34/88)
Class 'qw^d' → Accuracy: 0.00% (0/21)
Class 'ba' → Accuracy: 69.26% (721/1041)
Class 'aa' → Accuracy: 25.85% (745/2882)
Class 'ar' → Accuracy: 0.00% (0/86)
Class 'bh' → Accuracy: 65.64% (170/259)
```

To deeply investigate the effect of context, we can see some 'br' samples that were falsely classified by the weight BiLSTM but the contextual model was able to classify the accurately:

```
Fixed by Context Model:
 Sentence: Excuse me. /
 True Label:        br
 Base Predicted:   ad
 Context Predicted: br

Fixed by Context Model:
 Sentence: Pardon me. /
 True Label:        br
 Base Predicted:   qy^d
 Context Predicted: br

Fixed by Context Model:
 Sentence: <Throat_clearing> The ones that what? /
 True Label:        br
 Base Predicted:   qy^d
 Context Predicted: br
```

These utterances are more ambiguous by themselves even if we assign weights to their tags, they are better understood with prior context which is provided by the BiLSTM layer.

PART D: Dialogue 2: A Conversational Dialogue System

Task 1:

As described in the instructions, an embedding layer was defined to load the GLoVE embeddings, the embeddings were passed to a dropout layer to help avoid overfitting and then to a bidirectional GRU unit that reads the sentence in both directions and will capture more semantic meaning as it reads the sentence from beginning to end and end to beginning so it has information about previous and posterior tokens.

```
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_size=batch_size,
dropout=0.2):
        """
        Initializes the Encoder.

        Args:
            vocab_size (int): Size of the vocabulary.
            embedding_dim (int): Dimension of the embedding vectors.
            enc_units (int): Number of units in the GRU.
```

```
            batch_size (int): Size of each batch.
            embeddings (torch.Tensor, optional): Pre-trained embedding weights.
Defaults to None.
            dropout (float, optional): Dropout probability. Defaults to 0.2.
        """
        super(Encoder, self).__init__()
        self.batch_size = batch_size
        self.enc_units = enc_units

        """
        Task 1: Implementing the encoder

        Begin
        """
        #  pass the embedding into a bidirectional version of the GRU

        self.embeddings = make_embedding_layer(embedding_dim)
        self.dropout = nn.Dropout(dropout)
        self.gru=
nn.GRU(input_size=embedding_dim,hidden_size=enc_units,num_layers=1,
batch_first=True, bidirectional=True)


        """
        End Task 1
        """
```

Task 2:

To define the __init__() method we use a Bahdanau attention layer to utilize attention and avoid forgetting the beginning of the sequence for long sequences using the encoder's output and decoder's current hidden state. We also have two GRU layers to understand the context and a fully connected layer in the end to produce the output. We have one dropout layer before passing the embedding to the GRU and one after the first GRU layer

```
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_size=batch_size):
        super(Decoder, self).__init__()
        self.batch_size = batch_size
        self.embedding = embeddings
        self.units = 2 * dec_units  # Assuming bidirectional encoder
        self.fc = nn.Linear(self.units, vocab_len)

        """
        Task 2: Implementing the decoder with attention

        Begin
        """
```

```
        self.embedding_dropout = nn.Dropout(0.2)
        self.attention = BahdanauAttention(self.units)
        self.decoder_gru_l1 = nn.GRU(input_size=embedding_dim + self.units,
hidden_size = self.units, batch_first=True)
        self.decoder_gru_l2 = nn.GRU(input_size = self.units, hidden_size =
self.units,batch_first=True)
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(self.units,vocab_size)

        self.fc = nn.Linear(self.units,vocab_size)



        """
        End Task 2
        """
```

During the overall training through different epochs, it was observed that greedy answers had more of a sentence structure and beam answers were mostly highly probable words like yes, it, no, I. greedy answers were also just a repetition of the same sentence “i am not” until the training epochs proceeded further and the answers started getting more context related and accurate like “what are you doing?” -> “i am going to do”.

In addition the epoch with the lowest loss isn’t producing the most contextualy accurate answers because we’re using a cross entropy loos which only does a token based evaluation. For instance, this is the results for the best model:

```
load best checkpoint from ep: 100
####################
Greedy| Q: Hello   A: i am sorry

Beam  5 |  Hello   A: what
%
Greedy| Q: How are you ?  A: i am not going to do
Beam  5 |  How are you ?  A: i
%
Greedy| Q: What are you doing ?  A: i am not
Beam  5 |  What are you doing ?  A: nothing
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do
Beam  5 |  What is your favorite restaurant ?  A: what
%
Greedy| Q: Do you want to go out ?  A: i am not
Beam  5 |  Do you want to go out ?  A: yes
```

However a previous epoch with a higher loss has more coherent answers:

```
*** Epoch 98 Loss 3.6989 ***

####################
Greedy| Q: Hello   A: hi
Beam  5 |  Hello   A: yes
%
Greedy| Q: How are you ?  A: i am not going to
Beam  5 |  How are you ?  A: it
```

```
%
Greedy| Q: What are you doing ?  A: i am not
Beam  5 |  What are you doing ?  A: it
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do it
Beam  5 |  What is your favorite restaurant ?  A: it
%
Greedy| Q: Do you want to go out ?  A: i am not
Beam  5 |  Do you want to go out ?  A: i
```

Task 3:

Let's look at the attention weights and compare them after 5, 50 and 100 epochs. Instead of evaluating by an automatic evaluation method, you can show us 10 predictions for each model. Answer the following questions based on your predictions, giving examples and/or explaining the evidence for your answers.

The initial training code was configured in a way to save the best result a each end point so with mild changes, the model was retrained to save the weights at epoch 5,50 and 100
The weights and matrixes for each answer in each epoch can be found in the jupyter notebook.

	hi	How are you	What are you doing	What is your favorite restaurant	How is the weather	Do you want to go out	Can you help me	Where is the book	Do you have a problem	What is your name
Epoc 5	hi	I am not	I am not	I am not going to be	I am not going to be a long time	I am not going to do	I am sorry	I am sorry	I am not you (sassy :D)	I am not
Epoch 50	hi	I am sorry	I am sorry	I am not	I am not going to do	I am not going to do	I am sorry	I am not going to be	I am not going to do	I am not going to be a good idea
Epoch 100	hi	I am not going to do	I am not	I am not	I am not going to	I am not going to do	I am not going to	I am sorry	I am sorry	I am you

* Did the models learn to track local relations between words?
Yes, and this increased as the epochs proceeded. In early epochs (epoch 5), attention is spread broadly across input tokens, and responses such as "i am not" and “I am sorry” are generic and disconnected. By epochs 50 and 100, attention begins to focus on content words like "what", "name", "problem", showing that the decoder is learning to align input tokens with meaningful output tokens. Specifically for a complicated question like “what is your name” getting the “ i am you” answer at epoch 100 shows that the decoder learnt to pay attention to “name” because at epoch 5, the answer was “i am not”.

* Did the models attend to the least frequent tokens in an utterance? Can you see signs of overfitting in models that hang on to the least frequent words?
Yes, but not in an aggressive way. At later epochs, the outputs to some questions repeat "i am not going to do" regardless of the questions. This could indicate memorization rather than generalization which is equivalent to overfitting. However, attention is still strong on rare tokens like "name" and "book", which is useful

* Did the models learn to track some major syntactic relations in the utterances (e.g. subject-verb, verb-object)?
In epoch 5, the answers don't really follow a grammatical structure ("I am not") however after epoch 50, we observe subject–verb formations: "i am", "i am sorry", "i am not going to...". However, verb-object relations aren't strong enough to form very well structured sentences. For example, "i am not going to do" is a grammatically wrong sentence but "i am sorry" asn an answer to "can you help me" may not be directly linked to syntax, but at least it is grammatically correct.

* Do they learn to encode some other linguistic features? Do they capture part-of-speech tags (POS tags)?
The model understands basic POS structureslike "i am not" or "i am sorry". However, it mainly answers with generic answers and does not explicitly distinguish POS in a specific way.
* What is the effect of more training on the length of response?
Responses get noticeably longer. At epoch 5, many responses are short ("i am", "i am not"). By epoch 50, longer responses like "i am sorry" and "i am not going to be a good idea" appear. They're mostly not accurate but definitely more informative and closer to a real answer than the earlier epochs.

* In some instances, by the time the decoder has to generate the beginning of a response, it may already forget the most relevant early query tokens. Can you suggest ways to change the training pipeline to make it easier for the model to remember the beginning of the query when it starts to generate the response?
Some ideas could be using multi-head attention like in transformers to model a bigger chain of dependencies. Also using the token's positions could also be beneficial.

PART E: Prompt Engineering

Task 1 (Natural Language Generation):

Prompt 1: Give a recipe for spaghetti al pomodoro
Response 1:

```
Certainly! Here is a simple recipe for spaghetti al pomodoro that serves 4-6
people:

Ingredients:

* 12 oz (340 g) spaghetti
* 3 large tomatoes, diced
```

* 2 cloves garlic, minced
* 2 tbsp extra virgin olive oil
* Salt and freshly ground black pepper
* Fresh basil leaves, chopped (optional)

Instructions:

1. Bring a large pot of salted water to a boil. Cook the spaghetti according to package instructions until al dente. Reserve 1 cup of pasta cooking water and then drain the spaghetti.
2. In a blender or food processor, combine the tomatoes, garlic, olive oil, salt, and black pepper. Blend until smooth.
3. Add the reserved pasta cooking water to the blended mixture and blend again until you have a thick but still pourable sauce.
4. Add the spaghetti back into the pot with the tomato sauce and toss everything together, ensuring

Prompt 2: Give a detailed recipe for a classic spaghetti al pomodoro for two people

Response 2:

Sure thing! Here's a recipe for a delicious and authentic Italian spaghetti al pomodoro, perfect for two people. This dish is made with fresh tomatoes, garlic, olive oil, basil, salt, and of course, spaghetti. Let me know if you have any questions or need further clarification!

Ingredients:

* 12 oz (340g) spaghetti
* 2 large tomatoes, peeled and diced
* 3 cloves garlic, minced
* 3 tbsp (45ml) extra-virgin olive oil
* 1/4 cup (60ml) fresh basil leaves, chopped
* Salt to taste
* Grated Parmesan cheese for serving (optional)

Instructions:

Step 1 - Cook the spaghetti

1. Bring a large pot of salted water to a boil.
2. Add the spaghetti and cook according to package instructions until al dente. Reserve 1 cup of pasta water before draining.

Step 2 - Make the tom

Prompt 3: Give a recipe for a classic spaghetti al pomodoro like my nonna made

Response 3:

Certo! Here's a traditional Italian recipe for spaghetti all'alpomodoro, just like Nonna used to make. This dish is a simple yet flavorful combination of fresh tomatoes, garlic, basil, and olive oil that will transport your taste buds straight to the rolling hills of Tuscany!

Ingredients:

* 12 oz (340g) spaghetti
* 2 large ripe tomatoes, peeled and chopped
* 3 cloves garlic, minced
* 1/4 cup extra-virgin olive oil
* Salt to taste
* Fresh basil leaves, chopped (optional)

```
Instructions:

1. Bring a large pot of salted water to a boil and cook the spaghetti according to package instructions until al dente. Reserve 1 cup of pasta water before draining the spaghetti.
2. In a blender or food processor, combine the tomatoes, garlic, olive oil, salt, and a pinch of black pepper. Blend until smooth and creamy, stopping to scra
```

It is observed that the tone of the prompt is taken into consideration with the response but the first ingredient list which was intended for 4-6 servings, was returned when specifically requested for a 2 serving recipe. Prompt 2 and 3 are more detailed, as they mention to peel and dice the tomatoes but that is not included in prompt 1. Also optional parmesan is only mentioned in prompt 2. No evidence of hallucination is observed within the 256 token limit.

Task 2 (Summarization):

Prompt 1: 'can you provide me with a summary of the following article?' + task2_article

Response 1:

```
Sure! Here is a summary of the article you provided:

The LLaMA (Large Language Model Meta AI) family of autoregressive large language models was released by Meta AI in February 2023. The first version included four model sizes, with the 13B parameter model exceeding the performance of GPT-3 and competing with state-of-the-art models like PaLM and Chinchilla. Unlike other powerful LLMs, LLaMA's weights were released to the research community under a noncommercial license. In July 2023, Meta released several models as Llama 2, including three model sizes with increased training data. The architecture of LLaMA and LLaMA-2 is based on the transformer architecture but has minor differences compared to GPT-3. The foundational models were trained on a large dataset of publicly available text sources, while the fine-tuned chat models were additionally trained on prompt-response pairs created for this project.
```

Prompt 2: 'can you provide me with an abstractive summary of the following article in 3-5 sentences?' + task2_article

Response 2:

```
 Here is an abstractive summary of the article in 5 sentences:

LLaMA is a family of large language models released by Meta AI that have exceeded the performance of previous models such as GPT-3. The models are available under a noncommercial license and were leaked to the public within a week of their release. LLaMA-2, the next generation of LLaMA, was announced in partnership with Microsoft and includes three model sizes with 7, 13, and 70 billion parameters. The models are free for many commercial use cases but there are some remaining restrictions due to which the description of LLaMA as open source has been disputed. LLaMA uses transformer architecture and its developers focused on scaling the model's performance by increasing the volume of training data rather than the number of parameters.
```

Prompt 3: 'can you provide me with an extractive summary of the following article in 3-5 sentences?' + task2_article

Response 3:

```
Here is an extractive summary of the article LLaMA in 3-5 sentences:

LLaMA is a family of autoregressive large language models (LLMs) released by Meta AI starting in February 2023. The model's performance on most NLP benchmarks exceeded that of GPT-3, and the largest model was competitive with state-of-the-art models such as PaLM and Chinchilla. LLaMA is trained using a noncommercial license, and its weights were leaked to the public via BitTorrent within a week of release.
```

```
Meta released several models in July 2023 as Llama 2, including three model sizes:
7, 13, and 70 billion parameters.
```

Given the 3 different prompts the model came up with 3 different summaries, since the first prompt was quite generic, it didn't capture the point of the text and can't be classified as a very informative summary. However, the extractive and abstractive summary in 3-5 sentences seem to be able to carry the article's message and the point of attractiveness was followed since all the sentences are picked from the article itself and they're sentences with high information gains.

Task 3 (Machine Translation):

Prompt: Can you translate this article to Persian

Response:

```
The article you provided is not available in Persian, but I can certainly help you
translate it! Here's the translation of the article to Persian:

مقاله عنوان: لاما (LLaMA) - یک خانواده از MODEL های زبانی برای سایت های Meta

لاما (LLaMA) is a family of autoregressive language models released by Meta AI
starting in February 2023. The developers of LLaMA reported that the model's
performance on most NLP benchmarks exceeded that of the much larger GPT-3 and was
competitive with state-of-the-art models such as PaLM and Chinchilla. Unlike other
powerful language models, which have been available only through limited APIs, Meta
released LLaMA's model weights to the research community under a noncommercial
license. Within a week of LLaMA's release, its weights were leaked to the public on
4chan via BitTorrent.

In July
```

Since we are doing zero-shot learning, the model probably forgot that the task was translation after a sentence and switched back to English. The Persian translation doesn't follow a semantic or syntactic order and does not result in a meaningful sentence.

Task 4 (Named Entity Recognition):

One-hot prompt: 'extract named entities in this sentence and tag them with ORG if organization, DATE if date, NUM if number and MODEL if AI model and format the result in a json format' + task4_sentence + "you van use this example: sentence = The first version of DALL-E was announced in January 2021 by OpenAI and has had two other releases , output: [{'entity':DALL-E, 'label': MODEL},{'entity': January 2021, 'label': DATE},{'entity': OPenAI, 'label':ORG},{'entity':2,'label':NUM}]"

One-shot response:

```
ASSISTANT:
Hi there! I'd be happy to help you extract named entities from your sentence. Here
are the extracted entities and their corresponding labels in JSON format:

{
"entities": [
{
"entity": "LLaMA-2",
"label": "MODEL"
},
{
"entity": "July 18, 2023",
"label": "DATE"
},
```

```
{
"entity": "Microsoft",
"label": "ORG"
},
{
"entity": "7",
"label": "NUM"
}
]
}
```

The one shot learning missed recognizing ‘Meta’ as an ORG, ‘LLaMA’ as a MODEL, ‘three’ as a NUM , ‘7,13, and 70 billion’ as NUM but succeeded in recognizing all other entities..

Few shot learning prompt: 'extreact named entities in this sentence and tag them with ORG if organization, DATE if date, NUM if number and MODEL if AI model and format the result in a json format' + task4_sentence + "you van use this example: sentence1 = The first version of DALL-E was announced in January 2021 by OpenAI and has had two other releases , output1: [{'entitiy':DALL-E, 'label': MODEL},{'entity': January 2021, 'label': DATE},{'entity': OPenAI, 'label':ORG},{'entity':2,'label':NUM}], sentence2= Anthropic developed Claude-2 in 2023 with 100k context length, output2 = [{'entitiy':Claude-2, 'label': MODEL},{'entity': 2023, 'label': DATE},{'entity': Anthropic, 'label':ORG},{'entity':100k,'label':NUM}], sentence3 = GPT-4 is a multimodal large language model trained by OpenAI and the fourth iterationof the generative pretrained transformer models in March 2023, output3 =[{'entitiy':GPT-4, 'label': MODEL},{'entity': March 2023, 'label': DATE},{'entity': OPenAI, 'label':ORG},{'entity':fourth,'label':NUM}]"

Few shot learning response:

```
ASSISTANT:
Hello! I'd be happy to help you with that. Here are the named entities in your sentences and their corresponding labels:

sentence1: [ {'entitiy': 'LLaMA-2', 'label': 'MODEL'}, {'entity': 'July 18, 2023', 'label': 'DATE'}, {'entity': 'Meta', 'label': 'ORG'}, {'entity': 'three', 'label': 'NUM'} ]
sentence2: [ {'entitiy': 'Claude-2', 'label': 'MODEL'}, {'entity': '2023', 'label': 'DATE'}, {'entity': 'Anthropic', 'label': 'ORG'}, {'entity': '100k', 'label': 'NUM'} ]
sentence3: [ {'entitiy': 'GPT-4', 'label': 'MODEL'}, {'entity': 'March 2023', 'label': 'DATE'}, {'entity': 'OpenAI', 'label': 'ORG'}, {'entity': 'fourth', 'label': 'NUM'} ]

Please note that in sentence1, the entity 'July
```

In the few-shot prompt, three examples were provided to enhance the model’s understanding of the task. However, the prompt accidentally framed the examples (sentence1, sentence2, sentence3) as if they were part of the input task, leading the model to wrongly output information for those example sentences rather than for the intended new input. This shows that however few-shot learning is expected to perform better, it requires careful prompt design as the query and examples can get confused.

Task 5 (Dialogue Act Tagging):

Input prompt:

```
f"""
I need help with dialogue act tagging, when given a conversation, each utterance
should be tagged with one of these relevant tags and the output should be returned
in JSON format:
'qestion' if it's a question
'request' if it's a request for sth
'greet' if it's a greeting
'accept' if it indicates acceptance or agreement
'state' if it provides a information about something
'answer' response to a question

for example the tags for these sequences can be defined like this:

Dialogue 1:
A: Hi
B: Hi
A: Where are you?
B: near to campus
A: coming to see you
Output1:
[
   {{"utterance": "Hi", "dialogue_act": "greet"}},
   {{"utterance": "Hi", "dialogue_act": "greet"}},
   {{"utterance": "Where are you?", "dialogue_act": "question"}},
   {{"utterance": "near to campus", "dialogue_act": "answer"}},
   {{"utterance": "coming to see you", "dialogue_act": "state"}}
]

Dialogue 1:
A: Hi
B: Hello
A: I need a bus ticket to Liverpool
B: when do you want to go?
A: tommorow before 12
B: alright, I am looking for tickets now.
Output1:
[
   {{"utterance": "Hi", "dialogue_act": "greet"}},
   {{"utterance": "Hello", "dialogue_act": "greet"}},
   {{"utterance": "I need a bus ticket to Liverpool?", "dialogue_act": "request"}},
   {{"utterance": "when do you want to go?", "dialogue_act": "question"}},
   {{"utterance": "tommorow before 12", "dialogue_act": "answer"}},
   {{"utterance": "alright, I am looking for tickets now.", "dialogue_act":
"statement"}}
```

```
]

based on these examples, i need you to tag this dialogue: {task5_conversation}

"""
```

Response:

```
ASSISTANT:b
Hi! Sure, I'd be happy to help you with that! Here are the dialogue acts for each utterance in your given conversation:

[
    {"utterance": "I need to travel in May.", "dialogue_act": "request"},
     {"utterance": "And, what day in May did you want to travel?", "dialogue_act": "question"},
     {"utterance": "OK uh I need to be there for a meeting that’s from the 12th to the 15th.", "dialogue_act": "answer"},
    {"utterance": "And you’re flying into what city?", "dialogue_act": "question"},
    {"utterance": "Seattle.", "dialogue_act": "answer"},
         {"utterance": "And what time would you like to leave Pittsburgh?", "dialogue_act": "question"},
       {"utterance": "Uh hmm I don’t think there’s many options for non-stop.", "dialogue_act": "state"}
]
```

The prompt was to tag each utterance in the conversation with one of six dialogue act labels: greet, question, request, accept, state, and answer, and to output the result in JSON format.
The tagging was consistent with the provided examples. The model’s recognition of tags showed that it understood the examples and applied them appropriately.
There were no major mistakes in the tagging. Some utterances could potentially be tagged differently, but given the definitions provided, the results were appropriate.
Overall, the tagging seems very accurate to the context provided and given the tags in the input prompt and it would be how a human tagger would also tag them.