ECS7001P - NEURAL NETWORKS AND NLP - 2021/22

ASSIGNMENT 2: PRE-TRAINED TRANSFORMERS, INFORMATION EXTRACTION AND DIALOGUE

MSc Big Data Science 2021-2022 (September)

Student Name: Shivani Gurung Rakesh Student ID: 210268155

Part A: Using Pre-trained BERT: -

Task 1: Data pre-processing: -

An example of our training data, which will be the same dev and test data.

```
SENTENCE         ASPECT          LABLE   ASPECT-START-INDEX      ASPECT-END-INDEX
['the decor is not special at all but their food and amazing prices make up for it.', 'decor', 'negative', '4', '9']
['the decor is not special at all but their food and amazing prices make up for it.', 'food', 'positive', '42', '46']
['the decor is not special at all but their food and amazing prices make up for it.', 'prices', 'positive', '59', '65']
['when tables opened up, the manager sat another party before us.', 'tables', 'neutral', '5', '11']
['when tables opened up, the manager sat another party before us.', 'manager', 'negative', '27', '34']
```

The information is divided into three categories: training, development, and testing. Each of these samples has a review, an aspect, a label (positive/negative), and the start and end indices for that aspect. The BERT tokenizer is used to tokenize the reviews and convert them to integer values. Tokenized sentence ids, tokenized sentence masks, and tokenized sentence segments are the outputs of the BERT Tokenizer. To prepare the input for tokenizing the reviews and aspects, we'll use BERT Tokenizer. Use the function create train dev test to convert the samples into BERT tokens (dataset). To begin, the tokenize function tokenizes each review and aspect and returns the associated ids, mask values, and segments. After that, the values are appended to the temp list in the wake of navigating the examples in general, these temp records are returned, and the outcomes are saved in x train survey int, x train audit covers, x train viewpoint int, and x train angle veils. The equivalent is exact for the Turn of events and Test Sets.

The code is as follows:

```
# Please write your code to generate the following data

#Define lists for savinf train,dev,test reviews, aspects and their int values

x_train_review_int =[]
x_train_review_masks =[]
x_train_aspect_int =[]
x_train_aspect_masks =[]

x_dev_review_int =[]
x_dev_review_masks =[]
x_dev_aspect_int =[]
x_dev_aspect_masks =[]

x_test_review_int =[]
x_test_review_masks =[]
x_test_aspect_int =[]
x_test_aspect_masks =[]

# Function to create train dev and test dataset
def create_train_dev_test(dataset):

    review_ids_list,review_masks_list,aspect_ids_list,aspect_masks_list = [],[],[],[] # temp lists to store review ,asptect and their int values

    for example in dataset:
        review_ids, review_masks, segments1 = tokenize(example[0], tokenizer) # Tokenize the review using tokenize function defined above
        aspect_ids, aspect_masks, segments2 = tokenize(example[1], tokenizer) #Tokonize the aspect

        #Append tokenized review, aspect and their ids to temp lists
        review_ids_list.append(review_ids)
        review_masks_list.append(review_masks)
        aspect_ids_list.append(aspect_ids)
        aspect_masks_list.append(aspect_masks)
```

```
    return review_ids_list, review_masks_list, aspect_ids_list, aspect_masks_list

#for Train Dataset
x_train_review_int, x_train_review_masks, x_train_aspect_int, x_train_aspect_masks = create_train_dev_test(train)

#For Dev Dataset
x_dev_review_int, x_dev_review_masks, x_dev_aspect_int, x_dev_aspect_masks = create_train_dev_test(dev)

#for test Dataset
x_test_review_int, x_test_review_masks, x_test_aspect_int, x_test_aspect_masks = create_train_dev_test(test)
```

Output:

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2269: FutureWarning: The `pad_to_max_length` argument is deprecated and will b
  FutureWarning,
[  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  1026 19802  1028  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]
[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 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]

[  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  1026 19802  1028  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]
[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 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]
```

Task 2: Basic classifiers using BERT: Model 1 and Model 2: -

Model1 has an aspect (128) input layer, an aspect (128) concealed symbolic layer, and a Distil BERT for Arrangement Order model that has been pretrained. The information token and covered token are taken care of into the BERT transformer layer. As measurements, the model is worked with Adam Analyzer, paired crossentropy misfortune capacity, and precision.

```
model.summary()

Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_token (InputLayer)       [(None, 128)]        0           []

 masked_token (InputLayer)      [(None, 128)]        0           []

 tf_distil_bert_for_sequence_cl  TFSequenceClassifie  66955779   ['input_token[0][0]',
 assification (TFDistilBertForS  rOutput(loss=None,               'masked_token[0][0]']
 equenceClassification)         logits=(None, 3),
                                 hidden_states=None
                                , attentions=None)

==================================================================================================
Total params: 66,955,779
Trainable params: 66,955,779
Non-trainable params: 0
__________________________________________________________________________________________________
```

Accuracy and Loss:

```
[ ]  results = model.evaluate([x_test_int_np,x_test_masks_np], y_test)
     print(results)

     42/42 [==============================] - 8s 106ms/step - loss: 0.8935 - accuracy: 0.7732
     [0.8935271501541138, 0.7732036113739014]
```

In sequence classification, the Model2 contains two info layers: input token and veiled token, which are shipped off a pre-prepared BERT transformer. The Bert transformer's result is shipped off a worldwide normal 1D pooling layer, which is trailed by a Thick layer with 16 secret cells and sigmoid initiation work. To yield the right names, a result layer with three result cells and a SoftMax initiation work is utilized.

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
input_token (InputLayer)     [(None, 128)]             0           []

masked_token (InputLayer)    [(None, 128)]             0           []

tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880   ['input_token[0][0]',
BertModel)                   ast_hidden_state=(N                 'masked_token[0][0]']
                             one, 128, 768),
                              hidden_states=None
                             , attentions=None)

global_average_pooling1d_maske  (None, 768)          0           ['tf_distil_bert_model[0][0]']
d (GlobalAveragePooling1DMaske
d)

dense (Dense)                (None, 16)                12304       ['global_average_pooling1d_masked
                                                                  [0][0]']

dense_1 (Dense)              (None, 3)                 51          ['dense[0][0]']

==================================================================================================
Total params: 66,375,235
Trainable params: 66,375,235
Non-trainable params: 0
```

Accuracy and Loss:

```
results = model2.evaluate([x_test_int_np,x_test_masks_np], y_test)
print(results)

42/42 [==============================] - 8s 108ms/step - loss: 0.4126 - accuracy: 0.8383
[0.41258203983306885, 0.838323354721693]
```

We can see that model 1 has accuracy of 77.32% while model 2 has accuracy of 83.83% henceforth we can say that model 2 performs better in comparison to 1. The justification for this is the additional worldwide normal pooling layer in Model2 which have tracked down more secret example/setting of surveys and words in the information hence helped in anticipating more right marks.

In Lab 4,

The accuracy achieved was 65.26%

In lab 4, we got a precision of 65.26% with higher misfortune. While the model 1 precision was superior to lab 4's, however there was more misfortune, the model 1 actually would do well to exactness. The normal pooling approach with DistilBert is generally better than utilizing a static word implanting technique alone.

Task 3: Advanced classifier using BERT: Model 3: -

```
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
input_token (InputLayer)     [(None, 128)]             0           []

masked_token (InputLayer)    [(None, 128)]             0           []

tf_distil_bert_model_1 (TFDist  TFBaseModelOutput(l  66362880   ['input_token[0][0]',
ilBertModel)                 ast_hidden_state=(N                 'masked_token[0][0]']
                             one, 128, 768),
                              hidden_states=None
                             , attentions=None)

lstm (LSTM)                  (None, 100)               347600      ['tf_distil_bert_model_1[0][0]']

dense_2 (Dense)              (None, 3)                 303         ['lstm[0][0]']

==================================================================================================
Total params: 66,710,783
Trainable params: 66,710,783
Non-trainable params: 0
```

```
[ ] results = model3.evaluate([x_test_int_np,x_test_masks_np], y_test)
    print(results)

    42/42 [==============================] - 10s 122ms/step - loss: 0.4912 - accuracy: 0.8271
    [0.4911702871322632, 0.8270958662033081]
```

As may be obvious, model 2 has the best presentation contrasted with model 1 and model 3 with 82.1% precision making it the best model. Which we can say is an intriguing case as after examining with my colleagues I discovered that the vast majority of their model 3 was awesome. In the event that we put more LSTM layers in model 3, there is a high opportunity model 3 will outflank model 2. Yet, according to this execution model 2 has beaded model 3, consequently turning into the best model

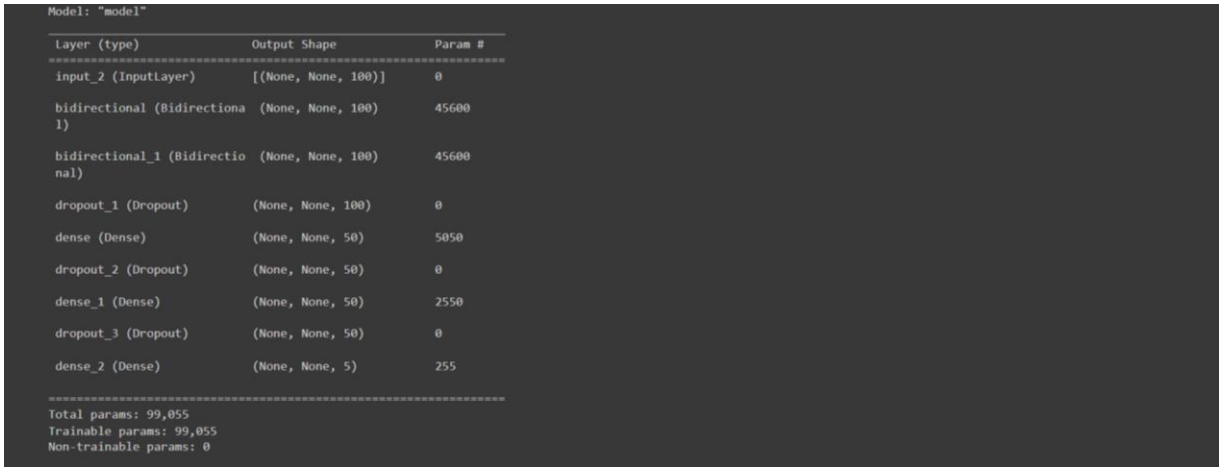
Part B - Information Extraction 1: Training a Named Entity Resolver: -

Task 1: Create a bidirectional GRU and Multi-layer FFNN: -

To develop the word embeddings, the models have an info layer and in this way a dropout with an embedding's dropout pace of 0.5. These word embeddings are then shipped off a bidirectional GRU layer with 50 units (i.e., 100 GRU cells) and a 0.2 intermittent dropout rate. The bidirectional GRU layer's statement yield is therefore shipped off a bidirectional GRU layer with tantamount arrangements. From that point onward, 3 dropouts and 2 thick layers with a dropout pace of 0.2 and 50 secret cells in thick layers are added to the organization. Since there are 5 closes to names, the result layer has 5 secret cells, and a SoftMax enactment work is used to appraise the probability of these marks for each word. The model is worked with Adam Analyzer and a meager straight out cross-entropy misfortune work, with precision as the measurement.

```
def build(self):
  word_embeddings = Input(shape=(None,self.embedding_size,))
  word_embeddings = Dropout(self.embedding_dropout_rate)(word_embeddings)
  """
  Task 1 Create a two layer Bidirectional GRU and Multi-layer FFNN to compute the ner scores for individual tokens
  The shape of the ner_scores is [batch_size, max_sentence_length, number_of_ner_labels]
  """
  #Bi-directional GRU1 having 50 GRU units i.e. 100 cells with recurrent dropout = 0.2 and return_sequences = True
  word_output = Bidirectional(GRU(50, return_sequences = True, recurrent_dropout = 0.2))(word_embeddings)
  #Bi-directional GRU2 having 50 GRU units i.e. 100 cells with recurrent dropout = 0.2 and return_sequences = True
  word_output = Bidirectional(GRU(50, return_sequences = True, recurrent_dropout = 0.2))(word_output)
  # Dropout layer with 0.2 as dropout rate
  dropout_1 = Dropout(self.hidden_dropout_rate)(word_output)
  # Dense layer with 50 neurons and ReLU Activation function
  dense = Dense(self.hidden_size, activation='relu')(dropout_1)
  # Dropout layer with 0.2 as dropout rate
  dropout_2 = Dropout(self.hidden_dropout_rate)(dense)
  # Dense layer with 50 neurons and ReLU Activation function
  dense_1 = Dense(self.hidden_size, activation='relu')(dropout_2)
  # Dropout layer with 0.2 as dropout rate
  dropout_3 = Dropout(self.hidden_dropout_rate)(dense_1)
  #Output Layer with 5 neurons and SoftMax Activation function
  ner_scores = Dense(5, activation ='softmax')(dropout_3)
  """
  End Task 1
  """
```

Models' summary,



```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, None, 100)]       0

bidirectional (Bidirectiona  (None, None, 100)         45600
l)

bidirectional_1 (Bidirectio  (None, None, 100)         45600
nal)

dropout_1 (Dropout)          (None, None, 100)         0

dense (Dense)                (None, None, 50)          5050

dropout_2 (Dropout)          (None, None, 50)          0

dense_1 (Dense)              (None, None, 50)          2550

dropout_3 (Dropout)          (None, None, 50)          0

dense_2 (Dense)              (None, None, 5)           255

=================================================================
Total params: 99,055
Trainable params: 99,055
Non-trainable params: 0
```

Task 2: Form the predicted named entities: -

For the Code at first the eval_fd_list is circled to get the word_embeddings, ner_labels_list, gold marks and sent_lens. After that expectation is done on the word_embeddings to get the forecasts the np.argmax is applied to the forecasts on pivot = 2. This gives us predicted_ner_labels for each expression of each sentence. After that a predicted_set() is characterized to store the accompanying data: (sid, begin, end, ner_label). Then, at that point, we identify over the predicted_ner_labels. This gives the sentence list and the sentence. After those two pointers are characterized as start = 0 and end = 0. This will check start and end file of ner marks. The worth 0 is added to all sent for 'O' mark. Once more, we have listed each sent. This gives us list worth and token for each expression of a penny. Then, at that point, we check if end! = token and end == 0-direct head toward current list. i.e., begin = j. In the event that misleading, we check if end! = 0 and end! = token on the off chance that valid, we add the (current list, begin, end - 1,

int(token)) to the predicted_set. Likewise, we set end = 0. Then, at that point, we actually take a look at whether the ongoing token! =0 or not. On the off chance that indeed, we guide head toward current file and end to current ner name. As we should contrast the anticipated named elements and the gold name substances to get the genuine positive, misleading positive and bogus negative. We want to compute genuine positive we observe the right paired examples utilizing python's 'convergence' work. We do this for all clump of sentences. For bogus negatives, we compute the quantity of gold.differences and for misleading positive we work out the length of predicted_set.difference(gold).

```python
def eval(self, eval_fd_list):
  tp, fn, fp = 0,0,0
  for word_embeddings, _, gold, sent_lens in eval_fd_list:
    predictions = self.model.predict_on_batch([word_embeddings])
    """
    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'
    """
    predicted_ner_labels = np.argmax(predictions, axis=2) # get the predicted ner labels
    predicted_set = set() # define a set
    for i, sent in enumerate(predicted_ner_labels):
      start = 0 # start pointer
      end = 0 # end pointer
      sent = np.append(sent, 0) # append sent
      for j, token in enumerate(sent):
        if end != token and end == 0: # check end index pointer doesnt point to token and pointing to 0 index
          start = j # point start pointer to token index
        elif end != 0 and end != token: # check end pointer doesnt point to start or token
          predicted_set.add((i, start, j - 1, int(end))) # add values to the predicted set
          end = 0 # set end pointer to 0
          if token != 0: # check if token points to 0 or not
            start = j # set start to token index
            end = token # point end to token
          continue
        end = token
    tp = tp + len(gold.intersection(predicted_set)) # calculate the true positive
    fp = fp + len(predicted_set.difference(gold)) # calculate the false positive
    fn = fn +len(gold.difference(predicted_set)) # calculate the false negative
    """
    End Task 2
    """
```

```
Starting training epoch 1/5
141/141 [==============================] - 75s 475ms/step - loss: 0.3660 - accuracy: 0.9370
Time used for epoch 1: 1 m 15 s
Evaluating on dev set after epoch 1/5:
F1 : 29.07%
Precision: 44.24%
Recall: 21.64%
Time used for evaluate on dev set: 0 m 5 s

Starting training epoch 2/5
141/141 [==============================] - 68s 484ms/step - loss: 0.1103 - accuracy: 0.9658
Time used for epoch 2: 1 m 8 s
Evaluating on dev set after epoch 2/5:
F1 : 63.17%
Precision: 64.13%
Recall: 62.23%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 3/5
141/141 [==============================] - 69s 490ms/step - loss: 0.0741 - accuracy: 0.9779
Time used for epoch 3: 1 m 9 s
Evaluating on dev set after epoch 3/5:
F1 : 71.31%
Precision: 71.04%
Recall: 71.58%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 4/5
141/141 [==============================] - 68s 484ms/step - loss: 0.0580 - accuracy: 0.9827
Time used for epoch 4: 1 m 8 s
Evaluating on dev set after epoch 4/5:
F1 : 77.18%
Precision: 80.72%
Recall: 73.93%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 5/5
141/141 [==============================] - 69s 492ms/step - loss: 0.0505 - accuracy: 0.9849
Time used for epoch 5: 1 m 9 s
Evaluating on dev set after epoch 5/5:
F1 : 78.62%
Precision: 81.19%
Recall: 76.20%
Time used for evaluate on dev set: 0 m 2 s

Training finished!
Time used for training: 6 m 4 s

Evaluating on test set:
F1 : 75.99%
Precision: 78.06%
Recall: 74.03%
Time used for evaluate on test set: 0 m 2 s
```

This is a healthy degree of accuracy. The two bidirectional GRU layers that fathomed the setting of the expressions and consequently appropriately conjecture ner marks could be the clarification.

For this test set, the f1 score is 75.99 percent, the precision is 78.06 percent, and the recall is 74.03 percent. In terms of accuracy, this is identical to our training set. As a result, we can say that our model is somewhat generic, but it still needs to be improved by adding more layers or training for longer epochs.

Part C - Information Extraction 2: A Coreference Resolver for Arabic: -

1. **Task 1:** Pre-processing (See section 4.4): -

The get data () function reads the json file first, then uses the preprocess arabic text function

to pre-process the data. The gold mentions, gold mention maps, cluster ids, and num mentions are obtained in the first job by passing clusters to the get mentions () function. Tensorize doc sentences is used in the following job to construct the padded document embeddings, which include a copy of the mention starts and ends indices that have been padded. For example, if the start index of the word "nlp" is 7 from the second sentence, it will change after padding since the length of the first sentence will vary. The generate pair's function is used in task 1.3 to create pairs of (anaphora indexes, antecedents indexes) and labels.

```
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 1.1 YOUR CODE HERE
      #  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)

      # TASK 1.2 YOUR CODE HERE
      # 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)

      # TASK 1.3 YOUR CODE HERE
      # 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, mention_pairs, mention_pair_labels, clusters, raw_mention_pairs))

    return processed_docs
```

It is utilized to make the cushioned embeddings of the reports, which are duplicates of the beginning and end lists, adapted to cushioning. Here generate_pairs capacities return sets of (anaphora files, precursors records) and their marks from the dummy_dataset prior to cushioning the start_index of "very" is 7 in the wake of padding the start_index of "very" will change because of the length of first sentence changing subsequent to padding.

Task 2: Building the model: -

Word embeddings layers and notice matches are the two contributions of the organization. The components of the word embeddings input layer have been diminished to the point that cluster size is as of now not a worry. These compacted word embeddings are shipped off a dropout layer with a 0.5 dropout rate. The word embeddings dropout is shipped off a Bidirectional LSTM with 50 units, or 100 lstm cells, in the following undertaking. This layer's result is sent to a Bidirectional LSTM layer with a comparable course of action. The second lstm layer's result is straightened, and a dropout of 0.2 is applied. The smoothed result and notice match input

layer are then used to develop notice pair embeddings. It's then leveled so that each sets of notices is determined by a 400d tensor. In task 2.3, two thick layers with 50 secret cells and the'relu' enactment work are added, with each layer going through a 0.2 dropout layer. One result cell and sigmoid actuation work make up the result layer. The result aspect is pressed out toward the end, leaving (bunch size x notice matches tensor) as the last aspect.

The code is as follows,

```
def build_model():
    # 1 (a.) Initialize the model inputs
    # TASK 2.1a YOUR CODE HERE
    word_embeddings = Input(shape = (None, None, EMBEDDING_SIZE,))# Word Embeddings Input layer with size (None,None,None,300)
    mention_pairs = Input(shape = (None, 4,), dtype = 'int32') # Mention_pairs input layer with shape (None,None,4)

    # squeeze the (batch_size X num_sents X num_words X embedding_size) into a
    # (num_sents X num_words X embedding_size) tensor
    word_embeddings_no_batch = Lambda(lambda x: K.squeeze(x,0))(word_embeddings)

    # 1 (b.). Apply embedding dropout to the squeezed embeddings.
    word_embeddings_dropped = Dropout(EMBEDDING_DROPOUT_RATE)(word_embeddings_no_batch)# TASK 2.1b YOUR CODE HERE

    # TASK 2.2. YOU CREATE A TWO LAYER BIDIRECTIONAL LSTM
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences = True), name = 'BiLSTM_1')(word_embeddings_dropped) #Bidirectional LSTM
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences = True), name = 'BiLSTM_2')(word_output) #Bidirectional LSTM

    # flattening the lstms output and apply dropout.
    flatten_word_output = Lambda(lambda x:K.reshape(x, [-1, 2 * HIDDEN_SIZE]))(word_output)
    flatten_word_output = Dropout(HIDDEN_DROPOUT_RATE)(flatten_word_output)

    # we gather the embeddings represented by [anaphor_start, anaphor_end, antecedent_start, antecedent_end] for each pair.
    mention_pair_emb = Lambda(lambda x: K.gather(x[0], x[1]))([flatten_word_output, mention_pairs])

    # we flatten them such that each mention_pair is represented by a 400D tensor.
    ffnn_input = Reshape((-1,8*HIDDEN_SIZE))(mention_pair_emb)

    # TASK 2.3. CREATE THE MULTILAYER PERCEPTRONS THEN SQUEEZE OUT THE LAST DIMENSION USING LAMBDA
    dense_layer1 = Dense(HIDDEN_SIZE, activation = 'relu')(ffnn_input) # Dense layer1 with 50 neurons having 'ReLU' Activation function
    dense_layer1 = Dropout(HIDDEN_DROPOUT_RATE)(dense_layer1) #Dropout of 0.2 to denselayer 1

    dense_layer2 = Dense(HIDDEN_SIZE, activation = 'relu')(dense_layer1) #Dense layer2 with 50 neurons having 'ReLU' Activation function
    dense_layer2 = Dropout(HIDDEN_DROPOUT_RATE)(dense_layer2) #Dropout of 0.2 to Denselayer2

    output = Dense(1, activation = 'sigmoid')(dense_layer2) # Output layer with 1 output neuron having Sigmoid Activation function

    mention_pair_scores = Lambda(lambda x:K.squeeze(x,-1))(output) # Squeeze out the last dimension

    model = Model(inputs=[word_embeddings,mention_pairs], outputs=mention_pair_scores)
    model.compile(optimizer='adam',loss='binary_crossentropy')
    print(model.summary())
    return model
```

Models' summary:

```
model = build_model()
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, None, None,  0           []
                                 300)]

lambda (Lambda)                 (None, None, 300)    0           ['input_1[0][0]']

dropout (Dropout)               (None, None, 300)    0           ['lambda[0][0]']

BiLSTM_1 (Bidirectional)        (None, None, 100)    140400      ['dropout[0][0]']

BiLSTM_2 (Bidirectional)        (None, None, 100)    60400       ['BiLSTM_1[0][0]']

lambda_1 (Lambda)               (None, 100)          0           ['BiLSTM_2[0][0]']

dropout_1 (Dropout)             (None, 100)          0           ['lambda_1[0][0]']

input_2 (InputLayer)            [(None, None, 4)]    0           []

lambda_2 (Lambda)               (None, None, 4, 100  0           ['dropout_1[0][0]',
                                )                                 'input_2[0][0]']

reshape (Reshape)               (None, None, 400)    0           ['lambda_2[0][0]']

dense (Dense)                   (None, None, 50)     20050       ['reshape[0][0]']

dropout_2 (Dropout)             (None, None, 50)     0           ['dense[0][0]']

dense_1 (Dense)                 (None, None, 50)     2550        ['dropout_2[0][0]']

dropout_3 (Dropout)             (None, None, 50)     0           ['dense_1[0][0]']

dense_2 (Dense)                 (None, None, 1)      51          ['dropout_3[0][0]']

lambda_3 (Lambda)               (None, None)         0           ['dense_2[0][0]']

==================================================================================================
Total params: 223,451
Trainable params: 223,451
Non-trainable params: 0
```

Task 3: Coreference evaluation (Section 6.2): -

the first step is to convert every cluster to a tuple in preference to a list. The gold clusters variable is looped for this, and every gold cluster list is saved in variable 'cl'. The variable 'cl' is then

cycled once more to reap each gold cluster, with the end result being saved as a tuple. we'll need to set up a mention to gold mapping dictionary for the following project. To acquire each cluster listing, the above-noted gold cluster tuple is cycled. we're going to loop via this cluster listing once more to find the associated mention, and we will shop the value in a dictionary with the key:price pair point out:cluster. We must acquire the anticipated clusters and the mention to expect in the final assignment. The get predicted clusters () feature is used to do this

```
[29] def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):

        # turn each cluster in the list of gold cluster into a tuple (rather than a list)
        # TASK 3.1 CODE HERE
        gold_clusters = [[tuple(m) for m in cl] for cl in gold_clusters]

        # mention to gold is a (mention: cluster of mentions it belongs, including the present mention) map
        mention_to_gold = {}
        # TASK 3.2 WRITE CODE HERE TO GENERATE mention_to_gold from gold_clusters
        for cluster in gold_clusters:
            for mention in cluster:
                mention_to_gold[mention] = tuple(cluster)

        # get the predicted_clusters and mention_to_predict using get_predicted_clusters()
        predicted_clusters, mention_to_predicted =  get_predicted_clusters(predicted_mention_pairs)# TASK 3.3 CODE HERE

        # run the evaluator using the parameters you've gotten
        evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

```
Starting training epoch 10/10
2775/2775 [==============================] - 39s 14ms/step - loss: 0.5624
Time used for epoch 10: 0 m 40 s
Evaluating on dev set after epoch 10/10:
Average F1 (py): 36.60%
Average precision (py): 41.81%
Average recall (py): 50.87%
Time used for evaluate on dev set: 0 m 1 s

Training finished!
Time used for training: 10 m 2 s

Evaluating on test set:
Average F1 (py): 35.57%
Average precision (py): 40.98%
Average recall (py): 52.09%
Time used for evaluate on test set: 0 m 1 s
```

The average f1-score for training is 35.57 percent, with precision 40.98 and recall value of 52.09 percent. However, in the test set, all of the measures are low. We can say that the model performs extremely well on both the train and the test, and that there is room for improvement by training more epochs or changing the model of the design.

Task 4: Some questions (Section 8): -

Question 1: Would the performance decrease if we do not pre-process the text? If yes (or no), then why?

The following code is the code we have had before pre-processing,

```
[ ] #get_data(json_file, is_training_preprocess_text) receives three inputs:
    #json_file (str) : the path to json file, preprocess_text)
    #is_training (boolean): this is used to with generate_pairs(...) function to balance the number of generated pairs
    #preprocess_text (boolean): whether to preprocess text or not
    DEV_DATA = get_data(DEV_PATH, False, True)
    TEST_DATA = get_data(TEST_PATH, False, True)
    TRAIN_DATA = get_data(TRAIN_PATH, True, True)
```

```
Starting training epoch 10/10
2775/2775 [==============================] - 39s 14ms/step - loss: 0.5624
Time used for epoch 10: 0 m 40 s
Evaluating on dev set after epoch 10/10:
Average F1 (py): 36.60%
Average precision (py): 41.81%
Average recall (py): 50.87%
Time used for evaluate on dev set: 0 m 1 s

Training finished!
Time used for training: 10 m 2 s

Evaluating on test set:
Average F1 (py): 35.57%
Average precision (py): 40.98%
Average recall (py): 52.09%
Time used for evaluate on test set: 0 m 1 s
```

After setting pre-process = False, the f1-score, precision, and recall for the test set have 35.5%, 40.98 and 52.09%, respectively. This occurs because the data was already pre-processed, and all reductant tokens, punctuations, and stop words were eliminated. As a result, the feature space dimension used to be tiny, but now it is vast, and the input contains a lot of inaccurate and redundant information, lowering the accuracy.

Question 2: Experiment with different values for max antecedent (MAX_ANT) and negative ratio (NEG_RATIO), what do you observe?

First, we put,

(MAX_ANT =300, NEG_RATIO = 2)

```
Starting training epoch 10/10
2775/2775 [==============================] - 29s 11ms/step - loss: 0.4180
Time used for epoch 10: 0 m 29 s
Evaluating on dev set after epoch 10/10:
Average F1 (py): 43.52%
Average precision (py): 47.77%
Average recall (py): 58.97%
Time used for evaluate on dev set: 0 m 1 s

Training finished!
Time used for training: 6 m 14 s

Evaluating on test set:
Average F1 (py): 41.09%
Average precision (py): 45.68%
Average recall (py): 58.36%
Time used for evaluate on test set: 0 m 2 s
```

Then,

(MAX_ANT = 100, NEG_RATIO = 2)

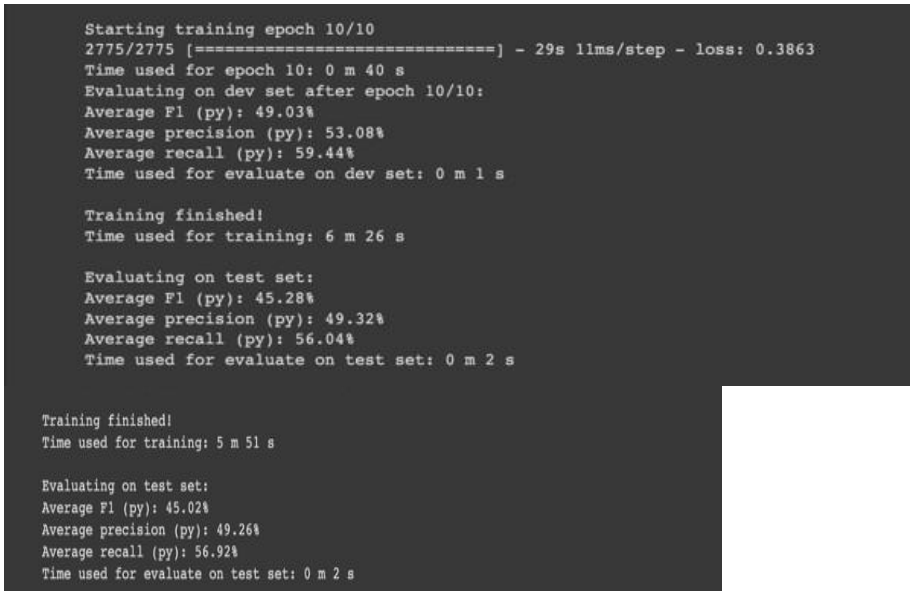
```
Starting training epoch 10/10
2775/2775 [==============================] - 30s 11ms/step - loss: 0.4205
Time used for epoch 10: 0 m 29 s
Evaluating on dev set after epoch 10/10:
Average F1 (py): 43.24%
Average precision (py): 47.96%
Average recall (py): 58.61%
Time used for evaluate on dev set: 0 m 1 s

Training finished!
Time used for training: 6 m 15 s

Evaluating on test set:
Average F1 (py): 41.22%
Average precision (py): 45.99%
Average recall (py): 57.90%
Time used for evaluate on test set: 0 m 2 s
```

After that,

(MAX_ANT = 400, NEG_RATIO = 4)

And finally

(MAX_ANT = 250, NEG_RATIO = 4)



Here, if MAX_ANT is high and the constant NEG_RATIO is 2, MAX_ANT is low and the F1 score is low compared to the constant NEG_RATIO. This may be due to more pairs to learn by setting MAX_ANT to a higher value. Keeping MAX_ANT constant and NEG_RATIO at 4 will result in a lower F1 score than keeping MAX_ANT constant and lowering NEG_RATIO. This may be due to the higher the NEG_RATIO, the more unbalanced the dataset.

Question 3: How would you improve the accuracy?

According to the findings, a very high or very low MAX ANT value will result in low f1-score accuracy, and a high NEG RATIO would result in a poor f1-score. In order to enhance accuracy, we need a low NEG RATIO number so that we may have a balanced dataset, and we need an ideal MAX ANT value (Not too high to not too low)

Part D - Dialogue 1: Dialogue Act Tagging: -

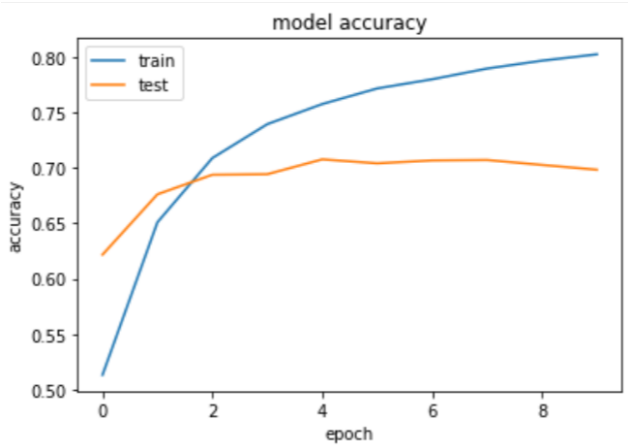
Task 1: Implementing an utterance-based tagger, using standard text classification methods from lectures: -

The input layer for Model 1 is Embedding, with an EMBED SIZE of 100 and a maximum

length of 150. These embeds are sent to a bidirectional LSTM layer with 43 units and 86 lstm cells. The output of this layer is passed to the bidirectional LSTM layer with return sequence = False. The dense layer with 43 hidden cells receives the output of the second bidirectional LSTM layer. To estimate the probability of 43 tags, the output layer has 43 output cells and a SoftMax activation function.

```
# Include 2 BLSTM layers, in order to capture both the forward and backward hidden states
model = Sequential()
# Embedding layer with output size = EMBED_SIZE
model.add(Embedding(VOCAB_SIZE, EMBED_SIZE, input_length = MAX_LENGTH))
# Bidirectional 1
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences= True)))
# Bidirectional 2
model.add(Bidirectional(LSTM(HIDDEN_SIZE)))
# Dense layer
model.add(Dense(HIDDEN_SIZE))
# Activation
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 150, 100)          4373200

 bidirectional (Bidirectiona  (None, 150, 86)          49536
 l)

 bidirectional_1 (Bidirectio  (None, 86)               44720
 nal)

 dense (Dense)               (None, 43)                3741

 activation (Activation)     (None, 43)                0

=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```




```
# Train the model - use validation
history = model.fit(train_input, train_labels, batch_size=512, epochs=10, validation_data=(val_input, val_labels))
```

```
Epoch 1/10
274/274 [==============================] - 77s 239ms/step - loss: 1.7744 - accuracy: 0.5132 - val_loss: 1.3442 - val_accuracy: 0.6216
Epoch 2/10
274/274 [==============================] - 63s 231ms/step - loss: 1.2037 - accuracy: 0.6508 - val_loss: 1.1286 - val_accuracy: 0.6760
Epoch 3/10
274/274 [==============================] - 63s 231ms/step - loss: 1.0135 - accuracy: 0.7087 - val_loss: 1.0617 - val_accuracy: 0.6936
Epoch 4/10
274/274 [==============================] - 64s 233ms/step - loss: 0.9029 - accuracy: 0.7392 - val_loss: 1.0326 - val_accuracy: 0.6941
Epoch 5/10
274/274 [==============================] - 64s 233ms/step - loss: 0.8285 - accuracy: 0.7573 - val_loss: 1.0169 - val_accuracy: 0.7075
Epoch 6/10
274/274 [==============================] - 64s 233ms/step - loss: 0.7737 - accuracy: 0.7714 - val_loss: 1.0417 - val_accuracy: 0.7039
Epoch 7/10
274/274 [==============================] - 64s 233ms/step - loss: 0.7359 - accuracy: 0.7796 - val_loss: 1.0283 - val_accuracy: 0.7065
Epoch 8/10
274/274 [==============================] - 64s 234ms/step - loss: 0.6990 - accuracy: 0.7893 - val_loss: 1.0310 - val_accuracy: 0.7069
Epoch 9/10
274/274 [==============================] - 64s 232ms/step - loss: 0.6725 - accuracy: 0.7964 - val_loss: 1.0530 - val_accuracy: 0.7025
Epoch 10/10
274/274 [==============================] - 63s 231ms/step - loss: 0.6465 - accuracy: 0.8021 - val_loss: 1.0911 - val_accuracy: 0.6981
```

```
[ ] score = model.evaluate(test_sentences_X, y_test, batch_size=100)
```

```
560/560 [==============================] - 33s 59ms/step - loss: 1.1279 - accuracy: 0.6842
```

```
[ ] print("Overall Accuracy:", score[1]*100)
```

```
Overall Accuracy: 68.41973662376404
```

The accuracy score grows as the number of epochs increases, while the validation accuracy remains somewhat stable after a few epochs and is also substantially lower than the training accuracy, as shown in the Accuracy plot. It's likely that the model is overfit as a result of this. The accuracy was found to be 68.41 percent.

Task 2: Minority DA tag class analysis and utterance-based tagger with re-balanced weighted cost function: -

In this case, the prediction comes first. After that, the sklearn library is used to generate a confusion matrix. The classification report and accuracy score are printed to examine the f1-score, precision, and recall for each label, as well as the overall accuracy of the model. We're now looking at the proportion of 'br' and 'bf' minority classes in our dataset. Only 0.13 percent of the data is classified as "br," while 0.42 percent is classified as "bf."

```
[ ]  # Calculate Accuracies for "br" and "bf"
     acc_class = confusion_matrix.diagonal()/confusion_matrix.sum(axis=1)

     index_br = list(one_hot_encoding_dic["br"][one_hot_encoding_dic["br"]==1].index)[0]
     br_accuracy = acc_class[index_br]*100
     print("br accuracy: {}".format(br_accuracy))

     index_bf = list(one_hot_encoding_dic["bf"][one_hot_encoding_dic["bf"]==1].index)[0]
     bf_accuracy = acc_class[index_bf]*100
     print("bf accuracy: {}".format(bf_accuracy))

     br accuracy: 60.71428571428571
     bf accuracy: 4.733727810650888
```

```
     # Print the frequency of the "br" and "bf" classes
     value_counts = reduced_df["act_tag"].value_counts()
     bf_frequency = value_counts["bf"]/sum(value_counts)
     print("bf frequency: " + str(bf_frequency*100) + "%")

     br_frequency = value_counts["br"]/sum(value_counts)
     print("br frequency: " + str(br_frequency*100) + "%")

     bf frequency: 0.42977379855638936%
     br frequency: 0.13371734211067682%
```

Report,

	precision	recall	f1-score	support
0	0.72	0.47	0.57	746
1	0.23	0.31	0.26	196
2	0.00	0.00	0.00	66
3	0.53	0.44	0.48	6988
4	0.00	0.00	0.00	25
5	0.06	0.04	0.05	126
6	0.53	0.83	0.65	312
7	0.76	0.80	0.78	3710
8	0.61	0.33	0.43	282
9	0.27	0.34	0.30	265
10	0.49	0.30	0.37	2384
11	0.00	0.00	0.00	26
12	0.66	0.31	0.43	181
13	0.50	0.05	0.09	20
14	0.79	0.92	0.85	10529
15	0.07	0.02	0.04	214
16	0.47	0.34	0.39	130
17	0.48	0.55	0.52	56
18	0.49	0.56	0.52	52
19	0.52	0.72	0.61	428
20	0.00	0.00	0.00	78
21	0.94	0.94	0.94	1085
22	0.19	0.24	0.21	45
23	0.73	0.79	0.76	18047
24	0.32	0.09	0.14	880
25	0.00	0.00	0.00	20
26	0.00	0.00	0.00	21
27	0.46	0.35	0.40	17
28	0.00	0.00	0.00	60
29	0.64	0.67	0.65	4867
30	0.02	0.01	0.01	272
31	0.20	0.01	0.03	74
32	0.00	0.00	0.00	15
33	0.72	0.55	0.62	1284
34	0.07	0.07	0.07	344
35	0.04	0.02	0.03	169
36	0.00	0.00	0.00	14
37	0.56	0.62	0.59	1132
38	0.04	0.03	0.03	156
39	0.77	0.75	0.76	248
40	0.00	0.00	0.00	218
41	0.00	0.00	0.00	23
42	0.58	0.68	0.63	259
accuracy			0.68	55902
macro avg	0.34	0.31	0.31	55902
weighted avg	0.66	0.68	0.67	55902

Here, we're calculating the precision of the 'br' and 'bf' classes. The true labels are obtained by calling the function confusion matrix.diagonal(), which is then divided by confusion matrix.sum(axis = 1). This will give us the accuracy for each class. Then, to get the percent number, multiply the accuracies for the 'br' and 'bf' classes by 100.

```
# Calculate Accuracies for "br" and "bf"
acc_class_balanced = confusion_matrix_balanced.diagonal()/confusion_matrix_balanced.sum(axis=1)

index_br = list(one_hot_encoding_dic["br"][one_hot_encoding_dic["br"]==1].index)[0]
br_accuracy = acc_class_balanced[index_br]*100
print("br accuracy: {}".format(br_accuracy))

index_bf = list(one_hot_encoding_dic["bf"][one_hot_encoding_dic["bf"]==1].index)[0]
bf_accuracy = acc_class_balanced[index_bf]*100
print("bf accuracy: {}".format(bf_accuracy))
```

```
br accuracy: 55.35714285714286
bf accuracy: 2.366863905325444
```

The results show that the model correctly predicts 55% of 'br' classes but only 2.3% of 'bf' classes. Furthermore, according to the categorization report, 'br' has better precision and recall, so while 'bf' has too low precision and recall.

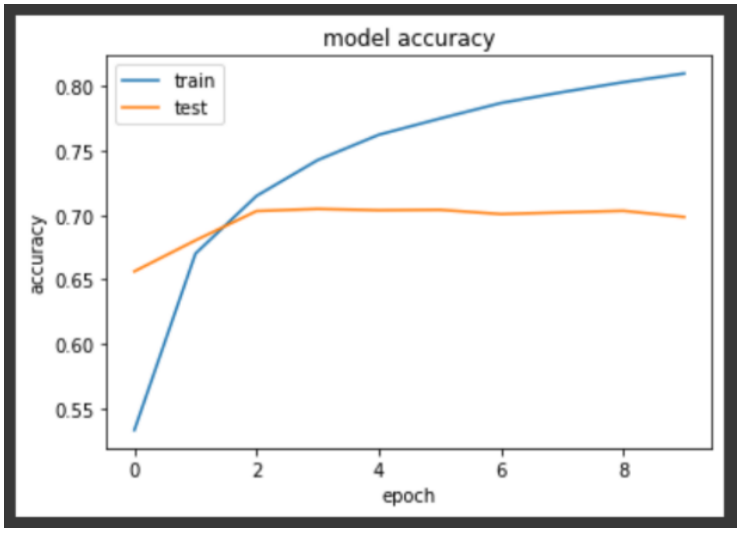
1. **Task 3:** Implementing a hierarchical utterance+DA-context-based tagger:

```
# Re-built the model for the balanced training
model_balanced = Sequential()
# Embedding layer
model_balanced.add(Embedding(VOCAB_SIZE, EMBED_SIZE, input_length = MAX_LENGTH))
# Bidirectional 1
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences= True)))
# Bidirectional 2
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE)))
# Dense layer
model_balanced.add(Dense(HIDDEN_SIZE))
# Activation
model_balanced.add(Activation('softmax'))

model_balanced.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

model_balanced.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 150, 100)          4373200

bidirectional_2 (Bidirectio  (None, 150, 86)           49536
nal)

bidirectional_3 (Bidirectio  (None, 86)                44720
nal)

dense_1 (Dense)              (None, 43)                3741

activation_1 (Activation)    (None, 43)                0

=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the balanced network - Seems to take long time to achieve good accuracy?
history1 = model_balanced.fit(train_input, train_labels, batch_size=512, epochs=10, validation_data=(val_input, val_labels))
```

```
Epoch 1/10
274/274 [==============================] - 72s 237ms/step - loss: 1.7139 - accuracy: 0.5329 - val_loss: 1.2643 - val_accuracy: 0.6563
Epoch 2/10
274/274 [==============================] - 63s 230ms/step - loss: 1.1510 - accuracy: 0.6701 - val_loss: 1.1090 - val_accuracy: 0.6804
Epoch 3/10
274/274 [==============================] - 63s 230ms/step - loss: 0.9802 - accuracy: 0.7149 - val_loss: 1.0286 - val_accuracy: 0.7031
Epoch 4/10
274/274 [==============================] - 63s 231ms/step - loss: 0.8807 - accuracy: 0.7428 - val_loss: 1.0223 - val_accuracy: 0.7048
Epoch 5/10
274/274 [==============================] - 63s 231ms/step - loss: 0.8093 - accuracy: 0.7624 - val_loss: 1.0270 - val_accuracy: 0.7038
Epoch 6/10
274/274 [==============================] - 63s 231ms/step - loss: 0.7574 - accuracy: 0.7750 - val_loss: 1.0222 - val_accuracy: 0.7041
Epoch 7/10
274/274 [==============================] - 63s 231ms/step - loss: 0.7143 - accuracy: 0.7871 - val_loss: 1.0473 - val_accuracy: 0.7008
Epoch 8/10
274/274 [==============================] - 63s 231ms/step - loss: 0.6814 - accuracy: 0.7955 - val_loss: 1.0644 - val_accuracy: 0.7021
Epoch 9/10
274/274 [==============================] - 63s 231ms/step - loss: 0.6523 - accuracy: 0.8033 - val_loss: 1.0845 - val_accuracy: 0.7033
Epoch 10/10
274/274 [==============================] - 63s 230ms/step - loss: 0.6247 - accuracy: 0.8100 - val_loss: 1.0883 - val_accuracy: 0.6985
```




```
[ ] # Calculate Accuracies for "br" and "bf"
    acc_class_balanced = confusion_matrix_balanced.diagonal()/confusion_matrix_balanced.sum(axis=1)

    index_br = list(one_hot_encoding_dic["br"][one_hot_encoding_dic["br"]==1].index)[0]
    br_accuracy = acc_class_balanced[index_br]*100
    print("br accuracy: {}".format(br_accuracy))

    index_bf = list(one_hot_encoding_dic["bf"][one_hot_encoding_dic["bf"]==1].index)[0]
    bf_accuracy = acc_class_balanced[index_bf]*100
    print("bf accuracy: {}".format(bf_accuracy))

    br accuracy: 55.35714285714286
    bf accuracy: 2.366863905325444
```

Model2's overall accuracy is slightly lower than model 1's, at 68.40 percent. Furthermore, as shown in the graph above, training accuracy improves as the number of epochs increases, whereas validation accuracy remains stable after a certain number of epochs. Furthermore, the training accuracy is higher than the testing accuracy, indicating that the model may indeed be overfit. The accuracy of the 'br' has increased slightly to 55.35 percent, while that of the 'bf' has decreased to 2.36 percent. This is because balanced data is transmitted during training.

	precision	recall	f1-score	support
0	0.72	0.47	0.57	746
1	0.23	0.31	0.26	194
2	0.00	0.00	0.00	66
3	0.53	0.44	0.48	6988
4	0.00	0.00	0.00	25
5	0.06	0.04	0.05	126
6	0.53	0.83	0.65	312
7	0.76	0.80	0.78	3710
8	0.61	0.33	0.43	282
9	0.27	0.34	0.30	265
10	0.49	0.30	0.37	2384
11	0.00	0.00	0.00	26
12	0.66	0.31	0.43	181
13	0.50	0.05	0.09	20
14	0.79	0.92	0.85	10529
15	0.07	0.02	0.04	214
16	0.47	0.34	0.39	130
17	0.48	0.55	0.52	56
18	0.49	0.56	0.52	52
19	0.52	0.72	0.61	428
20	0.00	0.00	0.00	78
21	0.94	0.94	0.94	1005
22	0.19	0.24	0.21	45
23	0.73	0.79	0.76	18047
24	0.32	0.09	0.14	800
25	0.00	0.00	0.00	20
26	0.00	0.00	0.00	21
27	0.46	0.35	0.40	17
28	0.00	0.00	0.00	60
29	0.64	0.67	0.65	4867
30	0.02	0.01	0.01	272
31	0.20	0.01	0.03	74
32	0.00	0.00	0.00	15
33	0.72	0.55	0.62	1284
34	0.07	0.07	0.07	344
35	0.04	0.02	0.03	169
36	0.00	0.00	0.00	14
37	0.56	0.62	0.59	1132
38	0.04	0.03	0.03	156
39	0.77	0.75	0.76	248
40	0.00	0.00	0.00	218
41	0.00	0.00	0.00	23
42	0.58	0.68	0.63	259
accuracy			0.68	55902
macro avg	0.34	0.31	0.31	55902
weighted avg	0.66	0.68	0.67	55902

Test the model

```
[ ] # Overall Accuracy
    score = model_balanced.evaluate(test_sentences_X, y_test, batch_size=100)
```

```
560/560 [==============================] - 33s 59ms/step - loss: 1.1268 - accuracy: 0.6841
```

```
print("Overall Accuracy:", score[1]*100)
```

```
Overall Accuracy: 68.40900182723999
```

The architecture of the Model 3 is as follows:

CNN, BiLSTM, and FCL are examples of word embeddings. The first step is to generate word embeddings that can be sent to the CNN layer. The input layer has the shape (150,), and the Embeddings are created by passing the input layer. The output is then rearranged before being sent to the CNN layer. Three 2D CNN layers and three Batch Normalization layers are now available. The kernel sizes used were (3,100), (4,100), and (4,100). (5,100). The number of kernels is set to 64 for all three levels, and the padding is set to valid. The output of these CNN layers is sent to three maxpool layers with pool sizes of (148,1), (147,1), and (146,1), respectively.

All the maxpooled layers' outputs are now concatenated and flattened before being fed into a dense layer with a 0.2 dropout rate. There are now two bidirectional LSTM layers, each containing 43 LSTM units (86 cells). The Dense layer, which has a dropout rate of 0.2, receives the output of the second bidirectional layer. In the end, the last two levels are combined. There are 43 hidden cells and a SoftMax activation function in the output layer.

CNN

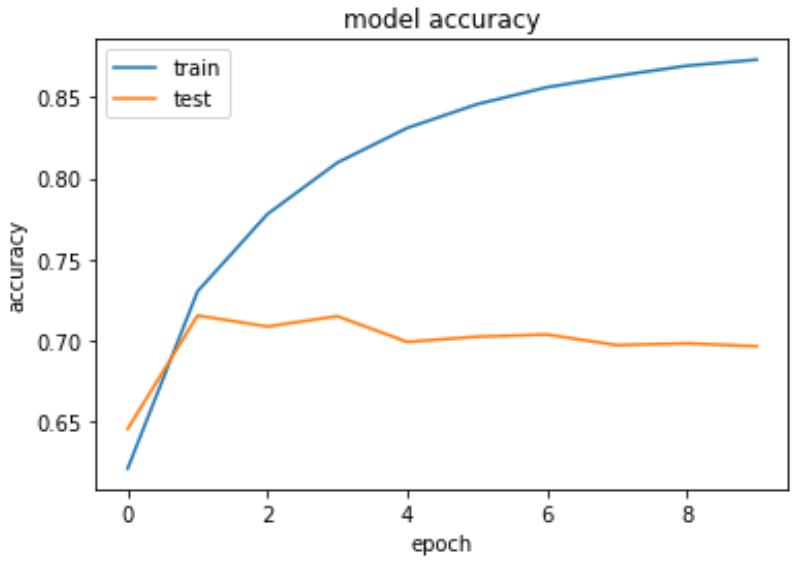
```
[ ] # concatenate tensors
    concatenated_tensors = Concatenate()([maxpool_0, maxpool_1, maxpool_2])
    # flatten concatenated tensors
    flatten_concatenated_tensors = TimeDistributed(Flatten())(concatenated_tensors)
    # dense layer (dense_1)
    dense_1 = Dense(HIDDEN_SIZE, activation='relu')(flatten_concatenated_tensors)
    # dropout_1
    dropout_1 = Dropout(drop)(dense_1)
```

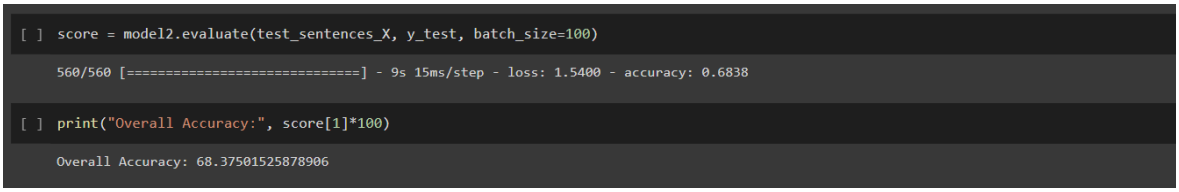
BLSTM

```
[ ]  # BLSTM model

     # Bidirectional 1
     Bidirectional1 = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences='true'))(dropout_1)
     # Bidirectional 2
     Bidirectional2 = Bidirectional(LSTM(HIDDEN_SIZE))(Bidirectional1)
     # Dense layer (dense_2)
     dense_2 = Dense(HIDDEN_SIZE, activation='relu')(Bidirectional2)
     # dropout_2
     dropout_2 = Dropout(drop)(dense_2)
```

```
     # concatenate 2 final layers
     flattened_dropout_1 = Flatten()(dropout_1)
     concatenate_2_final_layer = Concatenate()([flattened_dropout_1,dropout_2])

     # output
     output_layer = Dense(HIDDEN_SIZE, input_shape=(1,))(concatenate_2_final_layer)
     output = Activation('softmax')(output_layer)

     model2 = Model(inputs=[inputs], outputs=[output])

     # Compile the model
     model2.compile(loss='categorical_crossentropy', optimizer='adam', metrics = ['accuracy'])
     model2.summary()
```

```
Model: "model"
_________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
=================================================================================================
input_1 (InputLayer)            [(None, 150)]        0           []

embedding_2 (Embedding)         (None, 150, 100)     4373200     ['input_1[0][0]']

reshape (Reshape)               (None, 150, 100, 1)  0           ['embedding_2[0][0]']

conv2d (Conv2D)                 (None, 148, 1, 64)   19264       ['reshape[0][0]']

conv2d_1 (Conv2D)               (None, 147, 1, 64)   25664       ['reshape[0][0]']

conv2d_2 (Conv2D)               (None, 146, 1, 64)   32064       ['reshape[0][0]']

batch_normalization (BatchNorm  (None, 148, 1, 64)   256         ['conv2d[0][0]']
alization)

batch_normalization_1 (BatchNo  (None, 147, 1, 64)   256         ['conv2d_1[0][0]']
rmalization)

batch_normalization_2 (BatchNo  (None, 146, 1, 64)   256         ['conv2d_2[0][0]']
rmalization)

max_pooling2d (MaxPooling2D)    (None, 1, 1, 64)     0           ['batch_normalization[0][0]']

max_pooling2d_1 (MaxPooling2D)  (None, 1, 1, 64)     0           ['batch_normalization_1[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 1, 1, 64)     0           ['batch_normalization_2[0][0]']

concatenate (Concatenate)       (None, 1, 1, 192)    0           ['max_pooling2d[0][0]',
                                                                  'max_pooling2d_1[0][0]',
                                                                  'max_pooling2d_2[0][0]']

time_distributed (TimeDistribu  (None, 1, 192)       0           ['concatenate[0][0]']
ted)

dense_2 (Dense)                 (None, 1, 43)        8299        ['time_distributed[0][0]']

dropout (Dropout)               (None, 1, 43)        0           ['dense_2[0][0]']

bidirectional_4 (Bidirectional  (None, 1, 86)        29928       ['dropout[0][0]']
)

bidirectional_5 (Bidirectional  (None, 86)           44720       ['bidirectional_4[0][0]']
)

dense_3 (Dense)                 (None, 43)           3741        ['bidirectional_5[0][0]']

flatten_1 (Flatten)             (None, 43)           0           ['dropout[0][0]']

dropout_1 (Dropout)             (None, 43)           0           ['dense_3[0][0]']

concatenate_1 (Concatenate)     (None, 86)           0           ['flatten_1[0][0]',
                                                                  'dropout_1[0][0]']

dense_4 (Dense)                 (None, 43)           3741        ['concatenate_1[0][0]']

concatenate_1 (Concatenate)     (None, 86)           0           ['flatten_1[0][0]',
                                                                  'dropout_1[0][0]']

dense_4 (Dense)                 (None, 43)           3741        ['concatenate_1[0][0]']

activation_2 (Activation)       (None, 43)           0           ['dense_4[0][0]']

=================================================================================================
Total params: 4,541,389
Trainable params: 4,541,005
Non-trainable params: 384
_________________________________________________________________________________________________
```





Model 3's accuracy is 68.37%, which is higher than the other two versions. The addition of BiLSTM layers on top of CNN layers to interpret the model's output could be the reason for this. In the data, these layers may have discovered patterns and context information. The training accuracy is increasing on the graph, while the validation accuracy is stable after a few 1 epoch. In addition, the training accuracy is too high in comparison to the validation accuracy, indicating that the model is overfit. Overfitting can be overcome using regularisation techniques.

Bi-LSTM layers have improved the model's grasp of context word data. In general, accuracy has improved. The accuracy of the minority tag, on the other hand, remains unaltered from the previous model. As a result, improving the accuracy of minority DA tags may require a different sort of architecture. To increase minority class accuracy, one possibility is to use GRU instead of LSTM, or encoder-decoder based transformers with the capacity to grasp context word information.

Part E - Dialogue 2: A Conversational Dialogue System: -

NOTE: Lab 12 was only able to run 23 epochs due to google colab constraints

Task 1: Implementing the encoder: -

The embeddings layer, which has an embedding dim of 50, is the first layer in the Encoder Architecture. A 0.2 dropout is applied to this embeddings layer. The dropout output is then passed to a bidirectional GRU layer consisting of 50 GRU units or 100 GRU cells, all of which have return sequences equal to True. This layer also has a 0.2 dropout. This layer dropout layer's output is then transferred to a 100-cell bidirectional GRU. The output label, state f, and state b are all produced by the final output layer. The decoder's first GRU layer will be initialised using the second GRU layer's most recent concealed and cell state, resulting in the second GRU layer's return state is true.

```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units):
        super(Encoder, self).__init__()
        self.batch_sz = batch_size
        self.enc_units = enc_units

        #  pass the embedding into a bidirectional version of the GRU - as you can see in the call() method below, you can use just 1 GRU layer but could experiment with more
        self.embeddings = embeddings # Embedding layer
        self.dropout = Dropout(0.2) # Dropout Layer with 0.2 as dropout rate
        self.Inp = Input(shape=(max_len_q,)) # size of questions
        # Bidirectional GRU layer1 with 50 GRU units i.e 100 cells and return_sequences = True
        self.Bidirectional1 = Bidirectional(GRU(self.enc_units, return_state= False, return_sequences= True))
        # Bidirectional GRU layer2 with 50 GRU units i.e 100 cells and return_sequences = True and return_states = True
        self.Bidirectional2 = Bidirectional(GRU(self.enc_units, return_state= True, return_sequences= True))

    def bidirectional(self, bidir, layer, inp, hidden):
        return bidir(layer(inp, initial_state = hidden))

    def call(self, x, hidden):
        x = self.embeddings(x)
        x = self.dropout(x)
        x = self.Bidirectional1(x)
        x = self.dropout(x)
        output, state_f,state_b = self.Bidirectional2(x)

        return output, state_f, state_b

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

Task 2: Implementing the decoder with attention: -

The first layer, bahdanau Attention, is initialised with the encoders' most recent hidden and cell states. To identify the relevance of each input token for constructing the word at that time, the attention layer is utilised to compute the attention weights, which are a scaled dot product of all encoder hidden states and decoder hidden states at the current time step. The output is placed on a layer with the following dimensions: batch size, 1, embedding dim. The new decoder state for the next word generation at the following time step is the concatenated recent hidden and cell states.

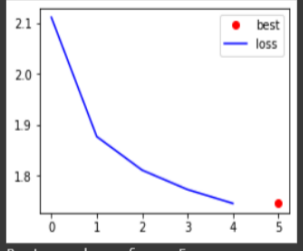
This output is fed into a GRU layer with 100 GRU cells, return sequences = True, and a 20% dropout. This output is then passed via a GRU layer with a value of 100 and a True return state. Once again, a 0.2 dropout is employed. The last output layer is a dense layer with 14500 output cells and a softmax activation function.

```python
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.batch_sz = batch_size
        self.embeddings = embeddings
        self.units = 2 * dec_units # because we use bidirectional encoder
        self.fc = Dense(vocab_len, activation='softmax', name='dense_layer')
        # Create the decoder with attention - as you'll see in the call() method below, it will need two GRU layers
        self.dropout = Dropout(0.2) # Dropout Layer with 0.2 as dropout rate
        self.attention = BahdanauAttention(self.units) # BahdanauAttention Layer with 100 units

        # GRU Layer 1 with 100 GRU units and return_sequences = True
        self.decoder_gru_l1 = GRU(self.units, return_sequences=True, return_state=False)
        # GRU layer 2 with 100 GRU units and return_sequences = True and return_state = True
        self.decoder_gru_l2 = GRU(self.units, return_sequences=False, return_state=True)



    def call(self, x, hidden, enc_output):

        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden, enc_output)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embeddings(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1) # concat input and context vector together

        # passing the concatenated vector to the GRU
        x = self.decoder_gru_l1(x)
        x = self.dropout(x)
        output, state = self.decoder_gru_l2(x)
        x = self.fc(output)
        return x, state, attention_weights
```

Task 3: Investigating the encoder-decoder network's behavior and attributes: -

For 5th Epoch:

```
Epoch 5 Batch 598 Loss: 1.4714
Epoch 5 Batch 1196 Loss: 2.1481
Epoch 5 Batch 1794 Loss: 1.9273
Epoch 5 Batch 2392 Loss: 2.0420
Epoch 5 Batch 2990 Loss: 1.8207
Epoch 5 Batch 3588 Loss: 2.1801

*** Epoch 5 Loss 1.7450 ***

####################
Greedy| Q: Hello   A: hi
%
Greedy| Q: How are you ?  A: i am not going to do
%
Greedy| Q: What are you doing ?  A: i am not going to do
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
```




```
Best epoch so far:  5
Time  646.135 sec
```

The model's accuracy is too high, and the model isn't answering the questions correctly, as we can see here.

10 Predictions for 5th Epoch:

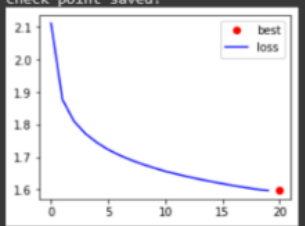
For 5th Epoch

```
checkpoint.restore(str(emb_dim)+"-ckpt-5")
test_bot_final()
```

```
###################
Greedy| Q: Hello   A: hi
%
Greedy| Q: How are you ?  A: i am not
%
Greedy| Q: Which is your favourite place ?  A: i am not
%
Greedy| Q: When will you come ?  A: i am not
%
Greedy| Q: Are you coming to the party ?  A: i am not
%
Greedy| Q: which is your favourite restaurant ?  A: i am not
%
Greedy| Q: What is your favorite film ?  A: i am not
%
Greedy| Q: what is your age ?  A: i am not
%
Greedy| Q: how is the temperature today ?  A: i am not
%
Greedy| Q: What is the capital city of India ?  A: i am not
####################
```

For 20th Epoch:

```
====================================
Epoch 20 Batch 598 Loss: 1.3456
Epoch 20 Batch 1196 Loss: 2.2188
Epoch 20 Batch 1794 Loss: 1.9011
Epoch 20 Batch 2392 Loss: 1.9216
Epoch 20 Batch 2990 Loss: 1.6948
Epoch 20 Batch 3588 Loss: 1.9117

*** Epoch 20 Loss 1.5969 ***

###################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not
%
Greedy| Q: What are you doing ?  A: i am not going to do
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do
%
Greedy| Q: Do you want to go out ?  A: no
###################
check point saved!
```




```
Best epoch so far:  20
Time  643.184 sec
```

The model's loss has been reduced here, but the context remains difficult to understand. For example, the answer to the question "How are you?" is "I'm not like a lot of things," indicating that the model has learned something but still needs more training.

10 Predictions for 20th Epoch:

For 20th Epoch

```
[ ] checkpoint.restore(str(emb_dim)+"-ckpt-20")
    test_bot_final()

    ###################
    Greedy| Q: Hello   A: hello
    %
    Greedy| Q: How are you ?  A: i am not sure
    %
    Greedy| Q: Which is your favourite place ?  A: yes
    %
    Greedy| Q: When will you come ?  A: i am sorry
    %
    Greedy| Q: Are you coming to the party ?  A: no
    %
    Greedy| Q: which is your favourite restaurant ?  A: oh
    %
    Greedy| Q: What is your favorite film ?  A: i am going to do it
    %
    Greedy| Q: what is your age ?  A: i am a witch
    %
    Greedy| Q: how is the temperature today ?  A: i do not know
    %
    Greedy| Q: What is the capital city of India ?  A: i am sorry i am sorry i am sorry i am sorry i am sorry i am sorry
    ###################
```

After the 20 epoch, the loss has not decreased significantly. However, the model is attempting to respond to new responses such as “yes, no” despite the fact that it is irrelevant to the inquiry.

Questions:

Question 1: Did the models learn to track local relations between words?

Yes, the model learns to track local word relationships. When we consider the question 'What is your age?', the matrix reveals a stronger connection between the question's terms and the answer, as well as between the question and the answer itself.

Question 2: 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?

For the first few epochs, we can see that the model repeats some responses regardless of the question asked. For example, in the above-shown epoch5, the answer 'I am not' appears frequently. The response 'I'm sorry' is also commonly used for epochs 20. These examples show that the model has been overfitted. The responses, however, begin to make more sense as the number of epochs increases. For example, if the question is "what is your favourite restaurant?" and the response is "I don't know," which makes sense, the response is "I am not," which is extremely common.

Question 3: Did the models learn to track some major syntactic relations in the utterances(e.g., subject-verb, verb-object)?

Yes, the model picks up on some subject-verb and verb-object relationships. For example, in the 11th epoch, the model answers 'I am not going to do,' but in the 15th epoch, it answers 'I am going to do with you,' demonstrating the subject-verb-object relationship. In addition, in the following predictions for the question 'How are you?' the model first answers 'I am not' in the 5th epoch, which is illogical, but then answers 'I am not sure' after the 20th epoch.

```
*** Epoch 11 Loss 1.6564 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not going to do
%
Greedy| Q: What are you doing ?  A: i am not going to do
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
Best epoch so far:  11
Time  645.084 sec
```

```
*** Epoch 15 Loss 1.6249 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not going to do
%
Greedy| Q: What are you doing ?  A: i am not going to do
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to do
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
```

Question 4: Do they learn to encode some other linguistic features? Do they capture part-of-speech tags (POS tags)?

Yes, increasing the epochs helps the model interpret the POS tag better. For example, in the 5th epoch, the answer to the question 'How are you?' is 'I am not,' but in the 140th epoch, the answer is 'I am not sure,' which is both grammatically correct and relevant. Furthermore, the response ‘six' to the question 'what is your age?' shows that the model can capture linguistic elements.

Question 5: What is the effect of more training on the length of response?

We can conclude from the results of the 5th and 20th epochs that there is some influence of greater training on response duration. However, we are unable to claim this finding because the loss is too great, and the majority of responses are irrelevant. As a result, there may or may not be an effect of more training on response length.

Question 6: In some instances, by the time the decoder has to generate the beginning of aresponse, it may already forget the most relevant early query tokens. Can you suggest waysto 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?

Although the GRU has two reset and update gates, it is incapable of recalling longer sequences. GRU is now faster and consumes less memory as a result. As an alternative to GRU, we can employ an LSTM, which has three gates: input, output, and forget. As a result, LSTM retains information more precisely and effectively in longer sequences. Transformers can also be used to store data for a longer period of time. Attention weights can also be used to solve this problem.