COMP6714 Project Report

z5185341 Chihao Chen
z5153871 Xiaowen Zhuang

* note: when saying a “tag”, we mean a “label”, i.e. a hyponym or hypernym

Q1. evaluate()

1. How do we implement

- **Pseudo code:**

```
evaluate(golden_list, predict_list):
        golden_tag_list = get_tag(golden_list)       // extract tags in the golden_list
        predict_tag_list = get_tag(predict_list)     // extract tags in the predict_list
        tp+fn = get_len(golden_tag_list)             // the number of results that are relevant
        tp+fp = get_len(predict_tag_list)            // the number of results that are retrieved
        tp = num of tags appearing in both golden_tag_list and predict_tag_list
        P = tp/(tp+fp)
        R = tp/(tp+fn)
        F1 = 2*P*R /(P+R)
Helper functions
get_tag(list): extract the tags from the list provided to form tags_list
get_dict_len(list): get the total number of tags in the collection
get_tp()
f1_score(P,R)
```

- **Implementation details:**
  1) We have to take care of the sentence dimension(the first dimension of the list) properly

  2) get_tag(): We use dictionary and sentence/word index to represent tags_list.
     E.g.
     golden_list = [['B-TAR', 'I-TAR', 'I-TAR', 'B-HYP'], ['B-TAR', 'O', 'O', 'B-HYP']]
     We have the golden_tag_list: {0: [[0, 1, 2], [3]], 1: [[0], [3]]}
     The key of dictionary is the index of the sentence, and the value is the a list of tags in that sentence. A tag is represented by a list of word index. [0,1,2] means word indexed 0, 1, 2 in that sentence form a tag

  3) get_dict_len(): we use a for loop to add up the number of tags in each sentence in the collection

  4) get_tp():

```
for tag in gold_tag_list:
    for each word_index in the tag:
```

check if the words with the index are the same in golden_list and predict_list,
If it is the end of the tag, check the word with next index in predict_list is not I-*
return corresponding result

5) Special cases:
We have: P = tp/(tp+fp) R = tp/(tp+fn) F1 = 2*P*R /(P+R)
   A. tp+fp == 0 <=> tp == 0 && get_len(predict_tag_list) ==0
   B. tp+fn == 0 <=> tp == 0 && get_len(golden_tag_list) ==0
   C. P + R == 0 <=> tp == 0 && fp/fn != 0
   D. tp==fp==fn==0
      <=> get_len(predict_tag_list) == get_len(golden_tag_list) ==0

Solutions:
- A/B/C will raise divided by zero exception, we use try-catch block to return F1 = 0
- Use if() to do with D

2. <u>How does it affect the performance</u>

Apply the test.txt data onto the saved trained model. Compare the performance of the baseline model and of the one with best F1 score.

- **Data collected:**

	total_training_time(s)	average_time/epoch(s)	loss	F1-score on test data
Baseline Model:	236.749160	4.734983	0.076307	0.719016
Model with best F1 score:	335.302752	6.706055	0.075469	0.812500

- **Conclusion:** It doesn't affect much on the performance except that it takes longer to train.
- **Analysis**
  - Accuracy:
  F-measure is a way to evaluate the information retrieval system. It is a way to measure "user happiness". We usually use it to help us select well-performance model.
  Actually we can see the F1 score is increasing in the early training process. At this time saving the best F1 model is the same as saving the newest model. When F1 stops increasing continually, the model itself is learning slowly as well.
  - Speed: Applying developing data on the model to get the best F1 score model takes time.

Q2. new_LSTMCELL()

1. <u>How do we implement</u>

- **Pseudo code:**

```
new_LSTMCELL(in, hidden, w_ih, w_hh, bih, bhh):
        Create dense(linear) layer(s) for each gate
        Apply activation functions on each gate:
            $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$        forget_gate: sigmoid
            $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$        cell_gate: tanh
            $o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$        out_gate: sigmoid
            $h_t = o_t * \tanh(C_t)$        output : outgate * tanh(cell_state)
            $C_t = f_t * C_{t-1} + \mathbf{(1 - f_t)} * \tilde{C}_t$        cell_state_out = forget * cell_state_in + (1-forget) * cell_gate
        Return output, cell_state_out for next cell
```

2. <u>How does it affect the performance</u>

Apply the test.txt data onto the saved trained model. Compare the performance of the baseline model and of the one employing new_lstmcell.

- **Data collected:**

	total_training_time(s)	average_time/epoch(s)	loss	F1-score on test data
Baseline Model:	236.749160	4.734983	0.076307	0.714286
Model using new_lstmcell::	227.668877	4.553378	0.077976	0.666667

- **Conclusion:** It doesn't affect much on the performance except that it takes shorter to train. In fact the accuracy/loss may be worse than the baseline model.
- **Analysis**
  - loss/Accuracy: We forget when we're going to input, and we input new values to the state when we forget something older. This makes the cell more "balanced". However, it turned out that it didn't help much in this seq2seq task
  - Speed: fundamental arithmetics (1-) is obviously faster than applying a sigmoid function

Q3. get_char_seqence()

1. <u>How do we implement</u>

- **Pseudo code:**
  get_char_sequence(model, batch_char_index_matrices, batch_word_len_lists):

```
minibatch = a batch of words reshaped from batch_char_index_matrices
minibatch_word_len_list = [word length for w in minibatch]    //reshaped from batch_len_list
minibatch_embedding = char_embeds(minibatch)

sort the minibatch
pack_padded_sequence the batch and pass it to the char_LSTM
get the output and the hidden state of the layer Matrix shape [2, word length, hidden_dim]
(2 because of Bidirectional LSTM)
Transpose first and second dimension of the matrix to form [word length, 2, hidden_dim],
and then reshape to form a [word length, hidden_dim * 2] matrix
recover the original order of the hidden state
reshape the hidden state back to [sentece_num, word_num, hidden_dim*2]
```

- **Implementation details:**
  1) Get a minibatch, where each element is a word. Sequence_len in the char_LSTM is the num_of_char_in_a_word
  2) Sort the minibatch, pack_padded it(batch first) to pass the minibatch correctly to the char_LSTM
  3) Use the hidden states to get the desired results we want, instead of the output(we surely could)
     hidden_state is of shape[num_layers*num_directions, batch_size, hidden_dim].
     We concatenate hidden_state[0](normal LSTM) and hidden_state[1](reverse LSTM)along the last dimension, to get a concatenated output of shape [1, batch_size, hidden_dim*2]
  4) Recover the order of the hidden_state of words
  5) Reshape it back to shape [sentece_num, word_num, hidden_dim*2]

2. How does it affect the performance

Apply the test.txt data onto the saved trained model. Compare the performance of the baseline model and of the one employing char_embedding.

- **Data collected:**

	total_training_time(s)	average_time/epoch(s)	loss	F1-score on test data
Baseline Model:	236.749160	4.734983	0.076307	0.714286
Model using char_embedding:	566.061671	11.321233	0.091182	0.571429

- **Conclusion:** It takes longer to train(around 2.5-3 times of the baseline model). The performance didn’t improve either.
- **Analysis**
  - Accuracy: Employing char_embedding is good for out-of-vocabulary(OOV) task.

- Speed: It is slower than the one without employing char_embedding because we have an additional BiLSTM layer.