Machine Learning Report

Code Structure

Common functions like reading files, writing to files, counting are found in `utilities.py` and are shared over all of the part[x].py source code files.

For each of the parts, the relevant functions are found in `part[x].py` where `x` here denotes the specific part.

Note that for part 4, we will still run `part4.py`. However, for the experiment on how we obtain the factor to allow the best F-score you can examine `part4_script.py`

Part 1: Emission Parameters

Estimating Emission Parameters

In part 1, we iterate through the training set, counting all the times a label has led to a particular word Count(y $\rightarrow$ x) being predicted in `utilities.count_tag_words`. We also count the number of times a label has occurred, Count(y) in the training set using `utilities.count_tags`.

In `part1.estimate_emission_parameters_with_unk`, we then divide Count(y $\rightarrow$ x) by Count(y) to obtain e(x|y). We store this emission probability within a dictionary `e_params`, which contains internal dictionaries for the emission probabilities for each label. This is so we can easily access the emission probability of x to y using `e_params[x][y]`.

For each internal dictionary, we add the emission probability for the special token #UNK#, which will be used for words in the test set that are not in the training set. This is done using the provided expression $\frac{k}{Count(y)+k}$.

We also need to count the number of unique words in the training set, so that we can check if the words in the test set are present in the training set, and change it to the #UNK# token if so. This is done using `utilities.get_training_set_words`.

Predicting using Emission Parameters

$$y^* = \arg\max_y e(x|y)$$

`predict_using_emission`, written in `part1.py`, helps us to get the most probable sequence based on the above expression.

```
def predict_using_emission(words, e_params, word_set):
    labels = ['O', 'B-positive', 'B-neutral', 'B-negative', 'I-positive', 'I-neutral', 'I-negative']
    out = []
    for word in words:
        max_label = 'O'
        max_probability = 0
        if word not in word_set:
            # The word is not seen in our test set. We replace it with #UNK#.
            word = "#UNK#"
        for u in labels:
            if word not in e_params[u]:
                # No possible emission to this word from this label so we skip it.
                continue
            current_prob = e_params[u][word]
            if current_prob > max_probability:
                max_label = u
                max_probability = current_prob
        out.append(max_label)
    return out
```

In this function, we loop through `words` which is a list of words comprising one sentence in the test dataset. For each word, we check if the word is not in the seen set of words of the training set. If so, we replace it with the token #UNK#. Then, we check `e_params` for all the possible labels to that word and take the maximum label and probability. When a word is not in `e_params` but is still in the word set, emission from that label is impossible so we continue the loop.

Results

	ES	RU
Entities in Gold Data	255	461
Entities in Prediction	1734	2089
Correct Entity	205	335
Entity Precision	0.1182	0.1604
Entity Recall	0.8039	0.7267
Entity F-Score	0.2061	0.2627
Correct Sentiment	112	136
Sentiment Precision	0.0646	0.0651
Sentiment Recall	0.4392	0.2950
Sentiment F-Score	0.1126	0.1067

Part 2: Transmission Parameters

Estimating Transmission Parameters

The first task we implemented in Part 2 is to estimate the ***Transmission Parameters*** from the training set using MLE with the help of the formula given in the `Project.pdf`. To estimate a transition parameter from a label at position i-1 to position i, denoted by $q(y_i|y_{i-1})$, we count the number of transitions from the label y_{i-1} to y_i and divide it by the count of the label y_{i-1} in the Dataset.

Here we also need to consider the special cases which is the transition from the START label at position 0 to the label y_1 at position 1, indicated by $q(y_1|START)$ as well as the transition from the label at position n to STOP indicated by $q(STOP|y_n)$. Below is the function to find the count of each possible transmission using `count_transmissions` written in part2.py.

```
# Outputs a dictionary {'label1': {'label1': count of label1->label1, 'label2': count of label1->label2...}...}
def count_transmissions(data):
    transmissions = {'START':{}, 'O':{},'B-positive':{},'B-neutral':{},'B-negative':{},'I-positive':{},'I-neutral':{},'I-negative':{}}
    for label in transmissions.keys():
        transmissions[label] = {'O':0,'B-positive':0,'B-neutral':0,'B-negative':0,'I-positive':0,'I-neutral':0,'I-negative':0, 'STOP':
    last_position = 'START'
    for line in data:
        if line =="\n": #if line is a slash n - need to concern, rewrite processing script to include \n
            transmissions[last_position]["STOP"] += 1 #add transmission from last read label to STOP state
            last_position = 'START' #reset state back to start
        else:
            transmissions[last_position][line[1]] += 1 #for transmission from the Label/State at LAST POSITION to the state read from
            last_position = line[1] #move/transit to the next state read from the line
    #transmissions[last_position]['STOP'] += 1 #add +1 for each transmission from label/state read to STOP
    return transmissions
```

The function above is used to count the number of all possible transitions from one label to another. In the function above, we supply the parameter **data** to be the dataset which can be either the ES or the RU dataset. We define a dictionary called **transmissions** which stores the labels as the **keys** and each of the **transmissions[label]** entries is another dictionary storing the number of transmissions counts from that particular label to every other label.

We start off by setting the variable **last_position** to be at "START" and proceed with reading the input dataset line by line. We check if the line is an *empty line* consisting of only the newline character "\n" or not, an empty line also indicates the separation between one document (set of lines) to another:

- If the line is not an empty line, add 1 to the dictionary entry corresponding to the transmission count from the current label (last_position) to the label read from the line (at line[1])
- If the line is an empty line, there will be a transmission from the current label (last position) to the STOP label, and we increase this transmission count by 1. Reset the last_position variable to be at "START" and proceed reading on the next document

At the end of this function, return the dictionary **transmissions** which store the count values of all transmissions from each label to every other label.

The next step is to use the function `estimate_transmission_parameters` to compute the estimated value of the transmission parameters using the MLE approach mentioned before.

```
# Outputs a dictionary {'label1': {'label1': probability of label1->label1, 'label2': probability of label1->label2...}...}
def estimate_transmission_parameters(transmission_count, tags_count):
    transmission_prob = {'START':{}, 'O':{},'B-positive':{},'B-neutral':{},'B-negative':{},'I-positive':{},'I-neutral':{},'I-negative'
    for label in transmission_prob.keys():
        transmission_prob[label] = {'START':0,'O':0,'B-positive':0,'B-neutral':0,'B-negative':0,'I-positive':0,'I-neutral':0,'I-negati
    for label_in, t_counts in transmission_count.items():
        for label_out, count in t_counts.items():
            transmission_prob[label_in][label_out] = count/tags_count[label_in]
    """
    #Special Cases
    for first_label, start_count in transmission_count['START'].items():
        if start_count > 0:
            transmission_prob['START'][first_label] = 1
    """
    return transmission_prob
```

The function takes in the transmission counts obtained using the function `count_transmissions` and the label counts which utilize the function implemented in part 1 called `utilities.count_tags`, we also define a dictionary called `transmission_prob` which will store the values of the transmission parameters for every possible transition from one label to another. To calculate the parameters, we iterate for every label and divide the transmission count from the current label (`label_in`) to the next label (`label_out`) by the count of the current label (`label_in`). Return the dictionary of estimated transmission parameters, `transmission_prob` in the end.

Viterbi Algorithm

We initialize a cache, which is a list (length n+2). Each element is a dictionary with 9 keys, 1 for each label. The elements of the dictionary are of the form [score, parent], which represents the maximum score for that label in this step, and the parent which led to that score. Since we are using log probability, we must initialize the scores as negative infinity (n_inf) as log(0) approaches negative infinity.

We initially faced underflow issues, as the probabilities we were computing were so small that Python was unable to store it in a float, and it just became 0 after a few Viterbi iterations. To deal with underflow, we used log function from python's math library to work with log probabilities instead. This allowed us to add the log probabilities together rather than multiplying increasingly small probabilities, hence avoiding the underflow issue.

This cache is of length n+2 because it includes START, n steps for each of the words in the dataset, and STOP.

```
n_inf = -math.inf
cache = [{'START':[n_inf, None],
  'STOP': [n_inf, None],
  'O':[n_inf, None],
  'B-positive':[n_inf, None],
  'B-neutral':[n_inf, None],
  'B-negative':[n_inf, None],
  'I-positive':[n_inf, None],
  'I-neutral':[n_inf, None],
  'I-negative':[n_inf, None]} for i in range(n+2)]
```

As with the usual Viterbi algorithm, we set `cache[0]['START'][0] = 0` since the probability of getting to the 'START' label in the beginning is always 1, and log(1) = 0.

$$\pi(j+1,u) = max_v\{\pi(j,v) * b_u(x_{j+1}) * a_{v,u}\}$$

We then iterate through each word in the dataset, from j = 0 to n-1. At each step, we find the maximum score on each of the labels (u), from the previous label (v) and the current word x_{j+1} using the above expression. Before computing the probability, we check if any of the transmission or emission probabilities are 0 for that v or x respectively, and the cache, and if so we skip the current combination. We store this maximum score for each of the labels u in the cache, along with the parent v that led to this score.

We repeat a similar process for the n+1 step, checking the score over v to the label "STOP", but without the emission probability and updating the cache respectively.

For backtracking, we do not need to do much processing since we already store the parent nodes in each cache element. We simply iterate backward through the stored parents, which gives us our maximum probability sequence.

Results

	ES	RU
Entities in Gold Data	255	461
Entities in Prediction	551	532
Correct Entity	131	219
Entity Precision	0.2377	0.4117
Entity Recall	0.5137	0.4751
Entity F-Score	0.3251	0.4411
Correct Sentiment	104	144
Sentiment Precision	0.1887	0.2707
Sentiment Recall	0.4078	0.3125
Sentiment F-Score	0.2581	0.2900

Part 3: 5th-Best Output Sequence

Modified Viterbi Algorithm

The algorithm used here is similar to Part 2 with modifications to accommodate the 5th best sequence. We defined a global variable to store the best output sequences up to the current path called `all_viterbi_list`. Its entries will consist of lists corresponding to the output sequences where in each of the lists the first element is the score of the output sequence (initialized as negative infinity) and the second element is the sequence itself (initialized with the 'START' label). This variable will be updated at each step so that our top sequences are up-to-date at all times.

```
    all_viterbi_list = [
        [n_inf,['START']]
    ]
```

We used a similar approach as the previous part which iterates through every position from 0 to n with a special case for the final position and compute the maximum probability score for each label by considering each label in the previous position. This time, we iterate through all the best output sequences stored in `all_viterbi_list` for each current label u, updating the list at every step to include one more label in each sequence and the updated probability of including that label. Since we are iterating through `all_viterbi_list`, we read the v, sequence and last_cached_value from that list:

```
for path in all_viterbi_list:
      v = path[1][-1]
      sequence = path[1][:]
```

```
# If any of the observed probabilities OF PREVIOUS STATE IS  0, we should skip because that is an impossible path
      if (t_params[v][u] == 0):
          continue
      prev_cached_value = path[0]
```

At each step, we keep a new variable `current_sequences` to keep track of the currently seen sequences in this step. We will set `all_viterbi_list = current_sequences` at the end of each step to update the `all_viterbi_list`.

As before, it is possible that no possible paths can be found for the current step. This might occur if the current word is only counted for a label u in the training set that is unreachable from each of the previous labels v as found in `all_viterbi_list`. In this case, `current_sequences` would have length 0 at the end of the step, so we return from the function early. This calls `get_5th_value`, which we will explain in the reverse step, to obtain the 5th best sequence from the previous list of sequences.

```
if len(current_sequences) == 0:
    return get_5th_value(all_viterbi_list, n)
```

We chose to truncate `all_viterbi_list` at a variable k; that is, we keep the top k sequences when the list is greater than size k. This is to improve processing time because the global top 5 sequences should only depend on the top k sequences at each node.

We initially kept k as 5, to keep only the top 5 sequences at each node, but realized that this might result in unreachable paths from those top 5 sequences, and so the 5th best sequence might be a sequence that was ended early. To allow for some of the other paths, we increased k to 50.

```
# Only keep top k number of paths to reduce computation time.
k = 50
if len(all_viterbi_list)>k:
    all_viterbi_list.sort(key = lambda x: float(x[0]),reverse=True)
    all_viterbi_list = all_viterbi_list[:k]
```

When the computation of Viterbi over all steps completes, we can call `get_5th_value` on the `current_sequence` (we skip updating `all_viterbi_list` on the very last step). This function retrieves the 5th best sequence from the Viterbi list. Sometimes, our list has fewer than 5 elements, which means there are fewer than 5 valid paths to the end. In this case, we will take the last output sequence of our list as the 5th best sequence.

It is also possible that this function is called before the last step is reached. For example: when the function terminates early when `len(current_sequence) == 0`, which occurs when there are no valid paths to a node, as mentioned previously. In this case the output will have a length < n+2 since the paths terminate early. We fill in the rest of the labels with 'O' as our default value.

We expect the output at this point to have length n+2 since it is ['START, 'label 1', 'label 2', ..., 'label n', 'STOP]. Returning `output[1:-1]` gives us ['label 1', ... 'label n'] and is our desired 5th best output sequence.

```
def get_5th_value(list, n):
    list.sort(key = lambda x: float(x[0]),reverse=True)

    if len(list) >= 5:
        # Get 5th best output sequence
        output = list[4][1]
    else:
        # Get last output sequence if list is shorter than 5.
        output = list[-1][1]

    if len(output) < n+2:
        # Append 'O' as default value to be returned (as well as STOP)
        output = output + ['O']*(n+1 - len(output)) + ["STOP"]

    return output[1:-1]
```

Results

	ES	RU
Entities in Gold Data	255	461
Entities in Prediction	572	788
Correct Entity	115	217

	ES	RU
Entity Precision	0.2010	0.2754
Entity Recall	0.4510	0.4707
Entity F-Score	0.2781	0.3475
Correct Sentiment	74	108
Sentiment Precision	0.1294	0.1371
Sentiment Recall	0.2902	0.2343
Sentiment F-Score	0.1790	0.1729

Part 4: Design Challenge

Our group did some research to attempt to increase the accuracy of the Viterbi algorithm.

For the purposes of explaining the approach, we will refer to the following code snippets to `part4.py`.

For the purposes of explaining how we obtain the factor, we will use `part4_script.py`

Approach

1. Data Preprocessing (Stop words)

Finding and removing stop words for both RU and ES datasets.

Stop words are a list of frequently used words in the respective languages which can be divided into language, time, location, and numeral-based stop words.

Reasoning: The removal of stop words for long sentences does not change the meaning of the original sentence but improves the performance of the model in terms of time complexity as we run our model on longer word sequences. Stop words only help to define the sentence structure but do not contribute much to defining the context.

a. Our group utilized open-sourced corpus that contains a list of stop words for each of the languages.

   The list of stop words .txt files is shown below:

   - List of Spanish Stop words: *https://github.com/Alir3z4/stop-words/blob/master/spanish.txt*
   - List of Russian Stop words: *https://github.com/Alir3z4/stop-words/blob/master/russian.txt*

b. Our function `read_stopwords` converts the downloaded corpus and converts them to a list for the ES and RU each. For the ES language, our corpus is `ES/stopwords_ES.txt` for the RU language, our corpus is `RU/stopwords_RU.txt`. This stop words list will be further used in the removal of stop words.

c. `remove_stopwords` help to assign the stop words in each of the training sets to the 'O' label by comparing the respective training set and that of the respective stop words set.

```
#remove all the russian and spanish stopwords
def remove_stopwords(dataset,stop_words,symbols_list=[]):
    original_dataset = []
    file_dataset = open(dataset,'r',encoding='utf-8')
    training_set = file_dataset.readlines()
    for line in training_set:
        # if to include \n
        if len(line) == 1:
            original_dataset.append("\n")
        else:
            line = line.rstrip('\n') #split into the "text" and the "tag/label" using line.split()
            line = line.rpartition(' ')
            line = list(line)
            del line[1]
            if line != ['', '']:
                if line[0] in stop_words: #if the word belongs to the list of stopwords or list of symbols, assign the label O
                    line[1] = "O" #assign label O
                elif line[0] in symbols_list:
                    continue
                original_dataset.append(line)
    Edataset = [ele for ele in original_dataset]
    return Edataset
```

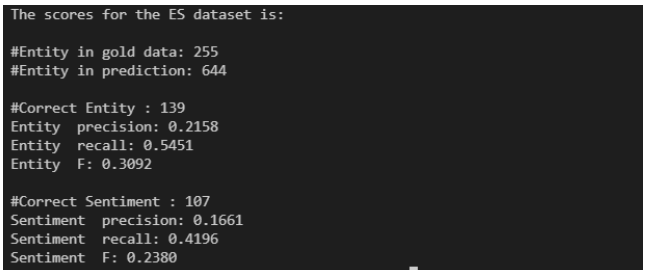
After applying the removal of stop words for both the ES and RU datasets from `ES/dev.in` and `RU/dev.in`, We utilized the processed datasets to estimate the emission parameters, transmission parameters, Viterbi scores, and generate the output files `ES/dev.p4.out` and `RU/dev.p4.out`. We then proceed to evaluate the output files with the gold standard to find the precision, recall, and F-score.

```
#evaluation portion
    print('The scores for the russian dataset is:')
    os.system('python3 EvalScript/evalResult.py RU/dev.out RU/dev.p4.out')
    print('The scores for the ES dataset is:')
    os.system('python3 EvalScript/evalResult.py ES/dev.out ES/dev.p4.out')
```

The table below summarizes the precision, recall, and F scores results are obtained after running the evaluation script.

1b. Stopwords Results

- **ES Dataset**



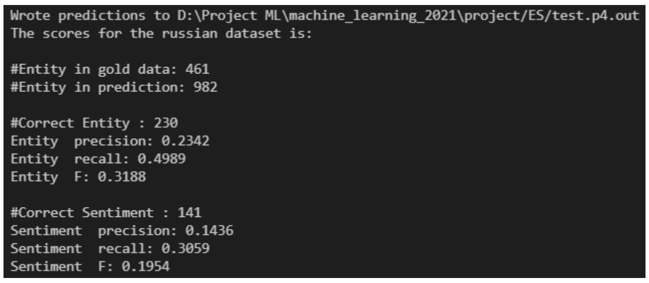
```
The scores for the ES dataset is:

#Entity in gold data: 255
#Entity in prediction: 644

#Correct Entity : 139
Entity  precision: 0.2158
Entity  recall: 0.5451
Entity  F: 0.3092

#Correct Sentiment : 107
Sentiment  precision: 0.1661
Sentiment  recall: 0.4196
Sentiment  F: 0.2380
```

- **RU Dataset**



```
Wrote predictions to D:\Project ML\machine_learning_2021\project/ES/test.p4.out
The scores for the russian dataset is:

#Entity in gold data: 461
#Entity in prediction: 982

#Correct Entity : 230
Entity  precision: 0.2342
Entity  recall: 0.4989
Entity  F: 0.3188

#Correct Sentiment : 141
Sentiment  precision: 0.1436
Sentiment  recall: 0.3059
Sentiment  F: 0.1954
```

	ES	RU
Entities in Gold Data	255	461
Entities in Prediction	644	982
Correct Entity	139	230
Entity Precision	0.2158	0.2342
Entity Recall	0.5451	0.4989
Entity F-Score	0.3092	0.3188
Correct Sentiment	107	141
Sentiment Precision	0.1661	0.1436
Sentiment Recall	0.4196	0.3059
Sentiment F-Score	0.2380	0.1954

From the above results, we can see that there are more correctly predicted entities and sentiment with slightly lower precision and F-scores, and higher recall. This is because assigning the stop words to be "O" labels allows detection of more True Positives but with more False Positives as well since the "O" label now encapsulates more candidate words that are likely to be of this label.

Aside from the stop words, we try to utilize another strategy to enhance the performance which will be described in the next part, the strategy of Label Smoothing. The above results are equivalent to setting the label smoothing factor to be 0.0 which indicates no label smoothing is performed.

2. Label Smoothing

Label smoothing for emission and transition parameters of the Hidden Markov Model

Reasoning: Despite the fact that smoothing is used for multi-class neural networks, we ran the label smoothing on both the emission and transition probabilities. We use smoothing because we want to prevent the overfitting of the model itself, keeping the most probable label to have the highest probability while other labels have a very small probability.

In our `part4.py`, the smoothing of the label is done for the emission and transmission parameters to smooth the probabilities of each label. We applied the following formula for smoothing for the emission and transition parameters respectively for each label. The general formula for Label Smoothing is provided below where we multiply each of the probabilities with the value of 1 minus the smoothing factor and this is followed by adding each of the probabilities by that smoothing factor divided by the length of the array or list of probabilities.

```
# factor is the smoothing factor (alpha)
# labels is an array of probabilities of the labels
labels *= (1 - factor)
labels += (factor / labels.shape[1])
```

where `factor` is used for hyperparameter tuning to the respective Viterbi parameters (transmissions and emissions). Below are the smoothing functions for the transmission and emission probabilities which are built based on the general idea above:

```
def smooth_labels_transmission(transmission_parameters,alpha_sm=0.01):
    smoothed_dict = transmission_parameters
    #alpha_sm = 0.1 #hyperparameter for label smoothing, default 0.1
    for entry in smoothed_dict: #for each dictionary
        for label in smoothed_dict[entry]: #for each label in each dictionary
            if label != 'START':
                smoothed_dict[entry][label] *= (1 - alpha_sm)
                smoothed_dict[entry][label] += (alpha_sm/(len(smoothed_dict[entry])-1)) #add by the smoothing factor/number of labels
            # there is no transition to START at all
    return smoothed_dict #return smoothed dictionary
```

In the above function to smooth the transmission probabilities which takes the collection of transmission probabilities from every label to every other possible label as the input, the general idea stays the same, however since there is no possible transmission from any label to the START label as it is the first label, we excluded this special case and divided the smoothing factor by the length of the dictionary minus 1 which corresponds to the START.

```
def smooth_labels_emission(emission_dict,alpha_sm=0.01):
    smoothed_dict = emission_dict
    for label, word_prob_dict in emission_dict.items():
        for word in word_prob_dict.keys():
            smoothed_dict[label][word] *= (1 - alpha_sm)
            smoothed_dict[label][word] += (alpha_sm/(len(word_prob_dict)))
    return smoothed_dict
```

We found that when the factor is used for hyperparameter tuning, there is some sort of trend where increasing the smoothing factor tends to increase the entity F-score but decreases the sentiments F-score, thus there is some sort of trade-off here.

We supplied different values for alpha_sm which is the last argument to the following function to test out the different values for the smoothing factor starting from the default value 0.0:

```
def run_viterbi(training_path, test_path, output_path,mode,alpha_sm=0.0): #mode ES or RU
    if mode =="ES":
        stopwords_a = read_stopwords(STOP_words_ES_file_path)
    elif mode =="RU":
        stopwords_a = read_stopwords(STOP_words_RU_file_path)
    train = remove_stopwords(training_path,stopwords_a)
    train_words = utilities.get_training_set_words(train)
    test = utilities.read_dev(test_path)
    tags = utilities.count_tags(train)
    tag_words = utilities.count_tag_words(train)

    transmission_counts = part2.count_transmissions(train)
    t_params = part2.estimate_transmission_parameters(transmission_counts, tags)
    t_params = smooth_labels_transmission(t_params,alpha_sm=alpha_sm)
    e_params = part1.estimate_emission_parameters_with_unk(tags, tag_words)
    e_params = smooth_labels_emission(e_params,alpha_sm=alpha_sm)
    prediction = part2.viterbi_loop(test, t_params, e_params, train_words)
    utilities.output_prediction(prediction, test, output_path)
```

We experimented and tuned with various smoothing factors and present the following results for both datasets below:

2b. Label Smoothing Results (Hyperparameter Tuning)

- **ES Dataset**

	Factor = 0.0	Factor = 0.05	Factor = 0.1	Factor = 0.2	Factor = 0.3
Entities in Gold Data	255	255	**255**	255	255
Entities in Prediction	644	686	**721**	832	890
Correct Entity	139	149	**155**	182	195
Entity Precision	0.2158	0.2172	**0.2150**	0.2188	0.2191
Entity Recall	0.5451	0.5843	**0.6078**	0.7137	0.7647
Entity F-Score	0.3092	0.3167	**0.3176**	0.3349	0.3406
Correct Sentiment	107	107	**108**	117	113
Sentiment Precision	0.1661	0.1560	**0.1498**	0.1406	0.1270
Sentiment Recall	0.4196	0.4196	**0.4235**	0.4588	0.4431
Sentiment F-Score	0.2380	0.2274	**0.2213**	0.2153	0.1974

- **RU Dataset**

	Factor = 0.0	Factor = 0.05	Factor = 0.1	Factor = 0.2	Factor = 0.3
Entities in Gold Data	461	461	**461**	461	461
Entities in Prediction	982	1064	**1143**	1244	1304
Correct Entity	230	254	**274**	312	325
Entity Precision	0.2342	0.2387	**0.2397**	0.2508	0.2492
Entity Recall	0.4989	0.5510	**0.5944**	0.6768	0.7050
Entity F-Score	0.3188	0.3331	**0.3416**	0.3660	0.3683
Correct Sentiment	141	145	**147**	146	147
Sentiment Precision	0.1436	0.1363	**0.1286**	0.1174	0.1127
Sentiment Recall	0.3059	0.3145	**0.3189**	0.3167	0.3189
Sentiment F-Score	0.1954	0.1902	**0.1833**	0.1713	0.1666

The above scores show that with increased factors of 0.1 as a step, the entity F-score increases but we notice that the sentiment F-score decreases meaning that our proposed idea is less sensitive as the factors increase. The number of correctly predicted entities and sentiments also increases as the smoothing factor increases, this is also followed by the increase of the precision and recall for the entities as smoothing allows better encapsulation of True Positives. The sentiment precision experiences a slight decrease as the factor increases but the recall shows an increasing trend. Hence, we chose 0.1 as our factor.

For more information about part4, you can look at our jupyter notebook `part_4_implementation.ipynb` to look at how we arrived at these values. However, do run our main script `part4.py` to obtain the final results.

To see how we decided the factor, you can refer to `part4_script.py`

Future work

If given more time and resources, we can work on normalization and lemmatization, as well as other potential strategies such as using bigrams, trigrams, or n-grams which combine words together to form an extra set of feature space and contextualize the words better, which could hopefully improve the F-score.

We could also utilize the mixture of a discriminative model like that of a support vector machine together with a generative model like hmm to gain a better F-score for the sentiment as well as that of the entity. To improve this further, we can also include the part about calculating the loss using stochastic gradient descent to minimize the loss of our proposed method.

We can also run the script on a larger number of epochs to obtain a better F-score.

References

https://www.pyimagesearch.com/2019/12/30/label-smoothing-with-keras-tensorflow-and-deep-learning/

https://www.analyticsvidhya.com/blog/2015/10/6-practices-enhance-performance-text-classification-model/

Contributions

Part 1- Leong Yun Qin Melody and Jerome Heng Hao Xiang

Part 2- Jerome Heng Hao Xiang

Part 3- Jerome Heng Hao Xiang and Leon Tjandra

Part 4- Leon Tjandra and Leong Yun Qin Melody