SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing

Final Project Report

Koe Jia Yee	1003107
Phang Ying Xian Bryan	1003112
Krishna Penukonda	1001781

CONTENT PAGE

Objective

Design a sequence labeling model for informal texts using Conditional Random Fields (CRF) model. The hypothesis is that the discriminative approach will be able to empirically improve the effectiveness of sequence labeling.

Data Given

2 folders (partial and full) that consist of a labelled training set `train`, an unlabelled development set `dev.in` and a labelled development set `dev.out`

Project Setup

In this project, we will be using google colab as it allows the code to be edited real time and setting the runtime to GPU allows the code to be processed faster as compared to the computer's CPU.

A `load_data` function is created to pre-process the data making it cleaner and more consistent to be used throughout the code.

Part 1

For the first part of the project, we are tasked to implement the Hidden Markov Model (HMM) to the dataset provided. HMM is a probabilistic graphical model that predicts a sequence of unknown variables given a set of observed variables. By knowing the joint probability (Generative model) of a sequence of hidden states, the sequence with the highest probability score will be the best possible sequence.

We are required to find our emission and transition probabilities given the data set.

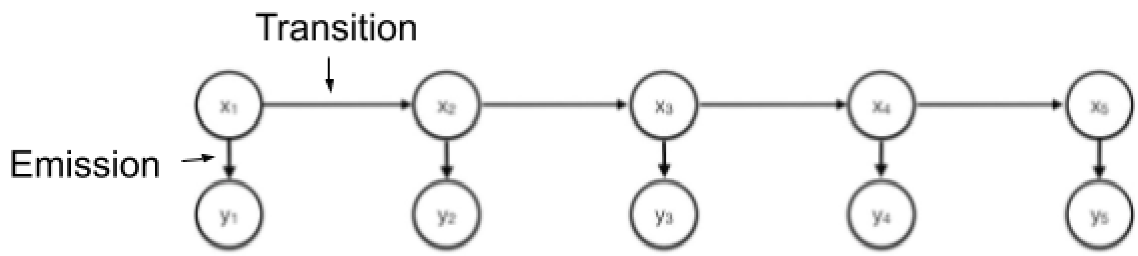

Fig.1 Hidden Markov Model

Emission

For part(i) a `get_emission_scores()` function is created takes in the perimeter of data and using the emission formula to calculate the emission probability scores

$$e(x|y) = \frac{\texttt{Count}(y \to x)}{\texttt{Count}(y)}$$

Fig.2 Emission Formula

and it returns an key-value pair dictionary output of

`dict{str(x,y): int} --- a dict mapping string to emission score`

Transition

For part(ii) a `get_transition_scores()` function is created and takes in the perimeter of data and using the transition formula to calculate transition probability scores

$$q(y_i|y_{i-1}) = \frac{\texttt{Count}(y_{i-1}, y_i)}{\texttt{Count}(y_{i-1})}$$

Fig.3 Transition Formula

and it returns an key-value pair dictionary output of

`dict{str(x,y): int} --- a dict mapping string to emission score`

After getting both emission and transition probability scores, combine them into a feature = dict{emission{str(x,y): int}, transition{str(x,y): int}}

Part 2

In this question, we are required to use the CRF model, CRF uses conditional probability (Discriminant model), it models the dependency between each state and the entire input sequences. As compared to HMM, CRF overcomes the label bias issue.

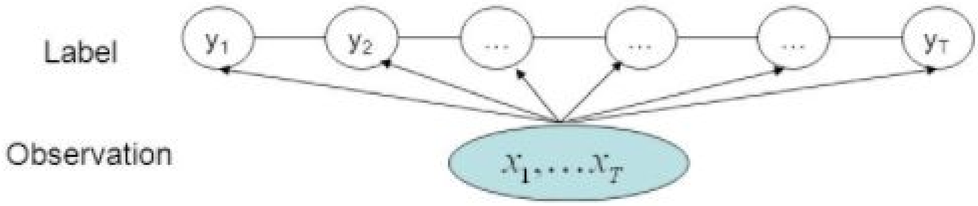

Fig.4 Conditional Random Field

CRF Score

For part(i) A `get_crf_score()` is created accepting the parameters

`x: list[str] --- complete input word sentence`

`y: list[str] --- complete output label sequence`

`f_weights: dict{str(x,y): int} --- a dict mapping feature to weight`
(derived in question 1)

Creating a temporary f_count = defaultdict(int) to store the int value f(x,y), where x = x1,....,xn is the input word sentence and y = y1,......,yn is the output label.

$$\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}, \boldsymbol{y}) = \sum_j w_j f_j(\boldsymbol{x}, \boldsymbol{y})$$

Fig.5 Calculating CRF score

Using the f_weights found in Q1, we sum the f_weights * f_count(x,y) will give the CRF score.

For part(ii), using Viterbi algorithm to find the most probable output sequence y* for a given input sequence x

In a sentence the probability of the next word is multiplied by the probability of the previous word from the <start> to the <end>. Keeping the edge which gives the largest value, and removing the other ones. The optimal path is found by starting from the <end> and tracing backwards to find the arg max value of the states. Hence the highest probability of each state that makes up the sentence will be selected.

Results

Applying Viterbi on the dataset `partial/dev.in` yields the results as seen below, evaluated using `conlleval.evaluate` .

```
processed 2097 tokens with 236 phrases; found: 182 phrases; correct: 135.
accuracy:  55.59%; (non-O)
accuracy:  92.04%; precision:  74.18%; recall:  57.20%; FB1:  64.59
              art: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              eve: precision:   0.00%; recall:   0.00%; FB1:   0.00  0
              geo: precision:  80.28%; recall:  67.06%; FB1:  73.08  71
              gpe: precision:  88.89%; recall:  64.00%; FB1:  74.42  18
              nat: precision:   0.00%; recall:   0.00%; FB1:   0.00  0
              org: precision:  53.85%; recall:  40.00%; FB1:  45.90  26
              per: precision:  73.68%; recall:  43.75%; FB1:  54.90  19
              tim: precision:  72.34%; recall:  64.15%; FB1:  68.00  47
```

Fig.6 Results for CRF score

Part 3

For part(i), we are tasked to find the loss function for CRF

$$-\sum_i \log p(\boldsymbol{y}_i \mid \boldsymbol{x}_i) = -\sum_i \left[\underbrace{\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y}_i)}_{\text{Found in Q2}} - \log \sum_{\boldsymbol{y}'} \underbrace{\exp(\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y}'))}_{\text{Required to find this}} \right]$$

Fig.7 Finding the second term (forward algo) for CRF loss

To find CRF loss, we first have to create a forward algorithm function and then compute the CRF loss function.

Forward Algorithm

First, a forward algorithm is created for the second term with the function `get_forward()` which takes in the perimeter of the

`x: str --- input sentence`

`tags: list[str] --- list of all unique tags (y) from dataset`

`f: dict{str(x,y): int} --- a dict mapping feature to weight` and returns

`scores: np.array --- forward scores`

`alpha: float --- forward score for input sequence`

for each unique tag, initialize the first node which will act as the base step (initial values for i = 1) and the subsequent nodes will act as inductive step (knowing the values of i = k, compute i = k+1)

Inductive step: getting the score of the current word -> next word in the input sentence and updating the score into an input matrix. Each pair-words in the sentence is the summation of the previous scores until it reaches the <EOS>

After reaching the <EOS> it will return a logarithm of sum_score for that input sentence.

Second, to compute the CRF loss passing to the function `compute_crf_loss()`

$$-\sum_i \log p(\boldsymbol{y}_i \mid \boldsymbol{x}_i) = -\sum_i \left[\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y}_i) - \log \sum_{\boldsymbol{y}'} \exp(\mathbf{w} \cdot \mathbf{f}(\boldsymbol{x}_i, \boldsymbol{y}')) \right]$$

Fig.8 CRF loss function

Compute CRF Loss

In this second part, we are finding the summation of the function `compute_crf_loss()` which was derived in Q2 subtracted by the `get_forward()` derived in Q3 given x, y value sequence and returns

`loss: float --- forward score for input sequence`

For part(ii), the objective is to create the backward algorithm, implement a function to calculate the gradients based on the forward and backward scores for each feature and then store it into a dictionary

A function `get_backward()` takes in the parameters:

`x: list[str] --- input sentence`

`tags: list[str] --- list of all unique tags (y) from dataset`

`f: dict{str(x,y): int} --- a dict mapping feature to weight`

and returns

`scores: np.array --- backwards scores`

`beta: float --- backward score for input sequence`

Similarly to the forward algorithm, our backward algorithm will start from the <EOS> , last layer of the given input sentence and it will back propagate to the <START> of the sentence. The `get_backward()` function finds the beta, which is the log of summation given (x, y) emission and transition values.

In the `get_expected_count()` and `get_actual_count()` we used the formula to find the expected and actual count for each feature.

In `get_expected_count()` we used the forward backward algorithm to get the score in relation to the word pair (x,y)

$$E_{p(y|x)}[f_{123}(x_i, y)] = \sum_y p(y|x_i) f_{123}(x_i, y)$$

Fig.9 Expected count formula

The function accepts the perimeter of

`x: list[str] --- input sentence`

`tags: list[str] --- list of all unique tags (y) from dataset`

`f: dict{str(x,y): int} --- a dict mapping feature to weight`

and returns

`f_e_counts: dict --- expected count for each feature.`

The `get_actual_count()` function counts the number of times the feature appears and it accepts the parameters of

`x: list[str] --- input sentence`

`y: list[str] --- list of all unique tags (y) from dataset`

`f: dict{str(x,y): int} --- a dict mapping feature to weight`

and returns

`f_a_counts: dict --- actual count for each feature`

Finally to find the gradient we will use the `compute_gradients()` function by differentiation of the (summation of expectation score - to summation of actual score). The purpose is to observe how much each signal component in the input needs to change to make the network output closer to the label. Updating a gradient value in a dictionary mapping to each word.

$$\frac{\partial L(w)}{\partial \lambda_k} = \sum_i E_{p(y|xi)}[f_k(x_i, y)] - \sum_i f_k(x_i, y_i)$$

Fig.10 Gradient vector formula

In the `compute_gradients()` function accepts the perimeter

`data: list[list[list[word, tag]]] --- data set as list of words with respective tags`

`tags: list[str] --- list of all unique tags (y) from dataset`

`f: dict{str(x,y): int} --- a dict mapping feature to weight`

and returns

`f_gradients: dict{str(x,y): float} --- dict mapping feature to gradient (forward-backward)`

Numerical Check

In the numerical check, we compute the difference between the feature gradient (analytical_gradient) and the crf loss (numerical_gradient) and check if the values are close (<=1e-3)

Part 4

L2 Regularisation term was added to both the `compute_crf_loss()` and `compute_gradients()` functions to control overfitting with the regularisation coefficient , η set to 0.1.

Training

Using the L-BFGS implementation for learning, below is the loss during the training process. The final loss reported is `331.8488504156289`.

```
Loss:945.6148
Loss:871.2612
Loss:798.2791
Loss:741.3911
Loss:665.6300
Loss:544.7417
Loss:460.3143
Loss:431.6958
Loss:415.3659
Loss:401.6317
Loss:376.1860
Loss:354.2691
Loss:341.6425
Loss:330.9837
Loss:326.3098
Loss:323.6951
Loss:317.3421
Loss:312.6864
Loss:309.8190
Loss:306.4859
Loss:301.0462
Loss:300.8695
Loss:301.2570
Loss:301.0523
Loss:299.6738
Loss:297.9056
Loss:296.6562
Loss:295.6590
Loss:295.3306
Loss:294.4702
Loss:294.4606
Loss:294.1119
Loss:293.6087
Loss:293.5047
```

```
Loss:293.5047
Loss:293.1621
Loss:292.9511
Loss:292.7737
Loss:292.9071
Loss:292.6851
Loss:292.8303
Loss:292.9032
Loss:292.8501
Loss:292.8153
Loss:292.7615
Loss:292.6717
Loss:292.5996
Loss:292.5880
Loss:292.5948
Loss:292.6126
Loss:292.6327
Loss:292.6485
Loss:292.6451
Loss:292.6506
Loss:292.6470
Loss:292.6424
Loss:292.6386
Loss:292.6415
Loss:292.6476
Loss:292.6484
Loss:292.6509
Loss:292.6501
Loss:292.6492
Loss:292.6495
Loss:292.6507
Loss:292.6537
Loss:292.6537
Loss:292.6543
```

Fig.11 Loss during training process

```python
print("-------- final loss --------")
print(result[1])
```

```
-------- final loss --------
331.8488504156289
```

Fig.12 Final loss after learning with L-BFGS algorithm

Decoding and Results

With the learned weights, the Viterbi algorithm was used to perform decoding on the development set `partial/dev.in`. The results evaluated using `conlleval.evaluate` are as seen below.

```
processed 2097 tokens with 236 phrases; found: 238 phrases; correct: 122.
accuracy:  54.41%; (non-O)
accuracy:  87.65%; precision:  51.26%; recall:  51.69%; FB1:  51.48
              art: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              eve: precision:   0.00%; recall:   0.00%; FB1:   0.00  0
              geo: precision:  69.74%; recall:  62.35%; FB1:  65.84  76
              gpe: precision:  55.17%; recall:  64.00%; FB1:  59.26  29
              nat: precision:   0.00%; recall:   0.00%; FB1:   0.00  0
              org: precision:  43.48%; recall:  28.57%; FB1:  34.48  23
              per: precision:  44.00%; recall:  34.38%; FB1:  38.60  25
              tim: precision:  38.10%; recall:  60.38%; FB1:  46.72  84
```

Fig.13 Evaluation of Viterbi decoding with learned weights

Part 5

Adding POS features to CRF based on `full/train` dataset. This is implemented by adding another key to the dictionary *f* with the format `emission:{tag}+{POS}` representing the POS feature. The model is then trained on L-BFGS algorithm

Evaluation of the outputs to `full/dev.p5.CRF.f3.out` using `conlleval.evaluate` can be seen below.

```
accuracy:  70.29%; (non-O)
accuracy:  93.75%; precision:  69.36%; recall:  69.07%; FB1:  69.21
              art: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              eve: precision:   0.00%; recall:   0.00%; FB1:   0.00  0
              geo: precision:  75.26%; recall:  85.88%; FB1:  80.22  97
              gpe: precision:  80.95%; recall:  68.00%; FB1:  73.91  21
              nat: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              org: precision:  48.48%; recall:  45.71%; FB1:  47.06  33
              per: precision:  76.67%; recall:  71.88%; FB1:  74.19  30
              tim: precision:  65.38%; recall:  64.15%; FB1:  64.76  52
```

Fig.14 Evaluation of model including POS feature with learned weights

Next, we applied the combined emission feature, combine:y_{i-1} +y_i +x_i to the CRF model, and trained it on the L-BFGS algorithm. Applying Viterbi on the new model gives an output evaluation as seen in Fig 15 below.

```
processed 2097 tokens with 226 phrases; found: 236 phrases; correct: 161.
accuracy:  72.98%; (non-O)
accuracy:  93.90%; precision:  68.22%; recall:  71.24%; FB1:  69.70
              art: precision:   0.00%; recall:   0.00%; FB1:   0.00  3
              eve: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              geo: precision:  85.88%; recall:  73.00%; FB1:  78.92  85
              gpe: precision:  68.00%; recall:  80.95%; FB1:  73.91  25
              nat: precision:   0.00%; recall:   0.00%; FB1:   0.00  2
              org: precision:  42.86%; recall:  51.72%; FB1:  46.88  35
              per: precision:  68.75%; recall:  75.86%; FB1:  72.13  32
              tim: precision:  64.15%; recall:  75.56%; FB1:  69.39  53
```

Fig.15 Evaluation of model including combination emission feature with learned weights

Structured Perceptron

We enumerated through the list of predictions and compared it with the list of correct states. If the prediction was wrong, we updated the weights in *f* to penalize wrongly predicted weights, and added a bonus to the correct weights.

Then, this process is repeated *n* times.

Using the structured perceptron model to find the global optimal sequence, the performance of the model when evaluated on `full/dev.in` is as seen below:

```
Complete prediction for dataset
processed 2097 tokens with 183 phrases; found: 236 phrases; correct: 138.
accuracy:  80.33%; (non-O)
accuracy:  92.66%; precision:  58.47%; recall:  75.41%; FB1:  65.87
              art: precision:   0.00%; recall:   0.00%; FB1:   0.00  3
              eve: precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              geo: precision:  65.88%; recall:  82.35%; FB1:  73.20  85
              gpe: precision:  72.00%; recall:  75.00%; FB1:  73.47  25
              nat: precision:   0.00%; recall:   0.00%; FB1:   0.00  2
              org: precision:  40.00%; recall:  58.33%; FB1:  47.46  35
              per: precision:  50.00%; recall:  72.73%; FB1:  59.26  32
              tim: precision:  64.15%; recall:  77.27%; FB1:  70.10  53
```

Fig.16 Evaluation using structured perceptron with 5-20 epochs