50.040 Nature Language Processing Final Project

Hong Pengfei (1002949)
Lu Jiankun (1002959)
Peng Shanshan (1002974)

August 2020

1 Introduction

In this project, we implemented a Conditional Random Field (CRF) to perform the Named Entity Recognition (NER) task. We also used a few other deep learning methods and compared the results with the State of The Art performances.

2 Part 1

2.0.1 Emission Probability Calculation

Figure 1 is a screenshot of the function to calculate emission probability.

```
def calc_e(x_data, y_data, x_vocab, y_vocab):
    count_emission = Counter([(x,y) for x_instance, y_instance in zip(x_data, y_data) for x, y in zip(x_instance, y_instanc
e)])
    count_label = Counter([oo for o in y_data for oo in o])

    e_score = {}
    for y in y_vocab:
        for x in x_vocab:
            feature = f"emission:{y}+{x}"

            if (x,y) not in count_emission:
                e_score[feature] = ninf
            else:
                score = np.log(count_emission[(x,y)]  /  count_label[y])
                e_score[feature] = score

    return e_score


emission_dict = calc_e(x_data, y_data, x_vocab, y_vocab)
```

Figure 1: Part 1 (i): calculating emission probability

2.0.2 Transition Probability Calculation

Figure 2 is a screenshot of the function to calculate transition probability.

```
def calc_t(y_data, y_vocab):
    count_transition = Counter([ (y_prev, y) for y_instance in y_data for y_prev, y in zip(['START'] + y_instance, y_instance
+ ['STOP'])])
    count_label = Counter([y for y_instance in y_data for y in ['START'] + y_instance])

    f_score = {}
    for y_prev in ['START'] + y_vocab:
        for y in y_vocab + ['STOP']:
            feature = f"transition:{y_prev}+{y}"

            if (y_prev,y) not in count_transition:
                f_score[feature] = ninf
            else:
                score = np.log(count_transition[(y_prev,y)]  /  count_label[y_prev])
                f_score[feature] = score

    return f_score

transition_dict = calc_t(y_data, y_vocab)
feature_dict = {**transition_dict, **emission_dict}
```

Figure 2: Part 1 (ii): calculating transition probability

3 Part 2

3.0.1 Score Calculation

Figure 3 is a screenshot of the function to calculate CRF scores.

```
def compute_score(x_instance, y_instance, feature_dict):
    feature_count = defaultdict(int)

    for x, y in zip(x_instance, y_instance): feature_count[f"emission:{y}+{x}"] += 1

    for y_prev, y in zip(['START'] + y_instance, y_instance + ['STOP']):
        feature_count[f"transition:{y_prev}+{y}"] += 1

    score = sum([feature_dict[feat]*count for feat, count in feature_count.items()])
    return score

x_instance = "This is the second U.N.-Congolese offensive against militias in the region since the DRC   's constitutional ref
erendum a week ago .".split()
y_instance = 'O    O  O   O      O              O         O       O       O  O   O      O     O   B-geo O  O             O
O O    O   O'.split()
compute_score(x_instance, y_instance, feature_dict)
```

```
-139.57175855522826
```

Figure 3: Part II (i): calculating CRF score

3.0.2 Viterbi Decoding

Figure 4 is a screenshot of the function to calculate emission probability.

```python
def viterbi(x_instance, y_vocab, feature_dict):
    n, d = len(x_instance), len(y_vocab)
    scores = np.full( (n,d), -np.inf) # initialize to be very negative
    bp = np.full( (n,d), 0, dtype=np.int)

    for i, y in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:START+{y}",  ninf)
        e_score = feature_dict.get( f"emission:{y}+{x_instance[0]}",  ninf)
        scores[0, i] = t_score + e_score

    for i in range(1, n):
        for y_i, y in enumerate(y_vocab):
            for y_prev_i, y_prev in enumerate(y_vocab):
                t_score = feature_dict.get( f"transition:{y_prev}+{y}", ninf)
                e_score = feature_dict.get( f"emission:{y}+{x_instance[i]}", ninf)
                score = t_score + e_score + scores[i-1, y_prev_i]
                if score > scores[i, y_i]:
                    scores[i, y_i] = score
                    bp[i, y_i] = y_prev_i

    final_score, final_bp = ninf, 0
    for i, y_prev in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:{y_prev}+STOP", ninf)
        score = t_score + scores[n-1, i]
        if score > final_score:
            final_score = score
            final_bp = i
    decoded_sequence = [ y_vocab[final_bp], ]
    for i in range(n-1, 0, -1):
        final_bp = bp[i, final_bp]
        decoded_sequence = [ y_vocab[final_bp] ] + decoded_sequence

    return decoded_sequence

print("y_pred: ", " ".join(viterbi(x_instance, y_vocab, feature_dict)))
print("y_labl: ", " ".join(y_instance))
```

```
y_pred:  O O O O O O O O O O O O O B-geo O O O O O O O
y_labl:  O O O O O O O O O O O O O B-geo O O O O O O O
```

Figure 4: Part II (ii): Viterbi Decoding

4 Part 3

4.0.1 Loss Calculation Based on the Forward Algorithm

Figure 5 is a screenshot of our function to calculate loss based on the forward algorithm.

```
def logsumexp(a):
    b = a.max()
    return  b + np.log( (np.exp(a-b)).sum() )

def forward(x_instance, y_vocab, feature_dict):
    n, d = len(x_instance), len(y_vocab)
    scores = np.zeros( (n,d) )

    for i, y in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:START+{y}", ninf)
        scores[0, i] = t_score

    for i in range(1, n):
        for y_i, y in enumerate(y_vocab):
            temp = []
            for y_prev_i, y_prev in enumerate(y_vocab):
                t_score = feature_dict.get( f"transition:{y_prev}+{y}", ninf)
                e_score = feature_dict.get( f"emission:{y_prev}+{x_instance[i-1]}", ninf)
                temp.append(e_score + t_score + scores[i-1, y_prev_i])
            scores[i, y_i] = logsumexp(np.array(temp))

    temp = []
    for i, y_prev in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:{y_prev}+STOP", ninf)
        e_score = feature_dict.get( f"emission:{y_prev}+{x_instance[-1]}", ninf)
        temp.append(e_score + t_score + scores[-1, i])
    alpha = logsumexp(np.array(temp))

    return scores, alpha


def loss_fn_instance(x_instance, y_instance, feature_dict, y_vocab):
    first_term = compute_score(x_instance, y_instance, feature_dict)
    _, forward_score = forward(x_instance, y_vocab, feature_dict)
    return forward_score - first_term


loss_fn_instance(x_instance, y_instance, feature_dict, y_vocab)
```

```
1.3302877199753311
```

Figure 5: Part II (ii): Forward Algorithm Loss Calculation

4.0.2 The Backward Algorithm and Gradient Calculation

Figure 6, 7, 8 are the screenshots of our function for the backward algorithm and calculate gradient.

5 Part 4

5.0.1 Gradient and Loss Calculation

For code, please refer to main.ipynb.
Here are the intermediate and final losses in the training process:

```python
def backward(x_instance, y_vocab, feature_dict, aggreg_fn=logsumexp):
    n, d = len(x_instance), len(y_vocab)
    scores = np.zeros( (n,d) )

    for i, y in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:{y}+STOP", ninf)
        e_score = feature_dict.get( f"emission:{y}+{x_instance[-1]}", ninf)
        scores[-1, i] = t_score + e_score

    for i in range(n-1, 0, -1):
        for y_i, y in enumerate(y_vocab):
            temp = []
            for y_next_i, y_next in enumerate(y_vocab):
                t_score = feature_dict.get( f"transition:{y}+{y_next}", ninf)
                e_score = feature_dict.get( f"emission:{y}+{x_instance[i-1]}")
                temp.append(e_score + t_score + scores[i, y_next_i])
            scores[i-1, y_i] = aggreg_fn(np.array(temp))

    temp = []
    for i, y_next in enumerate(y_vocab):
        t_score = feature_dict.get( f"transition:START+{y_next}")
        temp.append(t_score + scores[0, i])
    beta = aggreg_fn(np.array(temp))

    return scores, beta
```

Figure 6: Part III (ii) a: The Backward Algorithm Method

```python
def forward_backward(x_instance, y_vocab, feature_dict):
    n, d = len(x_instance), len(y_vocab)
    f_scores, alpha = forward(x_instance, y_vocab, feature_dict)
    b_scores, beta = backward(x_instance, y_vocab, feature_dict)

    feature_expected_count = defaultdict(float)

    for i in range(n):
        for y_i, y in enumerate(y_vocab):
            e_feature = f"emission:{y}+{x_instance[i]}"
            feature_expected_count[e_feature] += np.exp(f_scores[i, y_i] + b_scores[i, y_i] - alpha)

    for i, y_next in enumerate(y_vocab):
        t_feature = f"transition:START+{y_next}"
        feature_expected_count[t_feature] += np.exp(f_scores[0, i] + b_scores[0, i] - alpha)

        t_feature = f"transition:{y_next}+STOP"
        feature_expected_count[t_feature] += np.exp(f_scores[-1, i] + b_scores[-1, i] - alpha)

    for y_i, y in enumerate(y_vocab):
        for y_next_i, y_next in enumerate(y_vocab):
            t_feature = f"transition:{y}+{y_next}"
            t_score = feature_dict.get(t_feature, ninf)
            total = 0
            for i in range(n-1):
                e_score = feature_dict.get(f"emission:{y}+{x_instance[i]}", ninf)
                total += np.exp(f_scores[i, y_i] + b_scores[i+1, y_next_i] + t_score + e_score - alpha)
            feature_expected_count[t_feature] = total

    return feature_expected_count
```

Figure 7: Part III (ii) b: Expected Feature Score Calculation using Forward Backward Algorithm

```python
def get_feature_count(x_instance, y_instance, feature_dict):
    feature_count = defaultdict(int)

    for x, y in zip(x_instance, y_instance): feature_count[f"emission:{y}+{x}"] += 1

    for y_prev, y in zip(['START'] + y_instance, y_instance + ['STOP']):
        feature_count[f"transition:{y_prev}+{y}"] += 1

    return feature_count
```

Figure 8: Part III (ii) c: A Wrapper Function for Expected Feature Score Calculation

```
['18333.7955', '14132.2391', '13066.3711', '12663.9164', '12303.2443',
'11099.0803', '10489.4525', '9614.8077', '9067.4347', '8394.3281',
'8004.4294', '7673.0057', '7275.4810', '6911.1102', '6710.6507',
'6469.9394', '6113.1164', '6011.9983', '5886.4793', '5688.9170',
'5620.4006', '5526.2114', '5410.1797', '5263.5436', '5010.8256',
'4891.2793', '4772.6344', '4602.0067', '4504.9576', '4438.6535',
'4382.5379', '4312.2697', '4146.8862', '3931.2384', '3877.3722',
'3638.9999', '3547.3763', '3424.4592', '3289.2315', '3263.2814',
'3149.6003', '3129.3778', '3088.5620', '3031.8828', '2977.3570',
'2917.4135', '2890.3171', '2833.4111', '2783.4258', '2686.1813',
'2643.4129', '2600.7354', '2583.4919', '2549.4932', '2524.9203',
'2502.2955', '2495.3878', '2489.2714', '2473.9431', '2450.0810',
'2422.6368', '2398.3073', '2384.3053', '2377.2105', '2363.4270',
'2350.8484', '2339.7542', '2333.7093', '2324.2190', '2316.9614',
'2313.5039', '2309.5335', '2307.8828', '2301.3470', '2294.1057',
'2283.8769', '2280.3981', '2274.8810', '2271.5517', '2268.9940',
'2267.8335', '2265.7117', '2263.7747', '2262.8404', '2260.4822',
'2259.0091', '2255.6376', '2253.7486', '2252.5612', '2251.1748',
'2248.1788', '2247.5830', '2245.5654', '2244.8452', '2244.0129',
'2242.3143', '2240.6339', '2239.0969', '2238.5592', '2237.7739',
'2237.3731', '2237.0004', '2236.5396', '2235.7711', '2235.6152',
'2234.9615', '2234.5293', '2233.8771', '2233.1602', '2233.0260',
'2232.7104', '2232.4093', '2232.1178', '2231.7976', '2231.6077',
'2231.2559', '2231.1797', '2231.0717', '2230.8816', '2230.5278',
'2230.4104', '2230.2654', '2230.1896', '2230.1254', '2230.0949',
'2230.0348', '2229.9479', '2229.8907', '2229.8442', '2229.7602',
'2229.6432', '2229.4902', '2229.3438', '2229.3107', '2229.2153',
'2229.1924', '2229.1299', '2229.0838', '2229.0369', '2229.0219',
'2228.9987', '2228.9950', '2228.9663', '2228.9556', '2228.9346',
'2228.9000', '2228.8802', '2228.8474', '2228.8278', '2228.8128',
'2228.8048', '2228.7845', '2228.7662', '2228.7509', '2228.7365',
'2228.7296', '2228.7051', '2228.6981', '2228.6847', '2228.6779',
'2228.6640', '2228.6496', '2228.6362', '2228.6342', '2228.6308',
'2228.6273', '2228.6192', '2228.6082', '2228.6036', '2228.5870',
```

```
'2228.5838', '2228.5806', '2228.5778', '2228.5735', '2228.5719',
'2228.5692', '2228.5684', '2228.5673', '2228.5623', '2228.5580',
'2228.5531', '2228.5505', '2228.5480', '2228.5448', '2228.5420',
'2228.5413', '2228.5402', '2228.5399', '2228.5392', '2228.5383',
'2228.5370', '2228.5367', '2228.5347', '2228.5340', '2228.5334',
'2228.5326', '2228.5307', '2228.5303', '2228.5294', '2228.5288',
'2228.5282', '2228.5276', '2228.5266', '2228.5263', '2228.5258',
'2228.5255', '2228.5251', '2228.5246', '2228.5241', '2228.5240',
'2228.5237', '2228.5236', '2228.5235', '2228.5232', '2228.5229',
'2228.5228', '2228.5226', '2228.5224', '2228.5223', '2228.5221',
'2228.5220', '2228.5218', '2228.5217', '2228.5216', '2228.5216',
'2228.5215', '2228.5214', '2228.5213', '2228.5212', '2228.5212',
'2228.5211', '2228.5211', '2228.5211', '2228.5210', '2228.5210',
'2228.5210', '2228.5209', '2228.5209', '2228.5208', '2228.5208',
'2228.5208', '2228.5207', '2228.5207', '2228.5207', '2228.5207',
'2228.5207', '2228.5206', '2228.5206', '2228.5206', '2228.5206',
'2228.5206', '2228.5206', '2228.5206', '2228.5205', '2228.5205',
'2228.5205', '2228.5205']
```


Figure 9 visualizes the loss during training. The loss stably goes down from 18333.7955 to 2228.5205. The final loss is 2228.5205.

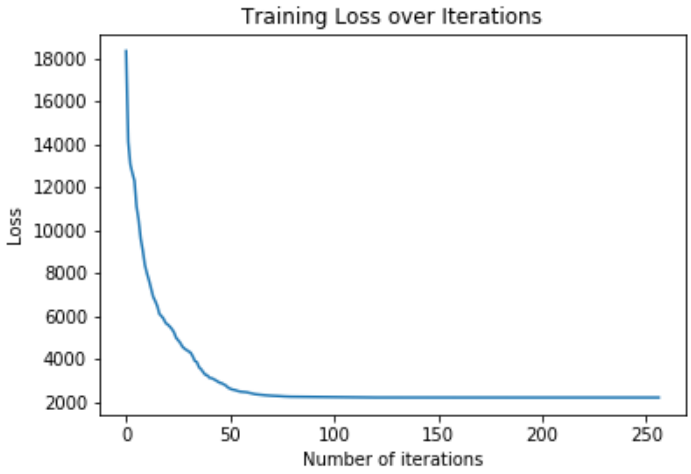

Figure 9: Part IV (I): CRF Training Loss over Iterations

5.1 Viterbi Decoding and Development Set Evaluation

For code, please refer to main.ipynb. We evaluated our model on the development set. The performance is shown in Table 1.

F_1	Precision	Recall
55.12	54.32	55.93

Table 1: Evaluating CRF Model on Development Set

F_1	Precision	Recall
prec 70.769	73.516	68.220

Table 2: Evaluating CRF with POS Tags on Development Set

6 Part 5

6.0.1 Adding POS Tags as Features

For code, please refer to $5-1.py$. We evaluated our model with POS tags as features on the development set. The performance is shown in Table 2.

6.0.2 Combining Transition and Emission Features

For code, please refer to $5-2.py$. We evaluated our model with combined transition and emission probabilities as features on the development set. The performance is shown in Table 3.

6.0.3 Structured Perceptron

For code, please refer to $p5-3.py$. We evaluated our Structured Perceptron model on the development set. The performance is shown in Table 4.

7 Part 6

7.1 Customized Features for CRF

Our used

7.2 Customized Model Design

7.2.1 BERT

Model. The BERT model was proposed in BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding by Jacob Devlin, Ming-Wei

F_1	Precision	Recall
71.24	72.17	70.34

Table 3: Evaluating CRF with Combined Transition and Emission Features on Development Set

F_1	Precision	Recall
70.769	73.516	68.220

Table 4: Evaluating Structured Perceptron on Development Set

Chang, Kenton Lee and Kristina Toutanova. It's a bidirectional transformer pre-trained using a combination of masked language modeling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia. The pre-trained BERT model was fine tuned with one additional output layer to deal with the NER task in this project. We used a Bert Model with a token classification head on top (a linear layer on top of the hidden-states output). We used a Bert Model with a token classification head on top (a linear layer on top of the hidden-states output).

Hyperparameters. For our experiments, we use a bert base model with mini-batch size of 32, a learning rate of 2e-5, a dropout probabiolity of 0.1 for all fully connected layers in the embeddings, encoder, and pooler, a dropout ratio of 0.1 for the attention probabilities, and train using Adam with a weight decay ratio of 0.01. This gives us a F1 score of 0.698. We did not further tune BERT as the performance is much lower than the otehr two models.

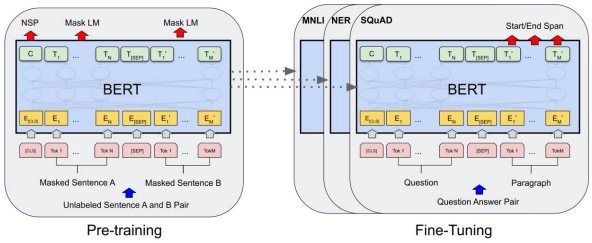

Figure 10: Overall pretraining and fine-tuning procedure for BERT

7.2.2 Flair Embedding + LSTM + CRF

Model. Flair is proposed by **?**. In this model, we dynamically aggregate contextualized embeddings of each unique string that we encounter. We then use a pooling operation to distill a 'global' word representation from all contextualized instances. We use these 'pooled contextualized embeddings' on common named entity recognition (NER) tasks with the standard BiLSTM-CRF sequence labeling architecture .

Hyperparameters. For our experiments, we use an LSTM one layer, a locked dropout value of 0.5, a word dropout of 0.05, and train using SGD with an annealing rate of 0.5 and a patience of 3. We perform model selection over the

LSTM hidden size	batch size	learning rate	F_1
256	16	0.5	0.760
512	16	0.5	0.759
256	32	0.5	0.762
512	32	0.5	0.759
256	64	0.5	0.766
512	64	0.5	0.769
256	16	0.1	0.746
512	16	0.1	0.756
256	32	0.1	0.760
512	32	0.1	0.739
256	64	0.1	0.771
512	64	0.1	0.768
256	16	0.05	0.764
512	16	0.05	0.757
256	32	0.05	0.771
512	32	0.05	0.772
256	64	0.05	0.770
512	64	0.05	**0.774**

Table 5: Hyper parameter tuning results for Flair + LSTM-CRF

LSTM hidden size $\in \{256, 512\}$, the learning rate $\in \{0.05, 0.1, 0.5\}$ and mini-batch size $\in \{16, 32, 64\}$, choosing the model with the best F-measure on the validation set. We then repeat the experiment 5 times with different random seeds, and train using both train and development set, reporting both average performance and standard deviation over these runs on the test set as final performance

Standard Word Embeddings. The paper recommends contextual string embeddings to be used in combination with standard word embeddings. We use GLOVE embeddings together with flair embedding in our experiments.

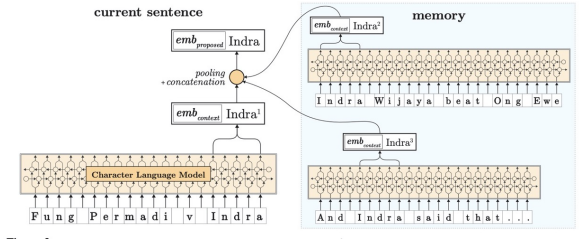

Figure 11: Example of how flair embedding is generated.

7.2.3 Iterated Dilated CNN

Proposed by Emma Strubell, et.al (2018), the Dilated CNN method has good capacity for large context and structured prediction. It permits fixed-depth convolutions to run in parallel across entire documents. It is able to achieve the SOTA performance with 8 times faster speed.

Model Architecture The model takes in a sequence of T tokens as input and predicts a sequence of T NER labels as output. The input tokens are first run through an embedding layer to obtain a sequence of word vectors $\boldsymbol{x_t}$. After word embedding is a dilation-1 convolution layer, followed by L_B dilated convolution blocks, each block constituting L_C dilated convolution layers with exponentially increasing dilation width and a final dilation-1 layer. After that, a simple linear transformation is applied to obtain the per class scores.

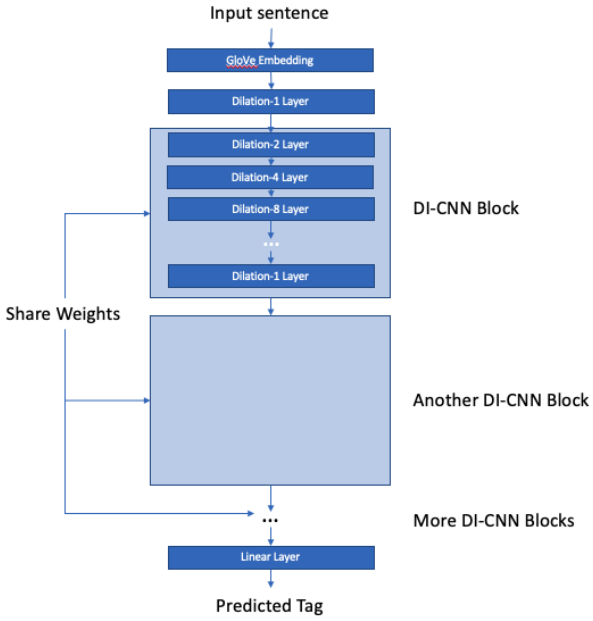

Figure 12: Model Architecture of Dilated-CNN

Loss Function The model is trained with a maximum likelihood method. The loss function is the averaged binary cross entropy loss across prediction after each dilated convolution block B:

$$Loss = \frac{1}{L_b}\sum_{k=1}^{L_b}\frac{1}{T}\sum_{t=1}^{T} logP(y_t|\boldsymbol{h_t}^{(L_b)}$$

Experiments We experimented with different batch sizes.

	Batch Size	F_1	Precision	Recall
0	16	71.24	72.17	70.34
1	32	73.73.92	73.73	73.73
2	40	73.32	75.11	71.61

Table 6: Hyper parameter tuning results for Dilated CNN

Models	F_1	Precision	Recall
CRF Part5	0.71	0.74	0.68
CRF Part6	0.6	0.6	0.6
BERT	0.698	0.723	0.674
Flair + LSTM-CRF	**0.774**	**0.795**	0.754
Iterated Dilated CNN	0.771	0.779	**0.763**

Table 7: NER results (F1, Precision, Recall) for different models. **Bold** indicates best result.

7.2.4 Results

Table 7 shows the results of our 3 different models.

8 Instructions to Run Code

Please refer to the readme file in our github repo.