CS 533: Natural Language Processing	**(Due: 03/31/20)**

Assignment 4

Name: Twisha Naik, *NetID:* tn268 *Students discussed with:*

Problem 1: HMM ((1 + 6 + 6 = 13 points))

1. Emission probabilities $o(x|y)$:

$o(x\|y)$	Emission Probability
$o(the\|D)$	1.0
$o(cut\|N)$	0.167
$o(man\|N)$	0.33
$o(saw\|N)$	0.33
$o(the\|N)$	0.167
$o(cut\|V)$	0.5
$o(saw\|V)$	0.5

Transition probabilities $t(x|y)$:

$t(y^{'}\|y)$	Transition probability
$t(D\|*)$	0.67
$t(N\|*)$	0.33
$t(D\|V)$	1.0
$t(N\|D)$	1.0
$t(V\|N)$	0.33
$t(N\|N)$	0.167
$t(STOP\|N)$	0.5

Non-zero emission probabilities for the word **cut** are:

- $o(cut|N) = 0.167$
- $o(cut|V) = 0.5$

2. Probability under the HMM that the **third word is tagged with V** conditioning on x(2) = "the saw cut the man" is as follows:

$$\sum_{(y1,y2,y3,y4,y5)\in Y:y3=V} p(y1, y2, y3, y4, y5|\text{the saw cut the man})$$
$$= alpha(3, V) * beta(3, V)$$
$$= 0.03755 * 0.1667$$
$$= 0.00626$$

3. Probability that the **fifth word is tagged with N** conditioning on x(1) = "the man saw the cut" is as follows:

$$\sum_{(y1,y2,y3,y4,y5)\in Y:y5=N} p(y1, y2, y3, y4, y5|\text{the man saw the cut})$$
$$= alpha(5, N) * beta(5, N)$$
$$= 0.00626 * 0.5$$
$$= 0.00313$$

Problem 2: PCFG ((1 + 6 + 6 = 13 points))

1. MLE parameter values of (u, b) estimated from the given corpus.
   Unary Rule Probabilities

$u(x\|y)$	Probability
$u(the\|D)$	0.67
$u(a\|D)$	0.33
$u(saw\|V)$	1.0
$u(with\|P)$	1.0
$u(boy\|N)$	0.33
$u(man\|N)$	0.33
$u(telescope\|N)$	0.33

   Binary Rule Probabilities

$b(X \rightarrow YZ)$	Probability
$b(S \rightarrow NP, VP)$	1.0
$b(PP \rightarrow P, NP)$	1.0
$b(NP \rightarrow D, N)$	0.857
$b(NP \rightarrow NP, PP)$	0.143
$b(VP \rightarrow VP, PP)$	0.33
$b(VP \rightarrow V, NP)$	0.67

2. Probability under the PCFG that **NP spans (4, 8)** (i.e., "the man with a telescope") conditioning on x is as follows:

$$\sum_{\tau \in GEN(x):root(\tau,4,8)=NP} p(\tau|x)$$
$$= alpha(4, 8, NP) * beta(4, 8, NP)$$
$$= 0.00259 * 0.12698$$
$$= 0.000329$$

3. Probability under the PCFG that **VP spans (3, 5)** (i.e., "saw the man") conditioning on x is as follows:

$$\sum_{\tau \in GEN(x):root(\tau,3,5)=VP} p(\tau|x)$$
$$= alpha(3, 5, VP) * beta(3, 5, VP)$$
$$= 0.12698 * 0.00605$$
$$= 0.000768$$

Problem 3: CRF ((1 + 1 + 1 + 1 + 1 + 1 + 8 + 8 = 22 points))

1. Understanding the code for class "TaggingDataset", the explanation of the program is as follows.

   (a) The word sequences are sorted in descending order of their length because the function batchfy uses the sequence length to generate batches with sequences of same size.

   (b) No, not every batch size will contain N sequences. The batch size (N) ensures that each batch will have a maximum of N sequences. As we want each batch to have sequences with same length, if the length of new sequence is less than previous sequence length ($length < prev\_length$) the batch size will be less than N.

   (c) No, there is no padding at the word level. This is because, the batch has same length word sequences discarding the need of padding.

   (d) The characters in each batch are stored into a list called *cseqs*. Each item in *cseqs* list consists of torch.LongTensors representing a word's characters converted to the corresponding index.

```
    cseqs = [torch.LongTensor([self.char2c[c]
            for c in word if c in self.char2c])  # Skip unknown
            for word in wordseqs[i]]  # Use original words

```

   Once we get the list of cseqs in cseqslist, they are all flattened into a single list.

```
    flattened_cseqs = [item for sublist in cseqslist
                       for item in sublist]  # List of BT tensors of varying lengths

```

   (e) Yes, there is padding at the character level.

```
    C = pad_sequence(flattened_cseqs, padding_value=self.PAD_ind,
    batch_first=True).to(self.device)  # BT x T_char

```

2. After careful observation of ***evaluate*** method in BiLSTMTagger, following are the answers.

   - **Accuracy**
     The number of correct predictions is divided by the total number of predictions. The number of predictions are calculated as follows:

```
    # Total Predictions
    num_preds += B * T
    # Number of correct predictions
    num_correct += (preds == Y).sum().item()

```

     Once, we get the numbers, a new key called "acc" is added in the output dictionary.

```
    output = {'acc': num_correct / num_preds * 100}

```

   - **F1 score**
     F1 score is calculated only if the labels have BIO format.
     If the tag 'O' is present in tag sequence Y, the values of true-positve (tp), false-negatives(fn) and false-positives(fp) are calculated. To compute this, the sequences are divided into chunks pertaining to a single entity using the ***'get_boundaries'*** function. Whenever any entity matches in the gold and prediction boundary, $tp['<all>']$ count is incremented. Similarly, necessary conditions are checked to increment $tn['<all>']$, $fn['<all>']$ and $fp['<all>']$ counts.
     Using these values, precision and recall scores are calculated and using that F1 score is computed and stored in **output**$['<all>']$.

```
    f1_denom = p_e + r_e
    f1_e = 2 * p_e * r_e / f1_denom if f1_denom > 0 else 0
    output['f1_%s' % e] = f1_e

```

3. Explanation of greedy tagging loss in crf.py.

- Loss for a single data point which is the Cross Entropy Loss:

$$\text{loss}(x^{(i)}, h^{(i)}, y^{(i)}) = -\log\left(\frac{\exp(h^{(i)}[y^{(i)}])}{\sum_{j=1}^{L}\exp(h^{(i)}[(j)])}\right)$$

- Greedy Loss: Average the loss over all examples in the batch.

$$\text{greedy\_loss} = \frac{\sum_{i \in Batch} \text{loss}(x^{(i)}, h^{(i)}, y^{(i)})}{\text{batch\_size}}$$

4. Incorporation of character-level information in the tagger.

   (a) Character embedding **cemb** is produced using *nn.Embedding* which takes the number of unique characters and the character emdedding dimension *dim_char* as parameters.
   The characters of the word are sent to the **BiLSTMOverCharacters** which is initialised using the character-level embedding cemb.
   This character embedding is concatenated with the word embedding and in this way character-level information is also passed to the tagger.

   (b) The character embeddings produced through the character-lstm described in previous question, is concatenated with the word embeddings.
   Thus, final dimension is **wdim + 2*cdim**.

5. Explanation of CRFLoss in crf.py.

   (a) The parameters of this class are:
   - self.start : A tensor of length L which denotes the dummy start token.
   - self.T : A tensor of size (LxL) which represents the transition of one label to another.
   - self.end : A tensor of length L which denotes the dummy end token.

   NOTE: L = number of label types

   (b) Forumla for the score_targets:
   Final score is the summation of the scores for the predicted labels and the internal transitions that led to the generation of the sequence.

$$\text{score}(h, y) = \sum_{i=1}^{T} h_i[y_i] + \text{self.start}[y_0] + \sum_{i=2}^{T-1} \text{self.T}[y_i, y_{i-1}] + \text{self.end}[y_T]$$

   (c) Code for loss:

```
    normalizers = self.compute_normalizers(scores)
    target_scores = self.score_targets(scores, targets)
    loss = (normalizers - target_scores).mean()

```

   Equation for loss:

$$\text{loss} = Mean(\text{normalizers} - \text{target\_scores})$$

   Here, target_scores = the score computed using the score_targets function described in the previous question.

6. Computation of normalizers and decoder.

   (a)
   - **compute_normalizers_brutes**: Calculates sequence scores for all possible target sequences and computes a final normalised score by taking the log of summation of each of these scores raised to the power of e.
   In other words, for normalization: Take exponent of each of the scores, add them and take log of that summation.
   - **compute_decode_brute**: Iterate over all possible combinations of target sequences of labels and calculates the score for each of the sequence. Then it returns the maximum score produced by a target sequence along with the corresponding sequence.

(b) Both the functions iterate over all possible target sequences = L^{T}
For each of the target sequence it computes the score by iterating each tag in the sequence = O(T)
Thus, time complexity = $O(T * L^T)$.

7. Implementation of **compute_normalizers** with complexity $O(|L|^2T)$

```
def compute_normalizers(self, scores):
    B, T, L = scores.size()
    scores = scores.transpose(0, 1) # Make scores: T x B x L

    # Initialise prev with the first token of each sequence by appending start token to it
    prev = self.start + scores[0]  # TODO (B x L)
    for i in range(1, T):
        prev = torch.logsumexp(prev.unsqueeze(2) + self.T.transpose(0, 1) + scores[i].unsqueeze(1), dim=1)  # TODO: implement only using prev (no new definition)

    # Append the end token to the end in each sequence
    prev += self.end
    normalizers = torch.logsumexp(prev, dim=1)  # TODO (B)
    return normalizers
```

8. Implementation of **decode** with complexity $O(|L|^2T)$

```
def decode(self, scores):  # B x T x L
    B, T, L = scores.size()
    scores = scores.transpose(0, 1)
    prev = self.start + scores[0]  # TODO (B x L)
    back = []
    for i in range(1, T):
        prev, indices = (prev.unsqueeze(2) + self.T.transpose(0, 1) + scores[i].unsqueeze(1)).max(dim=1)  # TODO (indices: B x L)
        back.append(indices)
    prev += self.end

    max_scores, indices = prev.max(dim=1)  # TODO (indices: B)
    tape = [indices]
    back = list(reversed(back))
    for i in range(T - 1):
        indices = back[i].gather(1, indices.unsqueeze(1)).squeeze(1)  # TODO
        tape.append(indices)
    return max_scores, torch.stack(tape[::-1], dim=1)
```

```
Twishas-MacBook-Pro:tn268_Assignment4 twishanaik$ python3 test_crf.py
...
----------------------------------------------------------------------
Ran 3 tests in 1.121s

OK
```

Figure 1: Passed the test cases in test_crf file