

HW4: Word Segmentation

Virgile Audi
vaudi@g.harvard.edu
Nicolas Drizard
nicolasdrizard@g.harvard.edu
Github: virgodi/cs287/HW5

April 17, 2016

1 Introduction

The goal of this assignment is to tackle the NLP task of identifying and labeling contiguous segments of text. We will use sequence models and a dynamic programming method to find the best scoring sequence.

2 Problem Description

The idea is here to label continuous sequence of words with BIO tagging of different entities. The entities are the following:

1. PER: a person
2. LOC: a location
3. ORG: an organization
4. MISC:

Furthermore, this tagging method identifies the continuous group of words belonging to the same entity: the prefix B stop the current tag and begins a new one whereas the prefix I continues adding to the previous tag. However, in our solution we just cared about predicting the entity tag and then we were grouping the contiguous predictions into the same entity because the training text does not contain any B-tag.

3 Model and Algorithms

We used three different methods to solve this problem. The first two are the equivalent of first the Naive Bayes and second the logistic regression from text classification tasks. The last one introduces a customized way to train a neural architecture for this task.

3.1 Hidden Markov Model

We implement here a standard first order hidden Markov Model. The hidden states are the tags and the observed states are the features we built (word counts, capitalization...). The model can be represented with the following graphical model and requires two distributions: emission and transition.

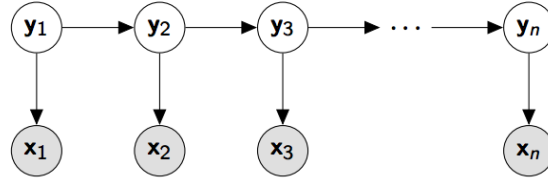


Figure 1: Graphical model of 1st order HMM with one feature

We represent the two distributions with multinomial as they model feature counts. As a result, we can infer them simply with the maximum likelihood estimator:

$$p(x_i = \delta(f) | y_i = \delta(c)) = \frac{F_{f,c}}{F_{.,c}}$$

$$p(y_i = \delta(c_i) | y_{i-1} = \delta(c_{i-1})) = \frac{T_{c_{i-1},c_i}}{T_{c_{i-1},.}}$$

with T_{c_{i-1},c_i} the counts of class c_{i-1} preceding class c_i and $F_{f,c}$ the counts of emission f with class c .

If we consider multiple features, then we still assume that the features are independent with each other (it's the main assumption in the Naive Bayes approach also). Only the emission distribution is changed and we can combine the probability together:

$$p(x_i = (\delta(f_1), \delta(f_2)) | y_i = \delta(c)) = p(x_i = \delta(f_1) | y_i = \delta(c)) p(x_i = \delta(f_2) | y_i = \delta(c)) = \frac{F_{f_1,c}}{F_{.,c}} \frac{F_{f_2,c}}{F_{.,c}}$$

3.2 Maximum-Entropy Markov Model

Next, we implemented a Maximum-Entropy Markov Model. The objective of the MEMM is to evaluate at each time step a distribution over the possible tags using features of the current word, denoted as $feat(x_i)$ and the tag of the previous word, c_{i-1} , using multi-class logistic regression, i.e.

$$p(y_i | y_{i-1}, feat(x_i)) = \text{softmax}([feat(x_i), c_{i-1}] \mathbf{W} + \mathbf{b})$$

3.3 Viterbi algorithm

The search algorithm that we implemented is the dynamic programming algorithm named after Andrew Viterbi. Its main difference with a greedy approach is that it evaluates at every step and for every previous state, the best possible next step. This guarantees a solution closer to the true optimal solution. The pseudo-code of the algorithm is given by:

```
procedure VITERBIWITHBP
   $\pi \in \mathbb{R}^{n+1 \times \mathcal{C}}$  initialized to  $-\infty$ 
   $bp \in \mathcal{C}^{n \times \mathcal{C}}$  initialized to  $\epsilon$ 
   $\pi[0, \langle s \rangle] = 0$ 
  for  $i = 1$  to  $n$  do
    for  $c_{i-1} \in \mathcal{C}$  do
      compute  $\hat{y}(c_{i-1})$ 
      for  $c_i \in \mathcal{C}$  do
         $score = \pi[i-1, c_{i-1}] + \log \hat{y}(c_{i-1})_{c_i}$ 
        if  $score > \pi[i, c_i]$  then
           $\pi[i, c_i] = score$ 
           $bp[i, c_i] = c_{i-1}$ 
  return sequence from  $bp$ 
```

3.4 Structured Perceptron

The final model, we implemented is the structure perceptron train algorithm. The way the model is trained uses the Viterbi search algorithm, presented above. At each epoch, we uses Viterbi to predict the highest scored sequence given the state of the model. We can then find the timesteps where the actual sequence for the given sentence and the predicted one differ and compute at each of these time steps, the gradient of a hinge type loss. These gradients have a -1 entry on the true class for this given word, and a 1 on the predicted class by the model. We can then propagate these gradients in the network, and update the weights with a learning rate that can be tuned.

The model itself is similar to the model of the MEMM without the final logsoftmax layer.

4 Experiments

4.1 Feature Engineering

The original paper suggests several features to use. We focus on the word counts and a capitalization feature. We defined our capitalization feature as follow:

1. 1 : word in low caps;
2. 2 : whole word in caps;
3. 3 : first letter in cap;
4. 4 : one cap in the word;

5. 5 : other

We then produced an embedding of the word counts using a pre-trained version.

We also used the Python "pattern.en" package to extract Part-of-Speech (PoS) features. The packages generates 41 features to which we added special feature for the opening and closing tabs `<s>` and `< \s>`.

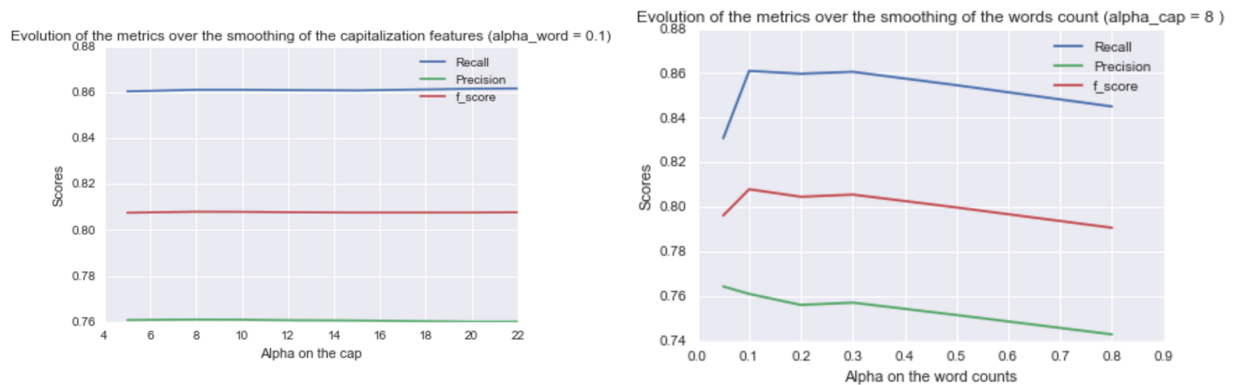
4.2 Model Evaluation

As used in the Kaggle competition, we used the f-score with the precision and recall measure to evaluate our model while tuning the hyperparameters. A positive prediction stands for a label (in the notation of the task, everything which is not the **O** tag):

1. recall: ratio of the true positive predictions among the positives tags in the correct sequence
2. precision: ratio of the true positive predictions among the positive predictions,
3. f-score (with $\beta = 1$): harmonic mean of the precision and the recall, i.e. $f_1 = \frac{2pr}{p+r}$

4.3 Hidden Markov Model

There is only the smoothing parameter α and eventually feature selection here to tune here. We evaluate the impact of adding more features and run experiments with different alpha values to tune them . One important details is to make sure to use a specific smoothing parameter for each distribution, i.e a smoothing parameter may be applied to the transition matrix but also to the emission matrix of each different feature. Each of this distribution has a different tail and need a different smoothing. For instance, the transition matrix need a very small α (around 0.1) because we are pretty confident in it but the capitalizations feature need one much bigger (around 20) because the counts are already high.



We notice that the model is less sensitive to the changes of the smoothing parameter on the capitalization feature as on the word counts. This is pretty reasonable as the feature counts are much higher in the capitalization feature than in the word counts. Tuning this parameter provides

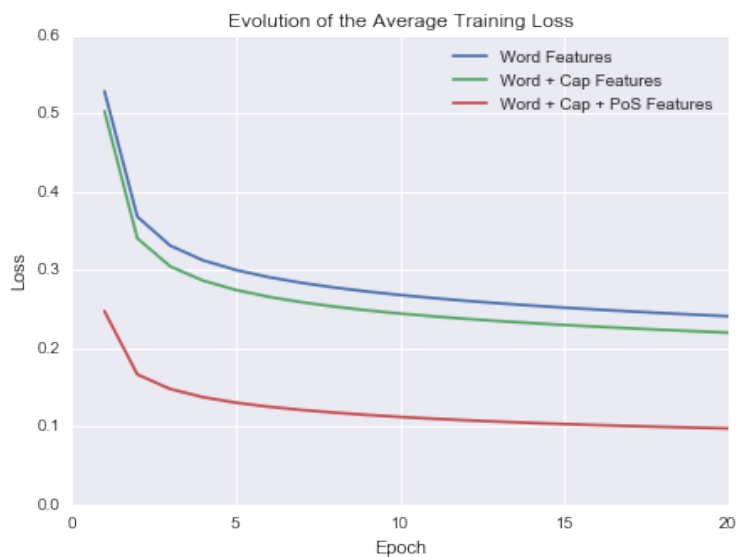
a model with a f-score of **0.808**. Using only the word counts features provide a best f-score of **0.764**.

We obtained a Kaggle score on the test set of :

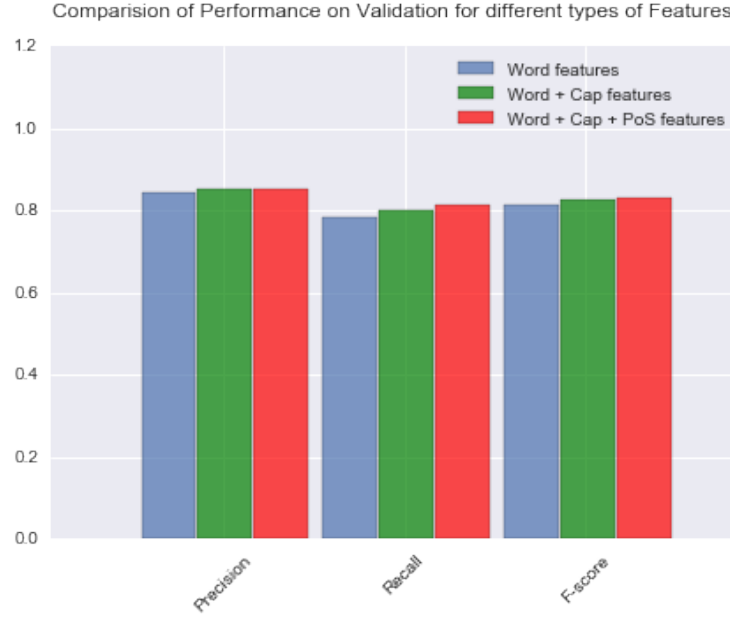
$$K_{HMM} = 0.48365$$

4.4 Maximum-Entropy Markov Model

We coded the MEMM using the nn module and trained using stochastic gradient descent. We also used the Glove embeddings using a lookup table. As for the HMM, we used two different sets of features, i.e. the words and the words and capitalisation of the words. We also added the Part of Speech features that were evaluated using the python package "pattern.en" in order to gain some time. We observed that the training algorithm converges quite rapidly, and that if adding caps to the features helped decrease the loss, the impact was not as strong as expected. On the other hand, adding PoS features impacted greatly the loss. Nevertheless, we trained the model on 20 epochs in order to learn the embeddings for the `<s>` and `<s>` "words" added during pre-processing.



We evaluated the performance of these two models using the f-score presented above:



Adding extra features yielded better results on both Precision and Recall and therefore on the f-score. But as we expected from the small differences in loss, we did not observe an important increase on the f-score using cap features. We were nevertheless surprised to see that the impact on loss using PoS features did not translate on the f-score. These results were later confirmed on the test set, as the kaggle score obtained for these two models were:

$$K_{nocaps} = 0.52057 \quad \text{and} \quad K_{caps} = 0.55482 \quad K_{PoS} = 0.57121$$

which are both slightly better than the results of the HMM.

4.5 Structured Perceptron

5 Conclusion

This segmentation task gave us the opportunity to implement different recurrent neural network architectures but also to compare them with more traditional methods. Whereas the count-based and even the simple neural network models are pretty fast to train, they still provide interesting results. The results provided by the three variants of RNN were interesting to illustrate the influence of gates and memory in such networks. The gated recurrent network ended as the best model on this task. One future work could be to stack more layers to our recurrent architecture or to implement a network with a dynamic memory part to give more flexibility in how the model uses the information it already processed.

Appendices

Preprocessing:

```
1  #!/usr/bin/env python
2
3  """NER Preprocessing
4  """
5
6  import numpy as np
7  import h5py
8  import argparse
9  import sys
10 import re
11 import codecs
12
13 # Your preprocessing, features construction, and word2vec code.
14
15
16 FILE_PATHS = {"CONLL": ("data/train.num.txt",
17                          "data/dev.num.txt",
18                          "data/test.num.txt",
19                          "data/tags.txt")}
20 args = {}
21
22
23 def main(arguments):
24     global args
25     parser = argparse.ArgumentParser(
26         description=__doc__,
27         formatter_class=argparse.RawDescriptionHelpFormatter)
28     parser.add_argument('dataset', help="Data set",
29                         type=str)
30     args = parser.parse_args(arguments)
31     dataset = args.dataset
32     train, valid, test, tag_dict = FILE_PATHS[dataset]
33
34     filename = args.dataset + '.hdf5'
35     with h5py.File(filename, "w") as f:
36         f['train_input'] = train_input
37         f['train_output'] = train_output
38         if valid:
39             f['valid_input'] = valid_input
40             f['valid_output'] = valid_output
41         if test:
```

```

42         f['test_input'] = test_input
43         f['nfeatures'] = np.array([V], dtype=np.int32)
44         f['nclasses'] = np.array([C], dtype=np.int32)
45
46
47 if __name__ == '__main__':
48     sys.exit(main(sys.argv[1:]))

```

Hidden Markov Model:

```

1  — Documentation:
2  — ——— How to call it from the command line?
3  — For example:
4  — $ th count_based.lua -N 5
5  — Other argument possible (see below)
6  —
7  — ——— Is there an Output?
8  — By default, the predictions on the test set are saved in hdf5 format
   as classifier .. opt.f
9
10 — Only requirements allowed
11 require("hdf5")
12 require 'helper.lua';
13
14 cmd = torch.CmdLine()
15
16 — Cmd Args
17 cmd:option('-datafile', 'data/words_feature.hdf5',
18           'Datafile with features in hdf5 format')
19 cmd:option('-alpha_t', 0.1, 'Smoothing parameter alpha in the
   transition counts')
20 cmd:option('-alpha-w', 2, 'Smoothing parameter alpha in the word counts
   ')
21 cmd:option('-alpha_c', 20, 'Smoothing parameter alpha in the caps
   counts')
22 cmd:option('-test', 0, 'Boolean (as int) to ask for a prediction on
   test, will be saved in submission in hdf5 format')
23 cmd:option('-datafile_test', 'submission/v_seq_hmm', 'Smoothing
   parameter alpha in the word counts')
24 cmd:option('-nfeatures', 2, 'Number of type of features to use')
25 cmd:option('-cv', 0, 'Boolean (as int) to run a cross validation
   pipeline')
26
27
28
29 — Formating as log-probability and smoothing the input

```



```

30 function format_matrix(matrix, alpha)
31     local formatted_matrix = matrix:clone():type('torch.DoubleTensor')
32     formatted_matrix:add(alpha)
33     — Normalize
34     local norm_mat = torch.expandAs(formatted_matrix:sum(1),
        formatted_matrix)
35     formatted_matrix:cdiv(norm_mat)
36     return formatted_matrix:log()
37 end
38
39 — log-scores of transition and emission
40 — corresponds to the vector y in the lecture notes
41 — i: timestep for the computed score
42 function score_hmm(observations, i, emissions, transition, C, nfeatures
    )
43     local observation_emission = torch.zeros(C)
44     for k=1,nfeatures do
45         observation_emission:add(emissions[k][observations[{i,k}]]))
46     end
47     observation_emission = observation_emission:view(C, 1):expand(C, C)
48     — NOTE: allocates a new Tensor
49     return observation_emission + transition
50 end
51
52 — Viterbi algorithm.
53 — observations: a sequence of observations, represented as integers
54 — logscore: the edge scoring function over classes and observations in
    a history-based model
55 function viterbi(observations, logscore, emissions, transition,
    nfeatures)
56     local y
57     — Formating tensors
58     local initial = torch.zeros(transition:size(2), 1)
59     — initial started with a start of sentence: <t>
60     initial[{8,1}] = 1
61     initial:log()
62
63     — number of classes
64     C = initial:size(1)
65     local n = observations:size(1)
66     local max_table = torch.Tensor(n, C)
67     local backpointer_table = torch.Tensor(n, C)
68
69     — first timestep
70     — the initial most likely paths are the initial state distribution

```

```

71  — NOTE: another unnecessary Tensor allocation here
72  local init_pred = initial:clone()
73  for i=1,nfeatures do
74      init_pred:add(emissions[i][observations[{1,i}]]))
75  end
76  local maxes, backpointers = init_pred:max(2)
77  max_table[1] = maxes
78
79  — remaining timesteps ("forwarding" the maxes)
80  for i=2,n do
81      — precompute edge scores
82      y = logscore(observations, i, emissions, transition, C,
83                  nfeatures)
84      scores = y + maxes:view(1, C):expand(C, C)
85
86      — compute new maxes (NOTE: another unnecessary Tensor
87      allocation here)
88      maxes, backpointers = scores:max(2)
89
90      — record
91      max_table[i] = maxes
92      backpointer_table[i] = backpointers
93  end
94  — follow backpointers to recover max path
95  local classes = torch.Tensor(n)
96  maxes, classes[n] = maxes:max(1)
97  for i=n,2,-1 do
98      classes[i-1] = backpointer_table[{i, classes[i]}]
99  end
100  return classes
101 end
102 — Prediction pipeline
103 function predict(observations, emissions, transition, alphas, nfeatures
104 )
105     — Formating model parameters (log and alpha smoothing)
106     — Alphas is a tensor : {alpha_t, alpha_w, alpha_c}
107     emissions_cleaned = {}
108     for i=1,nfeatures do
109         emissions_cleaned[i] = format_matrix(emissions[i], alphas[i+1])
110     end
111     local transition_cleaned = format_matrix(transition, alphas[1])
112
113     return viterbi(observations, score_hmm, emissions_cleaned,

```

```

        transition_cleaned , nfeatures)
113 end
114
115 — Cross validation pipeline
116 function cross_validation(observations , emissions , transitions ,
        true_classes ,
117                           alphas_table , alpha_t)
118     — alphas_table is a table of tensor with the range of parameters
        to use
119     — Current implementation for 2 features only
120     — alphas_table = {alpha_w_tensor , alpha_c_tensor}
121     — Return a tensor with first columns the alpha value and last the
        score for each
122     local nfeatures = #alphas_table
123     local v_seq_dev , precision , recall , f
124     local alphas = torch.DoubleTensor(3)
125     local size1 = alphas_table[1]:size(1)
126     local size2 = alphas_table[2]:size(1)
127     local num_evaluations = size1*size2
128
129     — Columns for 2 features are (alphas_w_value , alphas_c_value ,
        f_score , precision , recall)
130     local scores = torch.DoubleTensor(num_evaluations , nfeatures+3)
131
132     for i=1,size1 do
133         alpha_w = alphas_table[1][i]
134         for k=1,size2 do
135             alpha_c = alphas_table[2][k]
136
137             alphas:copy(torch.Tensor({alpha_t , alpha_w , alpha_c}))
138             v_seq_dev = predict(observations , emissions , transition ,
                alphas , nfeatures)
139             precision , recall = compute_score(v_seq_dev , true_classes)
140             f = f_score(precision , recall)
141
142             — Filling the scores tensor
143             scores[{(i-1)*size2+k , 1}] = alpha_w
144             scores[{(i-1)*size2+k , 2}] = alpha_c
145             scores[{(i-1)*size2+k , 3}] = f
146             scores[{(i-1)*size2+k , 4}] = precision
147             scores[{(i-1)*size2+k , 5}] = recall
148         end
149     end
150
151     return scores

```

```

152 end
153
154
155 function main()
156     — Parse input params
157     opt = cmd:parse(arg)
158
159     — Reading file from pre-processing
160     myFile = hdf5.open(opt.datafile, 'r')
161     data = myFile:all()
162     emission_w = data['emission_w']
163     emission_c = data['emission_c']
164     — Table of emission tensor (one tensor per feature)
165     emissions = {emission_w, emission_c}
166     — Assertion on number of features
167     nfeatures = opt.nfeatures
168     if nfeatures > #emissions then
169         error('Too many features specified')
170     end
171     print('Number of features used: '..nfeatures)
172     transition = data['transition']
173     input_matrix_train = data['input_matrix_train']
174     input_matrix_dev = data['input_matrix_dev']
175     input_matrix_test = data['input_matrix_test']
176     myFile:close()
177
178     — Parameters:
179     true_classes = input_matrix_dev:narrow(2,5,1):clone():view(
        input_matrix_dev:size(1))
180     — contain in each column feature observation
181     — (same order as the feature emission tensor in the emissoins
        table)
182     observations = input_matrix_dev:narrow(2,3,nfeatures):clone()
183     — Alpha parameter
184     alphas = torch.Tensor({opt.alpha_t, opt.alpha_w, opt.alpha_c})
185
186     — Prediction on dev
187     v_seq_dev = predict(observations, emissions, transition, alphas,
        nfeatures)
188     precision, recall = compute_score(v_seq_dev, true_classes)
189     f = f_score(precision, recall)
190
191     print('Prediction on dev')
192     print('Precision is : '..precision)
193     print('Recall is : '..recall)

```

```

194     print('F score (beta = 1) is : '..f)
195
196     — Cross validation
197     if (opt.cv == 1) then
198         alphas_table = {}
199         — alpha_w
200         alphas_table[1] = torch.Tensor({0.05, 0.1, 0.2, 0.3, 0.5, 0.8})
201         — alpha_c
202         alphas_table[2] = torch.Tensor({5, 8, 10, 12, 15, 20, 22})
203
204         scores = cross_validation(observations, emissions, transitions,
                                   true_classes,
205                                   alphas_table, opt.alpha_t)
206         print(scores)
207
208         — Saving the score
209         myFile = hdf5.open('plot_scores.hdf5', 'w')
210         myFile:write('scores', scores)
211         myFile:close()
212         print('CV on dev saved in '..plot_scores.hdf5')
213     end
214
215     — Prediction on test
216     if (opt.test == 1) then
217         print('Prediction on test')
218         observations_test = input_matrix_test:narrow(2,3,nfeatures):
219                               clone()
220         v_seq_test = predict(observations_test, emissions, transition,
221                               alphas, nfeatures)
222         — Saving predicted sequence on test
223         myFile = hdf5.open(opt.datafile_test, 'w')
224         myFile:write('v_seq_test', v_seq_test)
225         myFile:write('v_seq_dev', v_seq_dev)
226         myFile:close()
227         print('Sequence predicted on test saved in '..opt.datafile_test
228               )
229     end
230 main()

```

Max-Entropy Markov Model:

```

1 require 'hdf5';
2 require 'nn';

```

```

3  require 'helper.lua';
4
5  — Loading data
6  myFile = hdf5.open('../data/MM_data_pos.hdf5', 'r')
7  data = myFile:all()
8  input_matrix_train_pos = data['input_matrix_train_pos']
9  input_matrix_dev_pos = data['input_matrix_dev_pos']
10 input_matrix_test_pos = data['input_matrix_test_pos']
11 myFile:close()
12
13 nwords = input_matrix_train_pos:size(1)
14 train_output = input_matrix_train_pos:narrow(2,59)
15 train_input_pos = torch.Tensor(nwords-1,1+9+5+43)
16 train_input_pos:narrow(2,1,1):copy(input_matrix_train_pos:narrow(2,1,1)
   :narrow(1,2,nwords-1))
17 train_input_pos:narrow(2,2,9):copy(input_matrix_train_pos:narrow(2,2,9)
   :narrow(1,1,nwords-1))
18 train_input_pos:narrow(2,11,5):copy(input_matrix_train_pos:narrow
   (2,11,5):narrow(1,1,nwords-1))
19 train_input_pos:narrow(2,16,43):copy(input_matrix_train_pos:narrow
   (2,16,43):narrow(1,1,nwords-1))
20
21 observations_dev = input_matrix_dev_pos:narrow(2,1,1):clone()
22 dev_feat = input_matrix_dev_pos:narrow(2,11, 5 + 43)
23 dev_true_classes = input_matrix_dev_pos:narrow(2, 59,1):squeeze()
24
25 observations_test_pos = input_matrix_test_pos:narrow(2,1,1)
26 observations_test_feat = input_matrix_test_pos:narrow(2,2,5+43)
27
28 — Defining the model
29
30 model = nn.Sequential()
31 t1_pos = nn.ParallelTable()
32
33 t1_pos_1 = nn.Sequential()
34 t1_pos_1:add(LT)
35 t1_pos_1:add(nn.View(-1,50))
36
37 t1_pos_2 = nn.Identity()
38
39 t1_pos:add(t1_pos_1)
40 t1_pos:add(t1_pos_2)
41
42 model:add(t1_pos)
43 model:add(nn.JoinTable(2))

```

```

44
45 model:add(nn.Linear(50 + 9 + 5 + 43,9))
46 model:add(nn.LogSoftMax())
47
48 — Training function:
49
50
51 function train_model_cap(train_input, train_output, model, criterion,
    din, nclass, eta, nEpochs, batchSize)
52     — Train the model with a mini batch SGD
53     — standard parameters are
54     — nEpochs = 1
55     — batchSize = 32
56     — eta = 0.01
57     local loss = torch.Tensor(nEpochs)
58
59     — To store the loss
60     local av_L = 0
61
62     — Memory allocation
63     local inputs_batch = torch.DoubleTensor(batchSize, din)
64     local targets_batch = torch.DoubleTensor(batchSize)
65     local outputs = torch.DoubleTensor(batchSize, nclass)
66     local df_do = torch.DoubleTensor(batchSize, nclass)
67
68     for i = 1, nEpochs do
69         — timing the epoch
70         timer = torch.Timer()
71         av_L = 0
72
73         — mini batch loop
74         for t = 1, train_input:size(1), batchSize do
75             — Mini batch data
76             current_batch_size = math.min(batchSize, train_input:size(1)
                -t)
77
78             inputs_batch:narrow(1,1,current_batch_size):copy(
                train_input:narrow(1,t,current_batch_size))
79
80             targets_batch:narrow(1,1,current_batch_size):copy(
                train_output:narrow(1,t,current_batch_size))
81
82             — reset gradients
83             model:zeroGradParameters()
84

```

```

85      — Forward pass (selection of inputs_batch in case the
      batch is not full, ie last batch)
86      outputs:narrow(1,1,current_batch_size):copy(model:forward({
      inputs_batch:narrow(1,1,current_batch_size):narrow
      (2,1,1),
87      inputs_batch:narrow(1,1,current_batch_size):narrow(2,2,din
      -1})))
88      — Average loss computation
89      f = criterion:forward(outputs:narrow(1,1,current_batch_size
      ), targets_batch:narrow(1,1,current_batch_size))
90
91      av_L = av_L +f
92
93      — Backward pass
94      df_do:narrow(1,1,current_batch_size):copy(criterion:
      backward(outputs:narrow(1,1,current_batch_size),
      targets_batch:narrow(1,1,current_batch_size)))
95      model:backward({inputs_batch:narrow(1,1,current_batch_size)
      :narrow(2,1,1), inputs_batch:narrow(1,1,
      current_batch_size):narrow(2,2,din-1)},
96      df_do:narrow(1,1,current_batch_size))
97
98      model:updateParameters(eta)
99
100     end
101
102     print('Epoch '..i..'': '..timer:time().real)
103     loss[i] = av_L/math.floor(train_input:size(1)/batchSize)
104     print('Average Loss: '.. loss[i])
105
106     end
107
108     return loss
109 end
110
111 — Viterbi for MEMM:
112
113 — Evaluates the matrix of scores for all possible tags for the
    previous word, using the word features at timestep i
114
115 function compute_logscore_extrafeat(observations, feat, i, model, C)
116     local y = torch.zeros(C,C)
117     local hot_1 = torch.zeros(C+feat:size(2))
118     for j = 1, C do
119         hot_1:zero()

```



```

120         hot_1[j] = 1
121         hot_1:narrow(1,10,feat:size(2)):copy(feat:narrow(1,i,1))
122         y:narrow(1,j,1):copy(model:forward({ observations[i]:view(1,1),
            hot_1:view(1,C+feat:size(2))}))
123     end
124     return y
125 end
126
127 — Evaluates the highest scoring sequence:
128 function viterbi_extrafeat(observations, feat, compute_logscore, model,
    C)
129
130     local y = torch.zeros(C,C)
131     — Formating tensors
132     local initial = torch.zeros(C, 1)
133     — initial started with a start of sentence: <t>
134
135     initial[{8,1}] = 1
136     initial:log()
137
138     — number of classes
139     local n = observations:size(1)
140     local max_table = torch.Tensor(n, C)
141     local backpointer_table = torch.Tensor(n, C)
142     — first timestep
143     — the initial most likely paths are the initial state distribution
144     local maxes, backpointers = (initial + compute_logscore_extrafeat(
        observations, feat, 1, model, C)[8]):max(2)
145     max_table[1] = maxes
146     — remaining timesteps ("forwarding" the maxes)
147     for i=2,n do
148         — precompute edge scores
149
150         y:copy(compute_logscore_extrafeat(observations, feat, i, model,
            C))
151         scores = y:transpose(1,2) + maxes:view(1, C):expand(C, C)
152
153         — compute new maxes
154         maxes, backpointers = scores:max(2)
155
156         — record
157         max_table[i] = maxes
158         backpointer_table[i] = backpointers
159     end
160     — follow backpointers to recover max path

```

```

161     local classes = torch.Tensor(n)
162     maxes, classes[n] = maxes:max(1)
163     for i=n,2,-1 do
164         classes[i-1] = backpointer_table[{i, classes[i]}]
165     end
166
167     return classes
168 end
169
170 — Train Model
171
172 loss_pos = train_model_cap(train_input_pos, train_output,
    ultimate_t_pos, criterion, 1 + 9 + 5 + 43, 9, 0.1, 20, 32)
173
174 — Evaluate performance on dev set:
175
176 cl_pos_dev = viterbi_extrafeat(observations_dev, dev_feat,
    compute_logscore_extrafeat, ultimate_t_pos, 9)
177 f = f_score(cl_pos_dev, dev_true_classes)
178
179 — Predict on test:
180
181 v_seq_test_pos = viterbi_extrafeat(observations_test_pos,
    observations_test_feat, compute_logscore_extrafeat, ultimate_t_pos,
    9)
182
183 — Saving predicted sequence on test
184 myFile = hdf5.open('../submission/v_seq_test_mem_pos', 'w')
185 myFile:write('v_seq_test', v_seq_test_pos)
186 myFile:close()

```

Helper:

```

1 — function to evaluate the predicted sequence
2 — need to compute precision and recall (class 1 stands for negative
    class)
3 function compute_score(predicted_classes, true_classes)
4     local n = predicted_classes:size(1)
5     local right_pred = 0
6     local positive_true = 0
7     local positive_pred = 0
8     for i=1,n do
9         if predicted_classes[i] > 1 then
10             positive_pred = positive_pred + 1
11         end
12         if true_classes[i] > 1 then

```

```
13         positive_true = positive_true + 1
14     end
15     if (true_classes[i] == predicted_classes[i]) and true_classes[i
16         ] > 1 then
17         right_pred = right_pred + 1
18     end
19     — Verbose
20     — print('positive_true: '..positive_true)
21     — print('positive_pred: '..positive_pred)
22     — print('right_pred: '..right_pred)
23     local precision = right_pred/positive_pred
24     local recall = right_pred/positive_true
25     return precision , recall
26 end
27
28 function f_score(precision , recall)
29     return 2*precision*recall/(precision+recall)
30 end
```