50.007 Machine Learning, 1D

Introduction

Name	Student ID
Keith Low Wei Kang	1005866
Lee Le Xuan	1006029
Luv Singhal	1006250

Table 1: Team members

Given in table 1 are the members of our group. The code can be found on github at https://github.com/Lutetium-Vanadium/ml-1d.

Part 1

Given in table 2 are the scores of both the datasets. Note that in the code, the tokens are not actually replaced by the special token. Instead, for all tokens x in the training data, there exists a count for all times x occurs given label y, even if this count is 0. For tokens that do not occur in the training data, this count doesn't exist, and is replaced with the default value of k using Python's `dict.get` method.

Dataset	Precision	Recall	F
`RU` (entity)	0.1465	0.6838	0.2413
`RU` (sentiment)	0.0710	0.3316	0.1170
`ES` (entity)	0.1214	0.7773	0.2100
`ES` (sentiment)	0.0662	0.4236	0.1145

Table 2: Reported scores.

Part 2

Given in table 3 are the scores of both the datasets. To deal with the underflow issue, we used log probabilities[1].

[1]log probabilities were used in part 1 also for code consistency even though there were no underflow issues encountered then

Dataset	Precision	Recall	F
`RU` (entity)	0.3873	0.4859	0.4310
`RU` (sentiment)	0.2643	0.3316	0.2942
`ES` (entity)	0.2472	0.5852	0.3476
`ES` (sentiment)	0.1790	0.4236	0.2516

Table 3: Reported scores.

Part 3

In the following explanation, to distinguish between the variable k in Viterbi's algorithm and the k-th best sequence, we instead use the term t-th best sequence.

Viterbi's algorithm works by finding the highest probability sequence for every label v at every token k using dynamic programming. To find the t-th best sequence, we modify Viterbi's algorithm to store the t highest probabilities for every given label v and every token k. This is done by maintaining a sorted list (a min-heap priority queue in the code) considering the previous t highest probability sequences for token $k-1$ and then keeping just the top t of them for each of the tokens v. So now, the modified $\pi(k, v, i)$ stores the i-th best $r(y_1, y_2, \ldots, y_{k-1}, v)$.

When getting the t-th best labelled sequence at the end, we cannot use the same process as Viterbi's algorithm, as t-th best sequence may contain the best sequence until some token k and then a worse choice later. So, for every (k, v, i), we also store the previous (u, i') in the i-th best sequence. Then at the end to retrieve all the labels, we just follow this chain until we reach $k = 1$.

Since we now need to iterate over the top t probabilities for every k, v, and for each of those iterations we also need to insert into the priority queue, the new time complexity of the algorithm will be $\mathcal{O}\left(n|\mathcal{T}|^2 t \log t\right)$.[2]

Given in tables 4 and 5 are the scores of both the datasets.

Dataset	Precision	Recall	F
`RU` (entity)	0.2697	0.5193	0.3550
`RU` (sentiment)	0.1682	0.3239	0.2214
`ES` (entity)	0.2522	0.5109	0.3377
`ES` (sentiment)	0.1422	0.2882	0.1905

Table 4: Reported scores for $k = 2$.

[2]In the implementation, to check for uniqueness, every iteration also goes through the entire heap. This can be implemented within $\log t$ time, but for simplicity was left as a naive search. Thus our implementation runs in $\mathcal{O}\left(n|\mathcal{T}|^2 t^2\right)$

Dataset	Precision	Recall	F
RU (entity)	0.2374	0.4370	0.3077
RU (sentiment)	0.1313	0.2416	0.1701
ES (entity)	0.2284	0.4629	0.3059
ES (sentiment)	0.1272	0.2576	0.1703

Table 5: Reported scores for $k = 8$.

Part 4

Approaches

2nd Order HMM

We tried an extension of the HMM we learnt in class, whereby we introduced the assumption that the state at time k depends only on the states at time $k-2, k-1$. The equations were adapted from [3].

Suppose that we have the model parameters $P(k|i,j) = a_{i,j,k}$ for each $\{i,j,k\} \in \mathcal{T}$ and $P(o_1, o_2|u)$ for each $o_1, o_2 \in \mathcal{O}$ and $u \in \mathcal{T}$.

The new transition probabilities $P(k|i,j) = a_{i,j,k}$ were obtained

$$\frac{\texttt{Count}(u; v; w)}{\texttt{Count}(u; v)}$$

Let $S(k,u)$ be the set of all state sequences $y_1, \dots y_n$ such that $y_n = u$.
Let

$$r(z_1, \dots, z_n) = \prod_{k=1}^{n} a_{z_{k-2}, z_{k-1}, z_k} \prod_{k-1}^{n} b_{z_k}(x_k)$$

For the base case at time steps k=0

$$\pi_0(s) = 1 \begin{cases} 1 & \text{if s = START} \\ 0 & \text{otherwise.} \end{cases}$$

For k=1 we have

$$\pi_1(START, u) = P(u) \cdot a_{START,u} \cdot b_u(x_0)$$

In the forward recursion we have

$$\pi_k(v,w) = \max_{(u,v)\in\mathcal{T}} \{\pi_{k-1}(u,v) \cdot a_{u,v,w} \cdot b_k(x_k)\}, 2 \le k \le n$$

where $(u,v), (v,w)$ are two transitions between the states u to w.
For the final predicted state we have

$$y^*_{n-1}, y^*_n = \arg\max_{(v,w\in\mathcal{T})} \{\pi_n(v,w)\}$$

Now we fix $y^*_{n-1} = w$ and work backwards. For instance, the best value of y^*_{n-2} if we fix tag y^*_{n-1} in position $n-1$ is

$$y^*_{n-2}, w = \arg\max_{(v\in\mathcal{T})} \{\pi_{n-2}(v,w)\}$$

Based on the literature we used, the 2nd order HMM is good for capturing higher-order dependencies and reduce biases that may be present in only 1st order transition probabilities. However, we were unable to obtain better metrics in implementation, with the 2nd order HMM receiving lower precision, recall and F-scores than the 1st order HMM.

We believe that this is likely because of the maximum likelihood estimations not being fully representative of the true 2nd order transition probabilities possibly due to lack of data as there are significantly fewer counts of transitions from state 1 to state 3 as compared to when we were looking at 1st order transition probabilities.

The code can be found in the `2nd-order-hmm` branch of the git repository.

Recurrant Neural Network

To tackle the issue of having variable sequence lengths in neural networks we learnt in class, we also tried to adapt and implement a single layer RNN with reference to literature [2].

For this RNN, we have $x_1, \ldots, x_T$, each a word embedding vector corresponding to a corpus of T words. At this single hidden layer of the RNN, we have the relationship

$$h_t = \sigma(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

with the items:

- $x_t \in \mathbb{R}^d$: input word embedding vector at time t with an embedding dimension of d
- $W^{hx} \in \mathbb{R}^{D_h \times d}$: weights matrix
- $W^{hh} \in \mathbb{R}^{D_h \times D_h}$: weights matrix for the previous h_{t-1}
- $h_{t-1} \in \mathbb{R}^{D_h}$: output of the non-linear function at the previous time-step $t-1$, with h_0 being the initial vector at $t = 0$.
- σ: the non-linearity function sigmoid.

For the output vector $\hat{y}_t$, we used the softmax function to get the output probability distribution of the possible states at each time-step. For the loss function, we used categorical cross-entropy error

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \times \log \hat{y}_{t,j}$$

at each time-step t.

We have initialized our word embedding vectors, each word a vector of 7 dimensions corresponding to the maximum likelihood estimates of $P(\text{state}|\text{observation})$.

We tried several combinations of hidden dimensions, word embedding dimensions, learning rates and epochs. While we did see an improvement of F-scores, precision and recall in the metrics, the predictions were unable to predict the minority classes such as B-negative, I-negative possibly due to class imbalance. So, we opted to go for other approaches that yielded better evaluation results.

The code can be found in the `rnn` branch of the git repository.

Smoothing with Absolute Discount

We tried using Absolute Discounting [1] for smoothing in place of Laplace smoothing. Absolute Discounting works by subtracting a fixed discount, d, from each non-zero probability, and then redistributing it equally over the new words. Given a token x that occurs in the training data under y:

$$e(x|y) = \begin{cases} \frac{\texttt{Count}(y \to x) - d}{\texttt{Count}(y)} & \text{If the word token } x \text{ appears in the training set} \\ \frac{d}{\texttt{Count}(y)} & \text{If } x \text{ is the special token } \texttt{\#UNK\#} \end{cases}$$

The same equation was applied to the transition probabilities to account for possible transitions that did not occur in the training set.

However, there wasn't much improvement from the Laplace smoothing, and d had to be a very small value for an improvement.

This could be because absolute discounting reduces the probabilities with the same d value for all labels. Thus, it may result in bias probabilities for labels that occur less, as the d value would be spread out over smaller values, resulting in larger probabilities for tokens that did not occur in the training data set.

The code can be found in the `absolute-discounting` branch of the git repository.

Changing Emission Probabilities for Unknown Tokens

When we were looking through different methods of calculating the emission probabilities, we realised that there was a bias towards tags that did not appear often when calculating the probabilities of new words. Given a token x which doesn't occur in the training data, it is clear from the below equation that labels which have a lower count will have a higher e.

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

This is opposite of what we expect intuitively as labels which occur more often should be more likely to be emitted. To fix this, we changed the emission probability given the special token $x = \texttt{\#UNK\#}$ to use the total count minus the count of the given label as shown in the equation below.

$$e(x|y) = \frac{k}{(\sum_{v \in \mathcal{T}} \texttt{Count}(v)) - \texttt{Count}(y) + k}$$

Thus, the new emission probability function is as follows:

$$e(x|y) = \begin{cases} \frac{\texttt{Count}(y \to x)}{\texttt{Count}(y) + k} & \text{If the word token } x \text{ appears in the training set} \\ \frac{k}{(\sum_{v \in \mathcal{T}} \texttt{Count}(v)) - \texttt{Count}(y) + k} & \text{If } x \text{ is the special token } \texttt{\#UNK\#} \end{cases}$$

This led to a major improvement in our F-scores. The change in F-score can be seen in table 6.

Ignoring token case

Since, both Russian and Spanish are languages whose words do not change meaning based on the capitalisation of letters, we decided to also try ignoring the case of tokens. This was

Dataset	Precision	Recall	F	HMModel2 F
RU (entity)	0.6549	0.5781	0.5527	0.4310
RU (sentiment)	0.4577	0.3342	0.3836	0.2942
ES (entity)	0.6509	0.6026	0.6259	0.3476
ES (sentiment)	0.5094	0.4716	0.4898	0.2516

Table 6: Changed scores when using new emission function

done by converting every token to lower case when learning and calculating the emission probabilities.

This led to a slight improvement in our F-scores. The change in F-score can be seen in table 7.

Dataset	Precision	Recall	F	HMModel2 F
RU (entity)	0.4340	0.4987	0.4641	0.4310
RU (sentiment)	0.2998	0.3445	0.3206	0.2942
ES (entity)	0.2786	0.6070	0.3819	0.3476
ES (sentiment)	0.2084	0.4541	0.2857	0.2516

Table 7: Changed scores when ignoring case of tokens

Results

In the end, since ignoring the token case and modifying the emission probability function had positive impacts, we choose to include those 2 approaches in our final model. The reported scores for the dev dataset can be seen in table 8

Dataset	Precision	Recall	F	HMModel2 F
RU (entity)	0.6492	0.5090	0.5706	0.4310
RU (sentiment)	0.4426	0.3470	0.3890	0.2942
ES (entity)	0.6575	0.6288	0.6429	0.3476
ES (sentiment)	0.5160	0.4934	0.5045	0.2516

Table 8: Scores with the final model

References

[1] Sweatha Boodidhi. "Using smoothing techniques to improve the performance of hidden Markov's Model". PhD thesis. University of Nevada, Las Vegas, 2011.

[2] Richard Socher Milad Mohammadi Rohit Mundra. *CS 224D: Deep Learning for NLP*. Spring 2015. URL: https://cs224d.stanford.edu/lecture_notes/notes4.pdf.

[3] Yang Sung-Hyun et al. “Log-Viterbi algorithm applied on second-order hidden Markov model for human activity recognition”. In: *International Journal of Distributed Sensor Networks* 14.4 (2018), p. 1550147718772541.