CSE842 HW2

Javen W. Zamojcin
zamojci1@msu.edu

2024, October 4

1 Problem 1: Hidden Markov Model

1.1 How To Run

The program for this part was implemented using `Python 3.12` in the included Jupyter notebook titled `zamojci1_hmm.ipynb`. All libraries used and their respective versions may be found in the included `requirements.txt` file.

1.2 Part A: Transition Probabilities

To calculate the MLE of the transition probabilities matrix, I first took advantage of the NLTK library's `bigrams()` function to generate bigrams of the word-tag pairs, where the first term in the bigrams represented tag t_{i-1} , and the second represented tag t_i. Then, I generated a NLTK conditional frequency distribution where t_{i-1} was the conditional parameter. Finally, I converted the count distributions to frequencies, and converted everything to a Pandas DataFrame. See Figure 1.

1.3 Part B: Emission Probabilities

Similarly for calculating the MLE of the emission probabilities matrix, I generated a conditional frequency distribution to calculate the counts of tags-per-word. However, I only used unigrams at this step and also performed basic preprocessing by converting every word to lower case. At the DataFrame initialization step is where I filtered for the target words, this way each tag-word cell captured the global corpus probabilities. See Figure 2.

2 Problem 2: Neural Network for Part-of-Speech Tagging

2.1 How To Run

The program for this part was implemented using `Python 3.12` in the included Jupyter notebook titled `zamojci1_nn.ipynb`. All libraries used and their re-

Transition Probabilities Matrix (A)

	NOUN	ADJ	VERB	NUM	ADV	.	ADP	\
DET	0.647379	0.233559	0.054263	0.018966	0.013171	0.010536	0.008078	
NOUN	0.259640	0.017029	0.136752	0.010537	0.020650	0.252235	0.212664	
ADJ	0.709961	0.060990	0.015956	0.018938	0.005517	0.065464	0.072920	
VERB	0.127926	0.051254	0.202445	0.017640	0.073269	0.064727	0.173970	
ADP	0.306030	0.077135	0.037798	0.058600	0.010765	0.008580	0.016916	
.	0.234426	0.043095	0.100444	0.025405	0.052486	0.110673	0.102540	
ADV	0.055539	0.120334	0.273514	0.024186	0.075545	0.132875	0.156166	
CONJ	0.345234	0.107840	0.176297	0.027604	0.058152	0.016930	0.057416	
PRT	0.041519	0.017668	0.651502	0.011926	0.033127	0.041519	0.102032	
PRON	0.007495	0.009073	0.761341	0.001183	0.058383	0.064300	0.045759	
NUM	0.412742	0.069714	0.046168	0.015235	0.036934	0.235919	0.132502	
X	0.119565	0.000000	0.021739	0.000000	0.010870	0.217391	0.054348	

	PRON	DET	PRT	X	CONJ
DET	0.005707	0.005268	0.001756	0.000966	0.000351
NOUN	0.012201	0.013603	0.016833	0.000326	0.047531
ADJ	0.002535	0.005368	0.016105	0.000298	0.025947
VERB	0.032016	0.179943	0.067088	0.000069	0.009653
ADP	0.033994	0.440227	0.008580	0.000324	0.001052
.	0.077974	0.163243	0.024650	0.001006	0.064056
ADV	0.036429	0.081218	0.028367	0.000000	0.015826
CONJ	0.042326	0.143541	0.024291	0.000368	0.000000
PRT	0.002208	0.080389	0.009717	0.000000	0.008392
PRON	0.006706	0.012229	0.021696	0.000000	0.011834
NUM	0.004155	0.010619	0.006925	0.000000	0.029086
X	0.000000	0.000000	0.010870	0.554348	0.010870

Figure 1: Q1-P1 Transition Probabilities Matrix

Emission Probabilities Matrix (B)

	science	dog	blue	well	like	but	all
DET	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
NOUN	0.000326	0.000228	0.000000	0.000000	0.000000	0.000000	0.000000
ADJ	0.000000	0.000000	0.002833	0.000298	0.000149	0.000000	0.000000
VERB	0.000000	0.000000	0.000000	0.000000	0.001250	0.000000	0.000000
ADP	0.000000	0.000000	0.000000	0.000000	0.002347	0.000567	0.000000
.	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ADV	0.000000	0.000000	0.000000	0.012242	0.000000	0.000000	0.003583
CONJ	0.000000	0.000000	0.000000	0.000000	0.000000	0.101583	0.000000
PRT	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.074647
PRON	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
NUM	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
X	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Figure 2: Q1-P2 Emission Probabilities Matrix

spective versions may be found in the included `requirements.txt` file.

2.2 Design Overview

The deep learning framework I implemented for this part was PyTorch. All optimizers, network modules, and criterion were provided through PyTorch. Data preprocessing was largely done manually which included vectorizing the text through word2vec, binary feature engineering, padding sentences for homogeneous size, and converting to tensors.

The two RNN models I implemented were extensions of PyTorch neural network modules. `RNN_2`, the second model and the one actually used for the experi-

ments, utilized `ReLU` non-linearity.

For handling the inhomogeneous sequences (i.e. sentences with varying lengths), I chose to search for the longest word count sentence and pad the rest of the sentences to match this length. It ended up being 271 words, so each sentence was pre-padded to this length using zero vectors for the filler words.

The word vectorization and embeddings were made using a combination of word2vec and binary features such as whether the word is entirely alphabetic, made of digits, is all lowercase, is titlecase, or is all uppercase. For word2vec parameters, I used a vector size of 100, a window size of 5, and a minimum count of 1. The total embedding size therefore was 106, and the input data tensor dimensions were $(3914, 271, 106)$. At this step, I also normalize the input data tensor.

For cross-validation hyper-parameters, I chose to experiment with the criterion, optimizer, number of hidden nodes, and learning rate. The criterion were between negative log likelihood loss and cross-entropy loss and the optimizers were between RMSprop and Adam. The number of hidden nodes were 64 and 128, and learning rates of 0.001 and 0.0001. A total of 16 model configurations were evaluated. The constant hyper-parameters used across all experiments were an epoch limit of 100, and 5-folds for cross-validation.

2.3 Results

Table 1 displays the final evaluation results across the 16 models trained and tested using 5-fold cross-validation. The Accuracy metric was calculated by first aggregating each model fold by the highest accuracy score across their 100 epochs, and then aggregating each model by the average accuracy score across their 5 folds.

The highest accuracy across all models was 0.913311. Interestingly, this score was achieved by four different model configurations. The common factor out of all four of these models was the `RMSprop` optimizer. Root Mean Squared Propagation (RMSProp) is a gradient-based technique that utilizes the moving average of square gradients to scale the learning rates for each parameter. This can help stabilize the learning process.

Another interesting observation is how much drastic variance there is in the performance across various model configurations. For example, model 12, one of the four highest accuracy models, only differed from model 14 in terms of having a larger learning rate hyper-parameter, but model 14 scored an abysmal average accuracy of 0.016216. Perhaps this is the result of the vanishing gradient problem common to RNN models.

Model ID	Criterion	Optimizer	Learning Rate	Hidden	Accuracy
0	NLLLoss	RMSprop	0.001	64	0.913311
1	NLLLoss	Adam	0.001	64	0.553371
2	NLLLoss	RMSprop	0.0001	64	0.733595
3	NLLLoss	Adam	0.0001	64	0.208962
4	NLLLoss	RMSprop	0.001	128	0.913311
5	NLLLoss	Adam	0.001	128	0.736923
6	NLLLoss	RMSprop	0.0001	128	0.913311
7	NLLLoss	Adam	0.0001	128	0.196463
8	CrossEntropyLoss	RMSprop	0.001	64	0.020281
9	CrossEntropyLoss	Adam	0.001	64	0.013999
10	CrossEntropyLoss	RMSprop	0.0001	64	0.012741
11	CrossEntropyLoss	Adam	0.0001	64	0.200772
12	CrossEntropyLoss	RMSprop	0.001	128	0.913311
13	CrossEntropyLoss	Adam	0.001	128	0.013761
14	CrossEntropyLoss	RMSprop	0.0001	128	0.016216
15	CrossEntropyLoss	Adam	0.0001	128	0.194214

Table 1: Q2 Model Results

3 Problem 3: Extra Work

3.1 How To Run

The program for this part was implemented using `Python 3.12` in the included Jupyter notebook titled `zamojci1_extra.ipynb`. All libraries used and their respective versions may be found in the included `requirements.txt` file.

3.2 Description

For the extra work problem, I wanted to re-implement the solution for part 2 but this time using and becoming familiar with a different deep learning framework, Keras, evaluate different hyper-parameters, and see if I could improve the performance starting over with an overall better understanding of the problem.

Starting with the pre-processing phase, rather than immediately trying to vectorize and create embeddings using word2vec, I vectorized by converting every word in the corpus to their respective index based on frequency. First, I standardized the corpus by converting everything to lowercase. Then I manually created the vocabulary with a frequency distribution, sorted by highest frequency, then created a dictionary containing each word's enumerated index. Each word in the tagged data was then mapped to their frequency index while still maintaining the sentence structure. Each tag for each word was also mapped to its respective categorical class index. For padding, I instead used a max sentence length of 200 words and used the Keras `pad_sequences()` routine.

For creating the models, I used the Keras framework to first create an embedding layer with an input dimension the size of the entire vocabulary ($27,000$), and an output dimension as a hyper-parameter. Then, rather than using RNN models, this time I used LSTM models with the number of hidden nodes also as a hyper-parameter. The key here to output the many-to-many predictions was to use the `return_sequences=True` LSTM argument. Each model used sparse categorical cross-entropy as the loss function, but the optimizer was a hyper-parameter. I also added a dense layer with the number of output nodes matching the number of possible tag classes (12). Each model was then trained and evaluated using 5-folds and 3 epochs, for a total of 18 different model configurations.

The cross-validation hyper-parameters for the optimizer were also between RMSprop and Adam. The embedding dimension parameters were 32, 64, and 128. And finally for the number of LSTM hidden nodes, 25, 50, and 100 were the options.

3.3 Results

Overall, the model performances with this framework implementation were far more consistent, scored higher with accuracy, and required an order-of-magnitude less training time.

As Table 2 shows, all models regardless of configuration achieved average accuracy scores of above 91%. This is a dramatic improvement from the other implementation, where many of the models were around 1% averaged accuracy. The best model, number 17, scored the highest averaged accuracy of 96.46%, another major improvement from the previous implementation. In general it seems that the Adam optimizer performed better than RMSprop for any given configuration. The highest scoring model also had the largest configuration options for both embedding dimension (128) and number of LSTM hidden nodes (100).

Model ID	Optimizer	Embedding Dimension	LSTM Hidden	Accuracy
0	RMSprop	32	25	0.911852
1	Adam	32	25	0.913142
2	RMSprop	32	50	0.913638
3	Adam	32	50	0.919325
4	RMSprop	32	100	0.918806
5	Adam	32	100	0.922802
6	RMSprop	64	25	0.913018
7	Adam	64	25	0.920977
8	RMSprop	64	50	0.914553
9	Adam	64	50	0.929244
10	RMSprop	64	100	0.920149
11	Adam	64	100	0.941680
12	RMSprop	128	25	0.914397
13	Adam	128	25	0.928185
14	RMSprop	128	50	0.919551
15	Adam	128	50	0.950103
16	RMSprop	128	100	0.920723
17	Adam	128	100	0.964639

Table 2: Q3 Model Results