Assignment One

Qintai Liu (QL819)

October 7, 2018

Abstract

This report is about how I implemented bag of word model, what kind of hyper-parameters I tried to tune, and analyzing the result.

1 Bag of Word Model

The bag of word model sums all of the words' embedding in an example and then gets the average which is the vector's representation of the example. Finally, the vector is converted to a scalar and scale it between 0 and 1 by sigmoid function representing the possibility that the example is positive.

2 Implementation

Since the task was to do hyper-parameter search, I built a class which integrated the construction of vocabulary, model training, model selection, and model evaluation. Basically, the class did the following 6 tasks.

- Use torchtext.Field to process text data, and build vocabulary and word look-up table.
- Use torchtext.data.TabularDataset to load data from csv file and split the data into train, validation and test set. The number of training examples is equal to 20000, validation examples is 5000, and the test examples is 25000. Since the split function use stratified sampling, each example set contains approximately the same percentage of samples of each target class as the complete set. In this task, the proportion of positive examples and negative examples is equal in train, validation, and test examples.
- Use torchtext.data.BucketIterator to define an iterator that loads batches of data from a dataset. The benefit of BucketIterator is that this method can batch examples of similar lengths together instead of setting a fix_length for all of the examples.
- Train the model on training dataset. For each epoch, the model will be evaluated on validation examples in term of accuracy rate and AUC. We will keep track of the model having the best accuracy rate on validation examples. And if the best accuracy rate doesn't improve in the last several epochs, the training process will stop. And draw the training curve showing how the accuracy rate is changed after each epoch for training examples and validation examples respectively.
- Apply the best model in terms of accuracy rate on validation examples to test examples, and calculate the accuracy rate and draw the confusion matrix.
- Show 3 correct and 3 incorrect predictions of the model on the validation set.

There are total 9 hyperparameters for the class:

- *n_gram*: an N-token sequence of words
- *min_freq*: the minimum frequency needed to include a token in the vocabulary. This parameter is used to control vocabulary size
- *embedding_dim*: the vector size for each word
- *optimizer*: optimization algorithms. Adam or Adagrad
- *learning_rate*: step size
- *weight_decay*: weight decay (L2 penalty)
- *decay_rate*: the rate of the learning_rate reducing (via torch.optim.lr_scheduler.LambdaLR)
- *batch_size*: batch size
- *early_stopping_rounds*: will stop training if one metric of one validation data doesn't improve in last early_stopping_round rounds

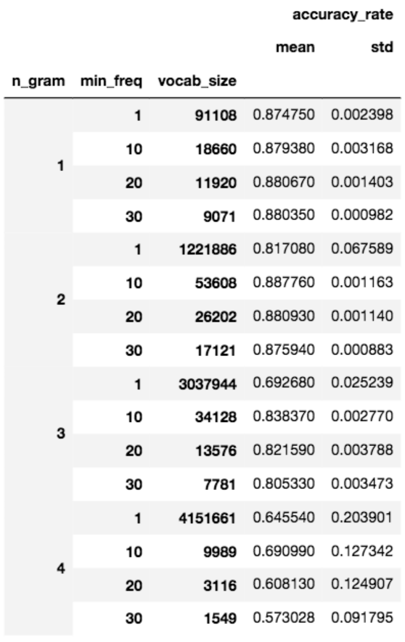
3 Analysis

3.1 n_gram, embedding_dim, min_freq

Since there are so many hyper-parameters could possibly affect the model performance, I first fixed *optimizer*=Adam, *learning_rate*=0.005, *weight_decay* = 0, *decay_rate* = 0.9, *batch_size* = 32, *early_stopping_rounds* =5, and see how n_gram, embedding_dim, min_freq would influence the final result. I did a gridsearch on *n_gram* = [1,2,3,4], *embedding_dim* = [30, 50, 100, 150], and *min_freq* = [1, 10, 20, 30]. Total 64 combinations.

For the result, I found parameter n_gram and min_freq play an important role to the model. And embedding_dim is important when the vocabulary size is extremely big since it can be used to control over-fitting. Now let analyze the result in detail.

From Figure1 which shows how the model performs in terms of accuracy rate on test examples with different value of n_gram, min_freq, and vocab_size. Since for specific combination of these there parameters, embedding_dim can be set to different 4 values (30, 50, 100, 150), the accuracy rate shown in the table is the average on these 4 different values and also the standard deviation is calculated.



			accuracy_rate	
			mean	std
n_gram	min_freq	vocab_size		
1	1	91108	0.874750	0.002398
	10	18660	0.879380	0.003168
	20	11920	0.880670	0.001403
	30	9071	0.880350	0.000982
2	1	1221886	0.817080	0.067589
	10	53608	0.887760	0.001163
	20	26202	0.880930	0.001140
	30	17121	0.875940	0.000883
3	1	3037944	0.692680	0.025239
	10	34128	0.838370	0.002770
	20	13576	0.821590	0.003788
	30	7781	0.805330	0.003473
4	1	4151661	0.645540	0.203901
	10	9989	0.690990	0.127342
	20	3116	0.608130	0.124907
	30	1549	0.573028	0.091795

Figure 1: Accuracy rate(on test examples) for different combination of n_gram, embedding_dim, min_freq

From Figure1, we can see get the following insights.

- When n_gram is equal to 1, the model has relatively good performance regardless of what value min_freq and embedding_dim are (in the reasonable range).
- When n_gram is equal to 2, as long as the vocabulary size is not extremely big, the model has pretty good performance regardless of what value embedding_dim is (in the reasonable range).
- When n_gram is equal to 3, the model is worse than models' whose n_gram is equal to 1 or 2. If we include all of the 3_gram appearing in the training examples, the vocabulary size is exploded to 3037944. Since such a huge vocabulary size can cause over-fitting, the model can not perform well.
- When n_gram is equal to 4, we can see that there are only 9989 tokens whose frequency is equal to or greater than 5, even if there are actually 4151661 tokens in the vocabulary. And the Figure1 shows that the model has a really bad performance when the vocabulary is less than 5000. I also have an interesting discovery. Then the vocabulary is too small, the model not only isn't able to capture any general pattern but also have a really bad performance on train examples. For example, when vocabulary size is equal to 3116 (min_freq= 20), the accuracy rate for both validation example and training example are the same and equal to 0.5. The result actually makes sense. If a sequence of 4 words can appear in training examples more than 20 times, it's highly likely that these sequences of 4 words have neutral meaning and are not used to express emotion.
- Since current min_freq setting is not well suitable when n_gram is equal to 4, I also try how the model performs when min_freq is equal to 3 and 4 respectively and fix the embedding size to 100. When min_freq=3, the vocabulary is equal to 87198 and accuracy rate is 0.80. When min_freq=3, the vocabulary is equal to 49534 and accuracy rate is 0.79.

In conclusion, for 1_gram and 2_gram, the model has promising performance even if the the vocabulary size is relative small. For 3_gram and 4_gram, in order to make the model perform well, careful setting for min_freq(or vocabulary size) need to be considered. Therefore the following analysis only consider 1_gram and 2_gram and ignore 3_gram and 4_gram.

3.2 Embedding_dim

Now let see how embedding_size can affect model performance for 1_gram and 2_gram. Figure2 shows how the model performs for different embedding_dim when n_gram is equal to 1 and 2 respectively. Please note, when calculating the average and standard deviation of the accuracy rate, only min_freq = [10, 20, 30] are considered and min_freq=1 is ignored. From Figure2, we can find embedding_dim doesn't have big impact on the model performance. And the performances for 1_gram and 2_gram are relatively same.

n_gram	embedding_dim	accuracy_rate mean	accuracy_rate std
1	30	0.878760	0.001768
	50	0.881653	0.000441
	100	0.880560	0.001606
	150	0.879560	0.002881

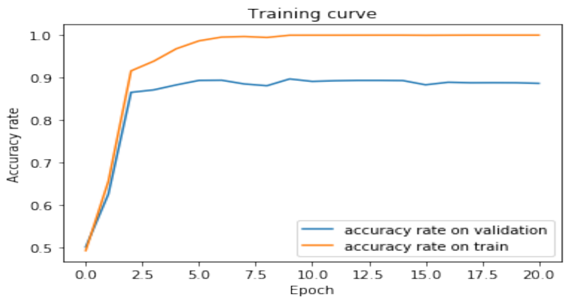
(a) Different Embed_dim on 1_gram

n_gram	embedding_dim	accuracy_rate mean	accuracy_rate std
2	30	0.880720	0.006311
	50	0.882160	0.005638
	100	0.882360	0.006742
	150	0.880933	0.005076

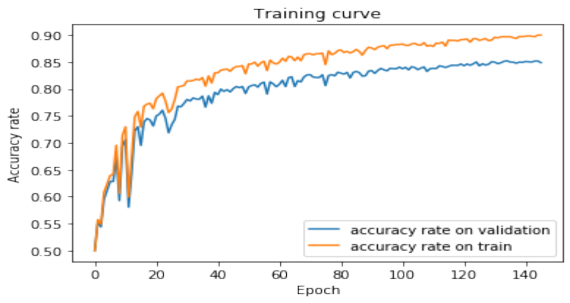
(b) Different Embed_dim on 2_gram

Figure 2: how embed_dim affects accuracy rate(on testing examples) for 1_gram and 2_gram

3.3 Optimizer



(a) Training curve for Adam



(b) Training curve for Adagrad

Figure 3: Training curve for Adam vs Adagrad

For simplicity, in the follow test, min_freq is fixed to 10, n_gram is fixed to 2, and embedding_dim is fixed to 50. The Figure3 show how the model performs for different optimizer(Adam,Adagrad) under learning_rate is 0.005 and early_stopping_rounds is 10. Adam converges very quickly compared with Adagrad. At 8th epoch, Adam reached its optimum while it takes Adagrad 133 epoch to reach the optimum. And the Adam surpasses Adagrad in terms of the accuracy rate on test examples which are 0.884 and 0.840 respectively.

3.4 Learning_rate

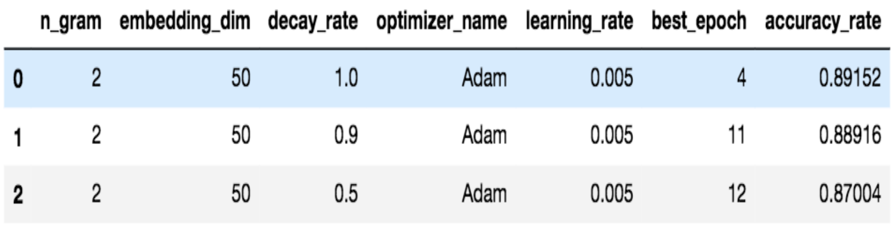
	n_gram	embedding_dim	min_freq	optimizer_name	learning_rate	best_epoch	accuracy_rate
0	2	50	10	Adam	0.100	8	0.86764
1	2	50	10	Adam	0.050	1	0.87532
2	2	50	10	Adam	0.010	4	0.88672
3	2	50	10	Adam	0.005	4	0.89124
4	2	50	10	Adam	0.001	23	0.88808

Figure 4: Accuracy on test examples for different learning_rate

Fix optimizer to be Adam, Figure4 shows how the model performs for different learning_rate. From the figure, we can see that learning_rate doesn't play an important role as long as it's relatively small. Under current setting, the best learning rate is 0.005

3.5 Decay_rate

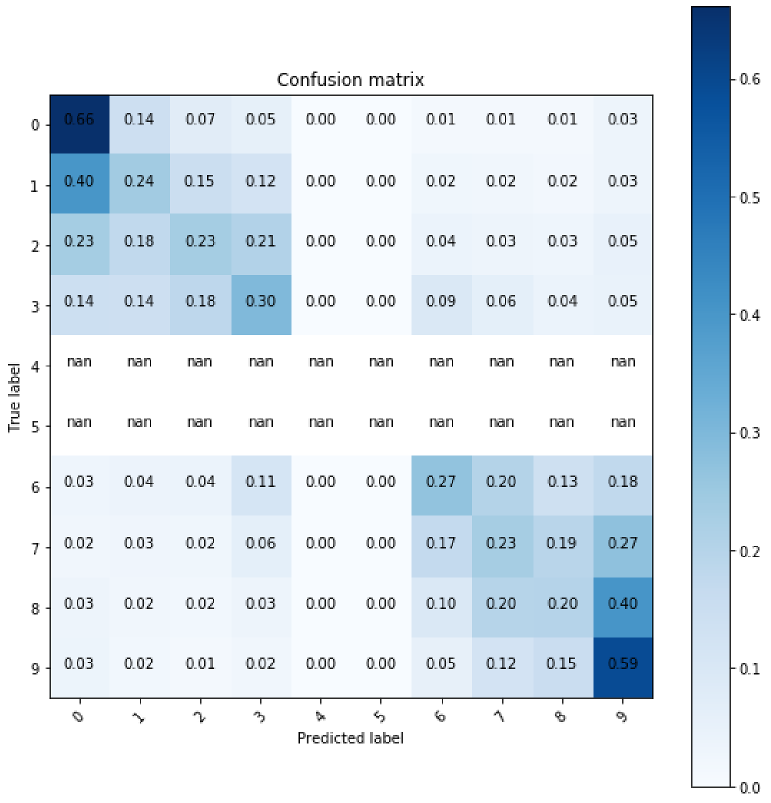
Figure5 shows how the decay_rate would affect model's performance. For each epoch, the learning rate would equal to $learning\_rate * (decay\_rate^{epoch})$. From the figure, we can see that since the initial learn_rate is set to 0.005 which is relative small, the effect of decay_rate is not very obvious. And when is decay_rate is smaller(learning_rate would decay more quickly), it takes more number of epoch to get the optimum.



	n_gram	embedding_dim	decay_rate	optimizer_name	learning_rate	best_epoch	accuracy_rate
0	2	50	1.0	Adam	0.005	4	0.89152
1	2	50	0.9	Adam	0.005	11	0.88916
2	2	50	0.5	Adam	0.005	12	0.87004

Figure 5: Accuracy on test examples for different decay_rate when learning_rate is set to 0.005

4 multiclass classification



	0	1	2	3	4	5	6	7	8	9
0	0.66	0.14	0.07	0.05	0.00	0.00	0.01	0.01	0.01	0.03
1	0.40	0.24	0.15	0.12	0.00	0.00	0.02	0.02	0.02	0.03
2	0.23	0.18	0.23	0.21	0.00	0.00	0.04	0.03	0.03	0.05
3	0.14	0.14	0.18	0.30	0.00	0.00	0.09	0.06	0.04	0.05
4	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
5	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
6	0.03	0.04	0.04	0.11	0.00	0.00	0.27	0.20	0.13	0.18
7	0.02	0.03	0.02	0.06	0.00	0.00	0.17	0.23	0.19	0.27
8	0.03	0.02	0.02	0.03	0.00	0.00	0.10	0.20	0.20	0.40
9	0.03	0.02	0.01	0.02	0.00	0.00	0.05	0.12	0.15	0.59

Figure 6: Confusion matrix for the prediction of review rate

The hyerparameters for this task are n_gram=2, embedding_dim=50, min_freq=10, optimizer=Adam, learning_rate=0.005, decay_rate=1, batch_size=32. Figure6 is the confusion matrix evaluated on the test examples and the accuracy rate is 0.397

5 Github link

`https://github.com/qltf8/ds_nlp/blob/master/ql819_nlp_hw_1.ipynb`