CS5740: Project X
https://github.com/cornell-cs5740-20sp/final-ks20

Kushal Singh
ks2377

20 April 2020

1 Introduction (6pt)

In this paper, we propose a model for name-entity recognition (NER) of Twitter tweets in order to identify sub-spans of words that represent named entities. The training and development data includes a token on every line, and is annotated with its respective label: B for beginning of named entity, I for continuation of a named entity, or O for a token that is not part of a named entity. Each named entity starts with B and all other tokens of the named entity, except the first one, are annotated with I. Sentences are separated by a new line.

For implementing aforementioned NER task on colloquial tweets, I began by perusing existing abstracts [1] in the field [2]. These papers demonstrated state-of-the-art performance in NER by virtue of bidirectional LSTM with a CRF model.

I used the training data to train the neural network model. Primary experiments included testing for optimal embedding dimensions, hidden layers, and number of training epochs. Cursory results of these experiments included an embedding dimension size of 100, number of hidden layers at 300, and 30 training epochs. The model was evaluated by testing development data prediction against the ground truth, with the F1 score generated from precision and recall. Final score achieved on leaderboard was **0.5321**.

2 Task (5pt)

As discussed above, we propose a model for name-entity recognition (NER) of Twitter tweets in order to identify sub-spans of words that represent named entities. To that extent, we define a Bi-LSTM CRF model, which takes as input a word-level and character-level embedding and yields as the output a matrix P of the emission probability of each word corresponding to each label, where $P_{a,b}$ is a projection probability of word w_a to tag_b. For CRF, we assume a transition matrix A of tag_a to tag_b with transition probability.

3 Data (4pt)

For data preprocessing, I generated sentences, labels (i.e. B, I, O), and tags (i.e. parts of speech, such as NN, VT) for the training, development, and test data. I also vectorized words and characters by creating a *word_to_id* dictionary that mapped words to integers and a *char_to_id* dictionary that mapped characters to integers, from the training set. Finally, I created a *label_to_id* dictionary that vectorized the tagged labels as follows: {START_TAG: 0, 'O': 1, 'B': 2, 'I': 3, STOP_TAG:4}.

Table 1: Data Statistics

Metric	Value
Word Vocab Size (Training)	9069
Character Vocab Size (Training)	67
Number of Sentences (Training)	2394
Word Vocab Size (Dev)	4572
Character Vocab Size (Dev)	261
Number of Sentences (Dev)	959
Word Vocab Size (Test)	9315
Character Vocab Size (Test)	378
Number of Sentences (Test)	2377
Total Parts of Speech	46

I created a list of dictionaries for each sentence. Each of these dictionaries contained the following keys: 'caps', 'chars', 'words', 'tags', 'str_words'. The 'caps' key mapped to a list of integers, where the ith value in the list indicated whether the ith word in the sentence was either: all lowercase (0), all uppercase (1), only first letter uppercase (2), only one letter except the first uppercase (3). The 'chars' key mapped to a list of lists, where the jth sublist contained a list of integers representing the vectorized characters of the jth word in a sentence based on the *char_to_id* dictionary discussed in the paragraph above. The 'words' key mapped to a list of integers, where the kth integer represented the vectorized version of the kth word in a sentence, based on the *word_to_id* dictionary discussed in the paragraph above. The 'tags' key mapped to a list of integers, where the lth integer represented the vectorized version of the tagged label for the lth word, as discussed in the *label_to_id* dictionary in the paragraph above. The 'str_words' key mapped to a list of strings, where the mth string represented the mth word in a sentence.

4 Model (25pt)

The LSTM model comprises of two components: word representation and bidirectional LSTM. The word embedding is represented as either raw text words or character-based encodings of the raw text words.

For generating the word-level embeddings from the training data, I built the *word_to_id* dictionary which mapped words to unique integers. This dictionary also contained a 'RARE' key to account for words that did not appear in the training set, but appeared in the dev set and/or test set (i.e. UNKNOWN words).

[1] https://arxiv.org/abs/1508.01991v1
[2] https://www.aclweb.org/anthology/Y18-1061.pdf

Each word is represented as a 100-dimensional vector which is subsequently trained in LSTM. Based on experimentation which will be discussed below, word embeddings in 100 dimensions yielded better results than word embeddings in 25, 50, 200, and 300 dimensions. To initialize the word embedding, we used the *glove.6B.100d.txt* pre-trained dataset [3].

For generating the character-level word embeddings from the training data, I built the *char_to_id* dictionary which mapped characters to unique integers. In the end, each word is represented as a list of z integers, where z is the length of the word. To create the input vector, we applied a small Bi-LSTM to handle the fact that words and, thus, the character-level embeddings have different lengths. Each character is represented as a 100-dimensional vector and is input into the Bi-LSTM to obtain the last step's output as input to the bidirectional LSTM.

The Bidirectional LSTM for Twitter NER is shown in Figure 1 below, and is motivated by the model presented in this abstract [4]. For each sentence, we generate word vector and character-based representations. Similar word representations are associated to the same words, are concatenated, and sequentially fed into bi-LSTM to learn word-based contexts within a sentence. At the output layer, I optimize the log-likelihood of labeling the whole sentence correctly by modeling interactions between two successive labels using Viterbi algorithm.

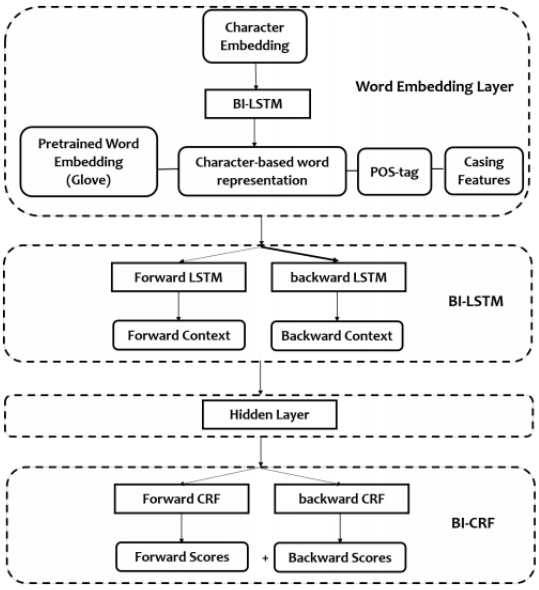


Figure 1: Architecture of the network

5 Learning (10pt)

In Bi-LSTM CRF, there are two kinds of potentials: emission and transition. The output of Bi-LSTM layer is a matrix P of the emission probability of each word corresponding to each label, where $P_{i,j}$ is a projection probability of word w_i to tag_j. For CRF, we assume a transition matrix A of tag_i to tag_j with transition probability. Let y be a tag sequence and X an input sequence of words.

$$S(X, y) = \sum_{i=0}^{n} A_{y_i, y_{i+1}} + \sum_{i=1}^{n} P_{i, y_i}$$

Given a word sequence X, the probability of tag sequence y can be written with Softmax:

$$\Pr(y|X) = \frac{e^{s(X,y)}}{\sum_{y' \in (Y_X)} e^{s(X,y)}}$$

The objective is to maximize $P(y|X)$, and the loss function is defined as $-log(P(y|X))$, with SGD as the optimizer.

The *word_to_id* dictionary discussed above contains a 'RARE' key to account for words that did not appear in the training set, but appeared in the dev set and/or test set (i.e. UNKNOWN words). For instance, there are a lot of emojis that appear in the test set and not the training set. These emojis were treated as unknowns and did not make a significant impact on the model, since they do not play a big role in determining named entities. The 'RARE' key is sufficient in this case—the need for chunking unknown words into different categories (e.g. EMOJI) is unnecessary because words also have their respective character-based embeddings.

6 Implementation Details (3pt)

The LSTM model uses stochastic gradient descent (SGD) as the optimization algorithm. It comprised of the following hyperparameters and their values:

```
word_embed_dim = 100
word_lstm_dim = 100
char_embed_dim = 100
char_lstm_dim = 100
hidden_dim = 300
pre_embed_file = 'glove.6B.100d.txt'
learning_rate = 0.05
momentum = 0.9
epochs = 30
```

During the results and experimentation phase, the following variables were fine-tuned to optimize performance: embedding size (for both character-level and word-level representations), hidden layer size, learning rate, number of epochs, and the GloVe pretrained embeddings file.

7 Experimental Setup (4pt)

The variables fine-tuned in the paragraph above were evaluated against the F1 score. While fine-tuning a particular variable, all other variables were held constant to prevent result obfuscation.

The evaluation metric is the F1 score which is calculated against the precision and recall. Specifically, the precision is calculated as $\frac{TP}{TP + FP}$, where TP is True Positives and FP is False Positives. Recall is calculated as $\frac{TP}{TP + FN}$, where FN is False Negatives. The F1 score is calculated as $\frac{2 \times Precision \times Recall}{Precision + Recall}$.

[3] https://nlp.stanford.edu/projects/glove/
[4] https://www.aclweb.org/anthology/Y18-1061.pdf

8 Results

Test Results (3pt) We generated this test score using our procedure for handling unknown words described above. The hyper-paramater values were $d = 100$ for character and word embeddings, $h = 300$ for hidden layers, and $epochs = 30$.

Table 2: Leaderboard Performance

F1	Precision	Recall
0.5321	0.6557	0.44771

Development Results (7pt) The F1 score generated in Tables 3 and 4 were trained on the dev dataset for 5 epochs. Based on experimentation, the Wiki + Gigaword dataset appeared to outperform the Twitter dataset, regardless of embedding dimension.

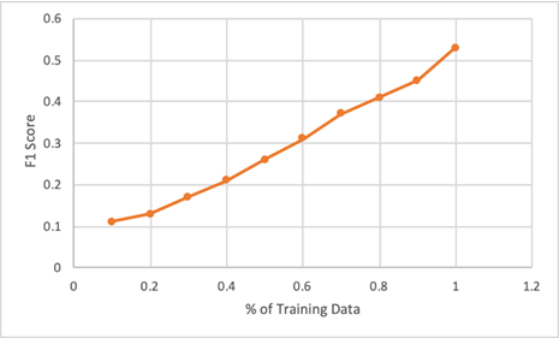
Table 3: Different Pre-trained Embeddings

GloVe File	F1 Score
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 50 dimensions	0.2834
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 100 dimensions	0.3658
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 200 dimensions	0.2773
Twitter 2B tweets, 27B tokens, 25 dimensions	0.2329
Twitter 2B tweets, 27B tokens, 50 dimensions	0.2542
Twitter 2B tweets, 27B tokens, 100 dimensions	0.2901

Table 4: Hidden Layer Performance

Hidden Layers	Embed Dims	F1 Score
200	100	0.2951
300	100	0.3658
400	100	0.3458

Learning Curves (4pt) From the graph below, we can see that as the percentage of training data increased, the F1 score increased proportionally.



Speed Analysis (4pt) Table 5 below shows the total time the model takes to go through an entire epoch, the total time per sentence, and the total time per token. Based on these calculations, we can estimate the algorithm speed to be approximately 4 mins / epoch * 30 epochs = 120 mins = 2 hours. The model was run on Google Colab with 12.72 GB RAM.

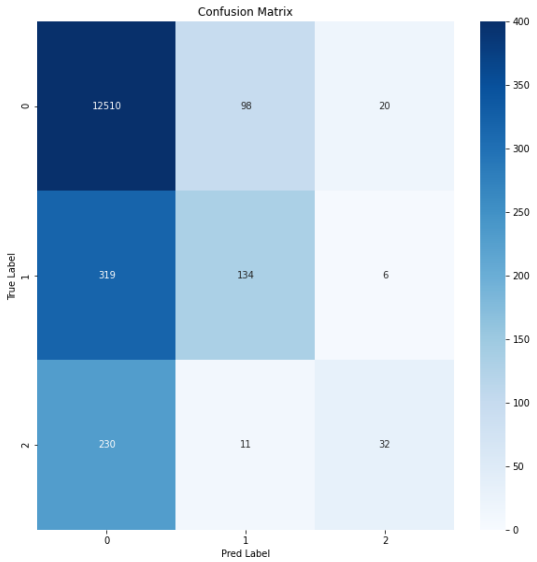
Table 5: Speed Analysis

Total Time	Time / Sentence	Time / Token
3:58	10.06 it/s	1.2575 tok/s
4:25	9.03 it/s	1.1288 tok/s
4:30	8.86 it/s	1.1075 tok/s

9 Analysis

Error Analysis (7pt) Looking at the first example in the dev set: ['rt', '@aleynastam', ':', 'i', "can't", 'wait', 'until', 'cedar', 'point', 'opens'], we can see that one category of mis-classifications occurs when named entities comprise words that also appear in regular expressions. For instance, the words 'cedar' and 'point' juxtaposed refer to the named entity of a theme park, but are more commonly seen as individual terms, which is what the network learns. Another category of mis-classifications stemmed from homonoyms, such as 'turkey' which may refer to the country, but which the network recognizes as food.

Confusion Matrix (5pt) Looking at the confusion matrix below, we can see that most tweets contain tokens that are not part of a named entity (indicated by the 12510 *O* tags).



10 Conclusion (3pt)

The largest improvement in NER task performance stemmed from CRF. Furthermore, character-level embeddings enriched contextual knowledge gleaned during neural network training phase, and also provided substantial improvements. Fine-tuning hyper-parameters (e.g. number of hidden layers, embedding dimensions, learning, rate, number of epochs) proved crucial in making incremental performance progress. For instance, fixing the embedding dimensions at 100 increased the F1 score by nearly 0.1, fixing the number of hidden layers at 300 increased the F1 score by nearly 0.15, and training for around 30 epochs seemed to be the sweet spot between underfitting and overfitting the model.