NLP202 assignment 3

Cheng-Tse Liu

March 28, 2024

1 Introduction

The goal of this assignment focuses on creating a Named Entity Recognition (NER) system to identify and categorize entities in text into predefined categories like people, locations, and organizations. Students are expected to work with the CoNLL 2003 shared task data, implementing both a linear model using the Perceptron algorithm with various features and a neural model using a Bidirectional LSTM with a Conditional Random Field (BiLSTM-CRF). They must use Viterbi decoding for sequence prediction and evaluate their models on precision, recall, and F1 metrics. Additionally, the neural model should explore character-level CNN embeddings and may consider alternative loss functions, such as SVM loss or ramp loss. The students are required to conduct experiments, use minibatch training, apply early stopping, and report their findings with comprehensive documentation, including any assumptions and hyperparameter values. They must also analyze the model's output against gold standard labels to understand error patterns and submit these observations along with the outputs for both the development (dev) and test datasets.

2 Part 1

When formulating the features, I could easily assign names to the features based on certain conditions being met. One of the few obstacles I encountered was ensuring that the features for the k-length prefix didn't activate for k values exceeding the word length, but this issue was resolved by simply iterating through the list of k values and refraining from triggering any feature for k values greater than the word length. For the decoder, I adopted a numpy array for the Viterbi and backpointer matrices, similar to my approach in previous assignments, with minor modifications to populate the Viterbi cells according to the maximum score of the current and preceding tags, adjusting for the input's different format compared to the hidden Markov model.

Implementing Stochastic Gradient Descent (SGD) became straightforward once I grasped the mathematical operations applicable to the FeatureVector object. I set up a training loop to process the dataset sequentially each epoch, applying the gradient computation at each step and updating the weights by invoking the times plus equal function, with -1 as the scalar, directing the gradient descent.

For Adagrad, the process was quite simple as well, necessitating only an additional FeatureVector to track the cumulative squared gradients. This would scale down future gradient updates. I introduced two new functions to FeatureVector: one to sum the square of the current gradient to the accumulating FeatureVector and another to take some value alpha, divide it by the square root of the accumulated squared value of the current feature, and then use this product to update the current gradient.

2.1 Model Performance Metrics

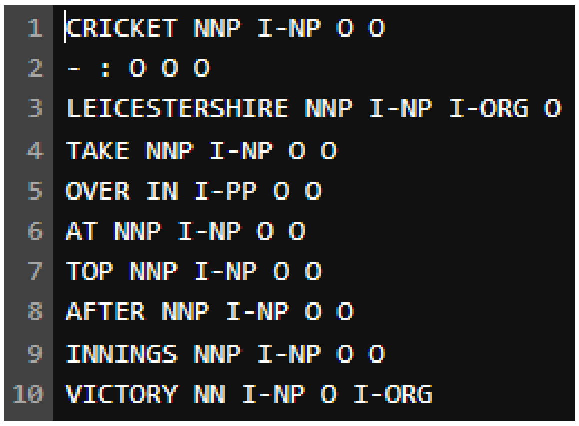
Table 4 showcases the performance for basic features SGD:

config	acc	recal	f1
SGD Dev	29.8	36.2	31.45
SGD test	23.7	29.78	26.67

Table 1: Performance for basic features SGD

2.2 Model Output

The following two images (Figure 1) are the sample of output file for basic features SGD.



```
CRICKET NNP I-NP O O
- : O O O
LEICESTERSHIRE NNP I-NP I-ORG O
TAKE NNP I-NP O O
OVER IN I-PP O O
AT NNP I-NP O O
TOP NNP I-NP O O
AFTER NNP I-NP O O
INNINGS NNP I-NP O O
VICTORY NN I-NP O I-ORG
```

Figure 1: Output Sample for SGD

2.3 Error Analysis

Observing our model's performance with just the initial set of features activated, which include the current and previous tags, the current word, and the part of speech, we've noticed it's quite adept at pinpointing the existence of a named entity. However, determining the exact type of entity is a challenge. Mistakes where an entity is completely missed or incorrectly ignored are infrequent. Yet, it's not uncommon for us to spot instances like 'Grace Road' being mislabeled as 'I-PER I-PER' instead of the correct 'I-LOC I-LOC'. Reflecting on our current feature set, this pattern is understandable. The model might have encountered 'Grace' as a person's name elsewhere in the data, and seeing 'I-PER' precede a proper noun probably reinforces the likelihood of a repeated 'I-PER'. This observation leads us to speculate whether including the previous part of speech might be beneficial, especially since it could provide additional context, like indicating a preceding preposition that might commonly lead into a location.

Conversely, our model seems quite proficient at correctly tagging 'I-ORG' entities. The training corpus appears to have ample instances where team names are mentioned, so it learns to associate certain names with sports teams effectively. Take 'Surrey', for example; it's a geographical location but also denotes a sports team. Our model generally classifies mentions of 'Surrey' correctly as 'I-ORG'. Upon reviewing the training set, we found that 'Surrey' is consistently tagged as 'I-ORG', which could lead to inaccuracies when it's actually referencing a place.

2.4 Full Feature Set Performance

Table 2 showcases the performance for full features SGD:

config	acc	recal	f1
SGD Dev	27.63	34.85	30.82
SGD test	22.23	29.45	25.34

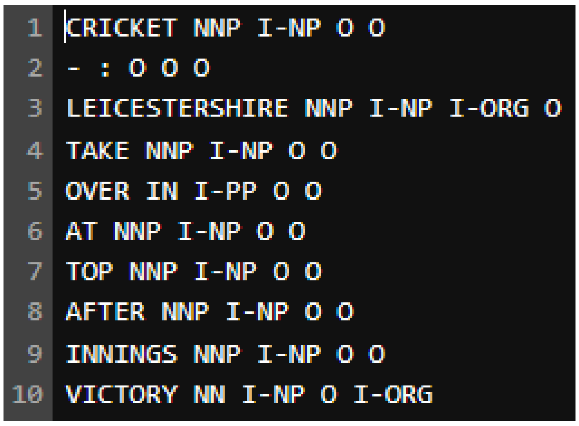
Table 2: Performance for full features SGD

2.5 Full Feature Set Output

The following two images (Figure 2) are the sample of output files for full features SGD.

2.6 Full Feature Set Error Analysis

When all features are turned on for the stochastic gradient descent (SGD) model, the output doesn't seem markedly different. 'Grace' remains an issue, and our model excels in similar areas as before. Interestingly, the simpler four-feature model performs slightly better than the full-featured one. Notably, the peak F1 score



```
CRICKET NNP I-NP O O
- : O O O
LEICESTERSHIRE NNP I-NP I-ORG O
TAKE NNP I-NP O O
OVER IN I-PP O O
AT NNP I-NP O O
TOP NNP I-NP O O
AFTER NNP I-NP O O
INNINGS NNP I-NP O O
VICTORY NN I-NP O I-ORG
```

Figure 2: Output Sample for SGD

of 25.34 was reached in the seventh epoch with fewer features, whereas the more complex model achieved it by the second epoch before plateauing. The added features didn't significantly address existing errors, but they did seem to stabilize performance over subsequent epochs.

2.7 Feature Weights Analysis

Looking at the strongest and weakest weights of our model, it's clear to see the trends that emerge in how it assesses and assigns tags. The highest weights are heavily associated with the 'B-PER' tag. For instance, this tag alone, when it starts a sentence or follows the start symbol, or is combined with the word 'EU', is significantly valued by the model. Even the presence of the letter 'E' at the beginning of a word tends to sway the model towards a 'B-PER' tag.
Conversely, the model seems to heavily penalize certain combinations when predicting 'I-LOC'. Names such as 'Sindh', 'Bells', and 'Newcastle' are generally not favored for this tag, which is intriguing and may point towards biases within our training data. Similarly, 'Baby' is a word that our model strongly disassociates from being an 'I-LOC' and even 'Arctic' is unlikely to be tagged as 'I-ORG'. These patterns might reflect the nuances and complexities of the data the model was trained on.
The model's heavy weighting against 'I-LOC' following 'Sindh' or 'Bells', and 'I-ORG' associated with 'Arctic', could be indicative of an uneven distribution of examples in the training set. It's also fascinating to see that 'Newcastle', typically a location, is weighted against 'I-PER'. This could imply a learning from the data that the model needs to be aware of multiple potential meanings of a word depending on context.
In essence, the model's weight distribution highlights the importance of certain tags and word associations while revealing potential areas where the training data may not be providing a balanced representation. This insight could be essential for understanding the model's behavior and for guiding improvements in its training regimen.
Table 3 showcases the top 5 and bottom 5 weights:

Top 5	Bottom 5
t=B-PER	t=I-LOC+w=Sindh
ti=B-PER+ti-1=¡START¿	t=O+w=Baby
t=B-PER+w=EU	t=I-LOC+w=Bells
prei=+t=B-PER	t=I-ORG+w=Arctic
prei=E+t=B-PER	t=I-PER+w=Newcastle

Table 3: Feature weights

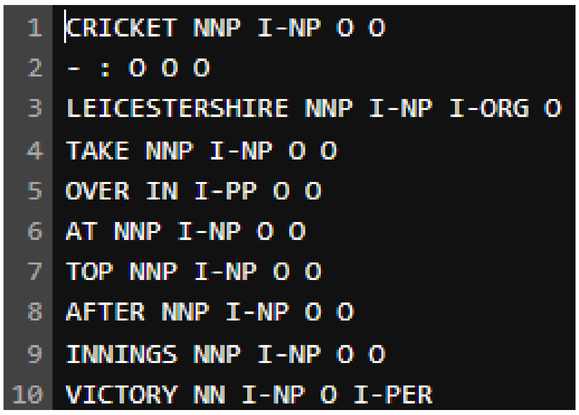
2.8 Adagrad

Table 4 showcases the performance for Adagrad:

The following images (Figure 3) are the sample of output files for Adagrad.

config	acc	recal	f1
Adagrad Dev	36.87	47.64	41.57
Adagrad test	30.53	41.86	35.31

Table 4: Performance for basic features Adagrad



```
CRICKET NNP I-NP O O
- : O O O
LEICESTERSHIRE NNP I-NP I-ORG O
TAKE NNP I-NP O O
OVER IN I-PP O O
AT NNP I-NP O O
TOP NNP I-NP O O
AFTER NNP I-NP O O
INNINGS NNP I-NP O O
VICTORY NN I-NP O I-PER
```

Figure 3: Output Sample for Adagrad

3 Part 2

In the second part of our project, we adapted the Pytorch starter code to create our LSTM CRF model suitable for our dataset which included minibatching. Initially, we preprocessed our tokens using torchtext's vocabulary object, which was also applied to the tagset and characters for the CNN element we added later. By encoding the inputs numerically, we were able to compile them into a Pytorch tensor, applying padding where necessary.

Once our model received the batched inputs, it processed them through the LSTM layer, aided by functions to manage the padded sequences. However, the starter code's functions for the forward algorithm and the decoder needed adjustments to handle batch lengths. We modified these functions to operate according to the specific length of each input within the batch, thus ensuring the Viterbi algorithm and forward calculation were performed correctly for each sequence, ignoring any padding.

For the CNN layer, we first decoded the batched tensor back to strings, then broke it down into characters. These characters were then encoded using a character-specific vocabulary and re-padded to form a new batched tensor that could be fed into the CNN layer. The CNN's output provided a tensor representing each word in the batch, with a set length appropriate for combining with the batched encoded tensor from the LSTM. By stacking these outputs, we formed a tensor shaped to match our sequence requirements, with an added dimension to accommodate the CNN output. This final tensor, combining both LSTM embeddings and CNN output, was then ready to be processed by the LSTM to complete our sequence tagging.

Table 5 showcases the performance for BiLSTM CRF with CNN:

	acc	recal	f1
Dev	73.03	71.4	72.13
Test	65.51	60.66	62.86

Table 5: Performance for BiLSTM CRF CNN

For BiLSTM CNN CRF, the errors identified in the BiLSTM CRF were corrected, although the issue with 'Kent' persisted, albeit with a different incorrect label of 'I-ORG I-ORG'.

The comparison of the two models highlighted a few noteworthy points:
The CNN's processing of words that were always represented as '¡UNK¿' seemed to be ineffective. This was

likely due to the way we encoded the tokens for the CNN layer, which might have benefited from using the raw, uncoded strings in conjunction with the encoded tensors. There's a possibility that the CNN layer wasn't handling padding optimally. Words of different lengths within the same batch could lead to embeddings influenced excessively by '¡PAD¿' tokens. While we are sure that the LSTM component is dealing with padding correctly, we are curious if there's a way to efficiently process batched tensors through the Viterbi and forward algorithms.