

NLP 202 Assignment 3

Ishika Kulkarni
ikulkar1@ucsc.edu

Abstract

This assignment explores the application of linear models and loss functions for structured prediction tasks using Bidirectional Long Short-Term Memory (BiLSTM) combined with Conditional Random Fields (CRF). To optimize the model's performance, the focus is on implementing and evaluating various training algorithms, including Structured Perceptron (with SSGD and Adagrad) and Structured SVM. The report details the dataset, feature engineering, and the Viterbi algorithm for decoding, concluding with an analysis of the results and potential improvements.

1 Introduction

Named Entity Recognition (NER) is a crucial task in Natural Language Processing (NLP) that focuses on identifying and classifying entities such as names of people, locations, organizations, and other miscellaneous entities within the text. This assignment aims to implement a BIO tagger for NER using linear models and structured prediction techniques. The objective is to create a system that accurately tags entities in the CoNLL-2003 dataset, a widely recognized benchmark for NER tasks.

The system uses a BiLSTM with CRF model sequential dependencies and enhances tagging accuracy. Key components of the implementation include feature engineering, the Viterbi algorithm for decoding, and training the model using structured perceptron with Stochastic Subgradient Descent (SSGD) and Adagrad optimizers, as well as structured Support Vector Machines (SVM).

2 Dataset

The dataset mentioned above used in this assignment is the CoNLL-2003 shared task dataset, a standard benchmark for Named Entity Recognition (NER). The dataset consists of annotated text in four columns, separated by spaces.

```
ATHENS NNS I-NP I-LOC  
1996-08-22 CD I-NP 0
```

Figure 1: Training Data

```
We PRP I-NP 0  
have VBP I-VP 0  
taken VBN I-VP 0  
those DT I-NP 0  
prudent JJ I-NP 0  
planning NN I-NP 0  
steps NNS I-NP 0  
. . 0 0  
" " 0 0
```

Figure 2: Development Data

```
All DT I-NP 0  
four CD I-NP 0  
teams NNS I-NP 0  
are VBP I-VP 0  
level NN I-NP 0  
with IN I-PP 0  
one CD I-NP 0  
point NN I-NP 0  
each DT B-NP 0  
from IN I-PP 0  
one CD I-NP 0  
game NN I-NP 0  
. . 0 0
```

Figure 3: Test Data

3 Base code - Tagger

• Model Architecture

- The model is implemented as a BiLSTM-CRF class, which inherits from `nn.Module` in PyTorch.
- It consists of the following layers:
 - * **Word Embeddings:** An embedding layer (`nn.Embedding`) to map words to dense vectors of size `embedding_dim`.
 - * **BiLSTM Layer:** A bidirectional LSTM layer (`nn.LSTM`) to capture contextual information from both past and future tokens. The hidden dimension is split into two for bidirectional processing.
 - * **Linear Layer:** A fully connected layer (`nn.Linear`) to map the LSTM output to the tag space.
 - * **Transition Matrix:** A learnable transition matrix (`nn.Parameter`) for the CRF, which models the likelihood of transitioning from one tag to another.

• Key Functions

- `_forward_alg`: Implements the forward algorithm for computing the partition function (log-sum-exp) of the CRF. This is used during training to calculate the likelihood of the tag sequence.
- `_get_lstm_features`: Extracts features from the BiLSTM for a given sentence. The LSTM processes the word embeddings and outputs a sequence of feature vectors, which are then passed through a linear layer to produce scores for each tag.
- `_score_sentence`: Computes the score of a given tag sequence using the CRF transition matrix and emission scores from the BiLSTM.
- `_viterbi_decode`: Implements the Viterbi algorithm for decoding the most likely sequence of tags given the emission scores and transition matrix.
- `neg_log_likelihood`: Computes the negative log-likelihood loss for a given sentence and its gold-standard tags. This is the objective function used during training.

- `forward`: Performs inference by running the Viterbi algorithm on the BiLSTM features to predict the most likely tag sequence.

• Character-Level Features

- The model uses character-level features using a CNN-based character embedding layer.
- Character embeddings are extracted using a 1D convolutional layer (`nn.Conv1d`) and max-pooling, which are then concatenated with word embeddings for enhanced feature representation.

• Data Preparation

- The `prepare_sequence` function converts a sequence of tokens or tags into their corresponding indices using predefined mappings (`word_to_ix` and `tag_to_ix`).
- The `make_data_point` function processes raw CoNLL-2003 data into a structured format.
- The `read_data` function reads the CoNLL-2003 dataset from a file and converts it into a list of dictionaries, where each dictionary represents a sentence.

• Training Loop

- The model is trained using the Adam optimizer with a learning rate 0.001.
- Training is performed in mini-batches of size 128. For each batch:
 - * Sentences and tags are converted into tensors using `prepare_sequence`.
 - * The negative log-likelihood loss is computed for each sentence-tag pair.
 - * Gradients are backpropagated, and the model parameters are updated using `optimizer.step()`.
- Training is run for 5 epochs, and the model is saved after each epoch.

```

accuracy: 0.56%; (non-0)
accuracy: 81.72%; precision: 2.17%; recall: 0.39%; FB1: 0.66
LOC: precision: 0.00%; recall: 0.00%; FB1: 0.00 13
MISC: precision: 2.31%; recall: 0.88%; FB1: 1.27 346
ORG: precision: 0.00%; recall: 0.00%; FB1: 0.00 13
PER: precision: 2.19%; recall: 0.82%; FB1: 1.19 686
(2.1739130434782608, 0.38871049518337, 0.6594982078853047)

```

Figure 4: Tagger Results Validation Data

```

accuracy: 0.49%; (non-0)
accuracy: 81.97%; precision: 1.67%; recall: 0.25%; FB1: 0.44
LOC: precision: 0.00%; recall: 0.00%; FB1: 0.00 7
MISC: precision: 1.58%; recall: 0.55%; FB1: 0.81 316
ORG: precision: 0.00%; recall: 0.00%; FB1: 0.00 7
PER: precision: 1.76%; recall: 0.55%; FB1: 0.83 568
(1.670378619153675, 0.2535068446848065, 0.4402054292002935)

```

Figure 5: Tagger Results Test Data

• Inference

- After training, the model is loaded from the saved checkpoint (bilstm_crf_model_epoch_4.pth).
- Inference is performed on the development and test sets using the forward method, which predicts each sentence's most likely tag sequence.
- Predicted tags are written to dev_predictions.txt and test_predictions.txt.

• Hyperparameters

- Embedding Dimension: 40.
- Hidden Dimension: 40.
- Character Embedding Dimension: 4.
- Batch Size: 128.
- Learning Rate: 0.001.

```

0 I-PER B-MISC 0 0 0 0 0
0 0 0 0
0 I-PER B-MISC 0 0 0 0 0 0 0
0 I-PER B-MISC 0 0 0 0 0 0 0
0 I-PER 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

```

Figure 6: Tagger Dev Predictions

```

0 0 0 0 0 0 0
0 I-PER B-MISC 0 0 0 0
0 0 0 0 0 0
0 I-PER B-MISC 0 0 0 0 0
0 I-PER B-MISC 0 I-PER 0 0 0 0
0 0 0 0 0 0

```

Figure 7: Tagger Test Predictions

4 Features

• Implementation Overview

- The Named Entity Recognition (NER) system is implemented using a structured perceptron model with a Viterbi decoder.
- The code is structured into several key components:

• Feature Extraction

- The Features class is responsible for computing features for a given token and its context.
- It uses the following inputs:
 - * **inputs**: A dictionary containing tokens, POS tags, chunk tags, and gold-standard NER tags.
 - * **feature_names**: A list of feature names to be computed (e.g., current word, previous tag, POS tag, etc.).
- The compute_features method computes the feature vector for a given token and its context. It uses the following features:
 - * **Current tag**: The current NER tag.
 - * **Previous tag**: The previous NER tag.
 - * **Current word**: The current token.
 - * **POS tag**: The part-of-speech tag of the current token.
 - * **Word shape**: A representation of the word's shape (e.g., capitalization, digits, etc.).
 - * **Prefixes**: Length-k prefixes of the current word (for $k = 1, 2, 3, 4$).
 - * **Gazetteer**: Whether the current word is in a gazetteer for the current tag.
 - * **Capitalization**: Whether the current word starts with a capital letter.

- **Training**

- **Mini-Batches**

- * The training data is divided into mini-batches of size 128.
 - * This allows the model to update its parameters more frequently, leading to faster convergence.
 - * For each mini-batch:
 - Sentences and tags are converted into tensors using `prepare_sequence`.
 - The negative log-likelihood loss is computed for each sentence-tag pair.
 - Gradients are backpropagated, and the model parameters are updated using `optimizer.step()`.

- **Parameters**

- * The model parameters are stored in a `FeatureVector` object, which maps feature names to their corresponding weights.
 - * The parameters are updated using the gradient of the perceptron loss, computed by the `perceptron_gradient` function.

- **Hyperparameters**

- * **Learning Rate:** The learning rate for SSGD is set to 1.0, while Adagrad uses an adaptive learning rate based on the sum of squared gradients.
 - * **Epochs:** The model is trained for 10 epochs.
 - * **Batch Size:** The batch size is set to 128.

- **Early Stopping**

- * The `training_observer` function evaluates the model on the development set after each epoch and saves the model parameters if the F1 score improves.
 - * This helps prevent overfitting and ensures that the model generalizes well to unseen data.

- **Decoding**

- The `decode` function implements the Viterbi algorithm to find the most likely sequence of tags given the emission scores and transition matrix.
 - The `predict` function uses the trained model to predict tags for a given input sequence.

- **Evaluation**

- The `evaluate` function computes precision, recall, and F1 score using the `conllevaluate` function.
 - The `write_predictions` function writes the predicted tags to a file in CoNLL format.

4.1 Features - 1 to 4

- **Current Word (`current_word`)**

- The current token being tagged.

- **Previous Tag (`prev_tag`)**

- The tag of the previous token in the sequence.

- **POS Tag (`curr_pos_tag`)**

- The part-of-speech tag of the current token.

- **Tag (`tag`)**

- The current NER tag is being predicted.

4.2 All Features

Previous four features along with new ones.

- **Word Shape (`shape_curr_word`)**

- A representation of the word's shape.

- **Prefixes (`len_k`)**

- Length-k prefixes of the current word (for $k = 1, 2, 3, 4$).

- **Gazetteer (`in_gazetteer`)**

- Whether the current word is in a gazetteer for the current tag.

- **Capitalization (`start_cap`)**

- Whether the current word starts with a capital letter.

5 Results

5.1 SSGD Four Features Output

Epoch	Accuracy (non-O)	Accuracy	Found Phrases	Correct Phrases
0	40.38%	79.94%	2053	540
1	24.36%	76.20%	2812	451
2	5.41%	74.01%	1488	107
3	14.62%	75.68%	1721	245
4	50.74%	80.38%	2788	713
5	49.96%	80.58%	2629	685
6	48.43%	80.94%	2601	700
7	32.04%	78.20%	2704	503
8	32.04%	78.68%	1995	476
9	30.76%	78.06%	2655	512

Table 1: Model Performance Across Epochs For SSGD with four features while Training

Metric	Value
Processed Tokens	51578
Total Phrases	5917
Found Phrases	7002
Correct Phrases	1566
Accuracy (non-O)	29.86%
Accuracy	80.39%
Precision	22.37%
Recall	26.47%
F1 Score (FB1)	24.24
LOC Precision	78.13%
LOC Recall	40.22%
LOC F1 Score	53.10
MISC Precision	18.06%
MISC Recall	7.33%
MISC F1 Score	10.43
ORG Precision	29.21%
ORG Recall	16.03%
ORG F1 Score	20.70
PER Precision	11.06%
PER Recall	29.91%
PER F1 Score	16.15

Table 2: Results for SSGD on Development Set on 4 Features

Thus, the model cannot determine the correct tags on the unseen data in the 1100 examples; it was trained on the limited feature set.

The results for this run are saved in the results folder, and the files ner.dev.out9 is the result for

Metric	Value
Processed Tokens	46666
Total Phrases	5616
Found Phrases	7149
Correct Phrases	1301
Accuracy (non-O)	26.80%
Accuracy	77.67%
Precision	18.20%
Recall	23.17%
F1 Score (FB1)	20.38
LOC Precision	76.96%
LOC Recall	41.30%
LOC F1 Score	53.75
MISC Precision	12.89%
MISC Recall	5.85%
MISC F1 Score	8.05
ORG Precision	26.28%
ORG Recall	10.63%
ORG F1 Score	15.13
PER Precision	7.53%
PER Recall	24.78%
PER F1 Score	11.55

Table 3: Results for SSGD on Test Set with 4 Features

```
print(data[display_id]['tokens'])
✓ 0.0s
['<START>', 'AL-AIN', ',', 'United', 'Arab', 'Emirates', '1996-12-06', '<STOP>']

print(all_gold_tags[display_id])
✓ 0.0s
['I-LOC', 'O', 'I-LOC', 'I-LOC', 'I-LOC', 'O']

print(all_predicted_tags[display_id])
✓ 0.0s
['O', 'O', 'I-LOC', 'O', 'I-MISC', 'B-PER']
```

Figure 8: Gold vs Predicted Output For 4 Features

the development set, and model.4features.txt is the result for the test set.

5.2 SSGD All Features Output

Epoch	Accuracy (non-O)	Accuracy	Found Phrases	Correct Phrases
0	40.38%	77.32%	313	67
1	31.92%	77.02%	276	53
2	26.54%	75.65%	277	42
3	48.08%	78.77%	298	78
4	44.23%	79.83%	271	77
5	36.15%	77.09%	278	56
6	35.77%	76.64%	285	63
7	39.23%	77.17%	287	60
8	31.15%	76.03%	272	55
9	41.15%	77.47%	295	72

Table 4: Model Performance Across Epochs For SSGD with All Features while Training

Metric	Value
Processed Tokens	51578
Total Phrases	5917
Found Phrases	7975
Correct Phrases	1949
Accuracy (non-O)	36.78%
Accuracy	81.79%
Precision	24.44%
Recall	32.94%
F1 Score (FB1)	28.06
LOC Precision	55.78%
LOC Recall	46.17%
LOC F1 Score	50.52
MISC Precision	30.63%
MISC Recall	20.68%
MISC F1 Score	24.69
ORG Precision	37.97%
ORG Recall	18.12%
ORG F1 Score	24.53
PER Precision	12.92%
PER Recall	36.68%
PER F1 Score	19.10

Table 5: Results for SSGD on Development Set on All Features

The results are saved in the folder `results_features`. The model's performance has improved as it can understand locations and other labels, although we still have some misclassified examples.

The top and bottom five features suggest that STOP is an essential feature as it shows up fre-

Metric	Value
Processed Tokens	46666
Total Phrases	5616
Found Phrases	8312
Correct Phrases	1757
Accuracy (non-O)	35.85%
Accuracy	79.24%
Precision	21.14%
Recall	31.29%
F1 Score (FB1)	25.23
LOC Precision	53.43%
LOC Recall	46.34%
LOC F1 Score	49.63
MISC Precision	19.22%
MISC Recall	15.55%
MISC F1 Score	17.19
ORG Precision	35.65%
ORG Recall	15.24%
ORG F1 Score	21.35
PER Precision	11.17%
PER Recall	39.01%
PER F1 Score	17.37

Table 6: Results for SSGD on Test Set on All Features

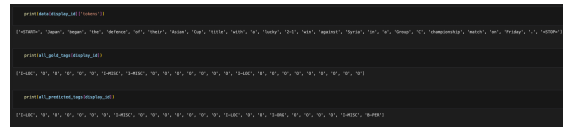


Figure 9: Gold vs Predicted Output For All Features

quently. In contrast, the bottom features are weakly correlated with the target variables and other features.

5.2.1 Interesting Features

- The features like `t=I-PERsaaaaaaaa` and `t=I-LOCsaaaaaaaa` highlight how the model uses **word shape patterns** (e.g., capitalization) to identify named entities such as persons (I-PER) and locations (I-LOC). These features are crucial for recognizing entities with specific capitalization rules, such as "McDonald" (person) or "San Francisco" (location).
- Features like `t=I-MISCsaaaaaaaa` and `t=I-ORGsaaaaaaaa` demonstrate the model's ability to capture **mixed-case patterns** in miscellaneous (I-MISC) and organization (I-ORG) entities. This helps the model

generalize across rare or complex entities, such as product names or company names.

- Features like $t=0+PRE4=True$ and $t=0+PRE3=True$ show how the model uses **prefix patterns** (e.g., "Thu" or "Thur") to identify non-entities (0). These features help the model avoid misclassifying common words (e.g., days of the week) as named entities.
- The feature $C=0+w-Thur\ sday$ indicates how the model associates specific words (e.g., "Thur sday") with the 0 tag. This helps the model correctly classify non-entities, even when the input contains typos or unusual formatting.

```
10020 ti=<STOP>+ti-1=0
7777 t=0
6558 t=0sid
4200 t=0+PRE1=.
4160 t=0+w=.
```

Figure 10: Top 5 Features For SSGD with All Features

```
-4200 t=B-PER+PRE1=.
-5100 ti=B-PER+ti-1=0
-6598 t=B-PERsid
-11000 ti=<STOP>+ti-1=B-PER
-14420 t=B-PER
```

Figure 11: Bottom 5 Features For SSGD with All Features

```
241 t=I-PERsiaaaaaaa
240 t=I-LOCsiaaaaaaa
222 t=I-MISCsiaaaaaaa
221 t=I-ORGsiaaaaaaa
```

Figure 12: Interesting Feature 1

```
t=0+w=Thursday
t=0+PRE4=Thur
t=0+PRE3=Thu
```

Figure 13: Interesting Feature 2

5.3 Adagrad Output

Epoch	Accuracy (non-O)	Accuracy	Found Phrases	Correct Phrases
0	42.40%	82.61%	8492	2357
1	44.32%	83.04%	7969	2359
2	42.36%	82.71%	7877	2256
3	39.94%	82.35%	7609	2166
4	38.57%	82.11%	7658	2136
5	38.59%	82.15%	7596	2123
6	37.42%	81.93%	7762	2094
7	36.58%	81.82%	7630	2057
8	35.12%	81.60%	7443	1986
9	35.07%	81.58%	7488	1979

Table 7: Model Performance Across Epochs for Adagrad on all features while training

Metric	Value
Processed Tokens	51578
Total Phrases	5917
Found Phrases	7488
Correct Phrases	1979
Accuracy (non-O)	35.07%
Accuracy	81.58%
Precision	26.43%
Recall	33.45%
F1 Score (FB1)	29.53
LOC Precision	57.29%
LOC Recall	49.62%
LOC F1 Score	53.18
MISC Precision	58.19%
MISC Recall	33.04%
MISC F1 Score	42.15
ORG Precision	53.74%
ORG Recall	20.88%
ORG F1 Score	30.08
PER Precision	10.06%
PER Recall	26.69%
PER F1 Score	14.61

Table 8: Result for Adagrad on Development Set with all features

The results for Adagrad runs are saved in the adagrad_results folder, where ner.dev.out is the development test result and ner.test.out is the test set result.

Metric	Value
Processed Tokens	46666
Total Phrases	5616
Found Phrases	7763
Correct Phrases	1711
Accuracy (non-O)	32.94%
Accuracy	79.03%
Precision	22.04%
Recall	30.47%
F1 Score (FB1)	25.58
LOC Precision	54.16%
LOC Recall	51.62%
LOC F1 Score	52.86
MISC Precision	40.44%
MISC Recall	23.82%
MISC F1 Score	29.98
ORG Precision	46.77%
ORG Recall	16.27%
ORG F1 Score	24.14
PER Precision	8.02%
PER Recall	25.97%
PER F1 Score	12.25

Table 9: Result for Adagrad on Test Set with all features

```

CRICKET NNP I-NP 0 0
- : 0 0 0
LEICESTERSHIRE NNP I-NP I-ORG 0
TAKE NNP I-NP 0 0
OVER IN I-PP 0 0
AT NNP I-NP 0 0
TOP NNP I-NP 0 0
AFTER NNP I-NP 0 0
INNINGS NNP I-NP 0 0
VICTORY NN I-NP 0 I-ORG
. . 0 0 B-PER

```

Figure 14: Adagrad Development Set Predictions

```

SOCCER NN I-NP 0 0
- : 0 0 0
JAPAN NNP I-NP I-LOC 0
GET VB I-VP 0 0
LUCKY NNP I-NP 0 0
WIN NNP I-NP 0 0
, , 0 0 0
CHINA NNP I-NP I-PER I-LOC
IN IN I-PP 0 0
SURPRISE DT I-NP 0 0
DEFEAT NN I-NP 0 I-ORG
. . 0 0 B-PER

```

Figure 15: Adagrad Test Set Predictions

6 SVM Training

6.1 Implementation

The model is a Bidirectional LSTM-CRF (BiLSTM-CRF) with Structured SVM Training.

- **BiLSTM:** Captures contextual information from both past and future tokens.
- **CRF:** Models dependencies between tags to ensure valid tag sequences.
- **Structured SVM Loss:** Uses a Hamming distance cost function to penalize incorrect predictions during training.

The training process is as follows:

- **Cost-Augmented Decoding:**
 - During training, the model performs cost-augmented decoding using the Viterbi algorithm.
 - The Hamming distance cost function penalizes incorrect predictions by a factor of 10.
- **Early Stopping:**
 - Training stops early if the F1 score on the development set does not improve for a specified number of epochs (patience=1).
- **Parameter Tuning:**
 - The model is trained with different combinations of learning rates and regularization strengths.
 - The best configuration is selected based on the F1 score on the development set.
- **Batch Training:**
 - Training is performed in mini-batches of size 1024 to improve efficiency.

The result is saved in svm_results.