NLP with Convolutional Neural Networks (RNNs)

Alviona Mancho	f3322405	alv.mantso@aueb.gr / alviona.mantso@gmail.com
Anna Chatzipapadopoulou	f3322411	ann.chatzipapadopoul@aueb.gr / annachatzipap@gmail.com

Natural Language Processing, 2024-2025[1]

Notebook link (ex. 2)

Notebook link (ex. 3)

1. Exercise 2 (Sentiment Analysis using a Multikernel Stacked CNN)

For this exercise we used the **IMDb Movie Reviews** dataset. The implementation includes an CNN classifier in PyTorch.

Dataset Composition

- **Content**: The dataset contains movie reviews from IMDb, which are labeled as either positive or negative.
- **Labels**: It's a binary classification dataset, with labels 0 (negative) and 1 (positive).
- **Vocabulary size**: Is set to 10,000.
- **Average document (review) length**: 234.76 words
- **Data Split**: The dataset has 25,000 reviews for training and 25,000 for testing, ensuring a balanced split of positive and negative reviews in each set. We further divide the testing data into a development (or validation) set and a final test set, each containing 12,500 reviews.

```
Average review length: 234.76 words

Randomly selected examples
1. Review: [bos] i second the motion to make this into a movie it would be great i was also amazed at the storyline and characte
   Sentiment: Positive

2. Review: [bos] i used this film in a religion class i was teaching the golden fish is swimming happily in his bowl in an upper
   Sentiment: Positive

3. Review: [bos] i have to admit i did not finish this movie because it was so amazingly stupid and not worth watching i watched
   Sentiment: Negative

4. Review: [bos] i really wanted to like this movie i absolutely love kenny [oov] and [oov] rice has a charming side to him not
   Sentiment: Negative

5. Review: [bos] this must have been one of the worst movies i have ever seen br br i have to disagree with another [oov] who sa
   Sentiment: Negative
```

[1] *Both team members engaged equally in all tasks, collaborating synchronously throughout the assignment.*

Data preprocessing and Feature Extraction:

The class `DataPreprocessor` handles the preprocessing of the data. It features the following methods:

- `fetch_data(num_words):` Retrieves the IMDb dataset, limits the vocabulary size to the specified `num_words`, and splits the data into training, testing, and development sets. It also computes the average review length across all sets. Additionally, it converts each review from a sequence of word indices into readable text using a word index dictionary, where certain indices are mapped to special tokens (e.g., `[pad], [bos], [oov]`). Finally, it returns the processed training, testing, and development data, along with the average review length.
- `preprocess_data(X):` Preprocesses the text data by cleaning, tokenizing, and lemmatizing the words based on their parts of speech to improve the quality of the input data.
- `build_vocab(X_tokenized, max_words=100000):` Creates a vocabulary of the most common tokens from tokenized sentences, assigning unique indices to each token. It includes special tokens <pad> (for padding) and <unk> (for unknown tokens).
- `encode_and_pad(X_tokenized, vocab, max_length=250):` Encodes and pads tokenized sentences using the vocabulary.
- `prepare_embedding_matrix(vocab,word2vec_model,embedding_dim=300, max_words=100000):` Prepares the embedding matrix for the vocabulary using a pre-trained Word2Vec model.
- `prepare_data_loaders(train_data, val_data, test_data, y_train, y_val, y_test, batch_size=64):` Prepares PyTorch DataLoaders for training, validation, and testing.
- `generate_embeddings(X, model, embedding_dim = 300):` Generates word embeddings for the given documents using a pre-trained word embedding model. Each document is represented as the average of its word vectors.

Initially, the data is fetched using `fetch_data(num_words),` where `num_words` is set to 10,000, namely the vocabulary size. As described above, this method converts each review from a sequence of word indices into readable text using a word index dictionary, where certain indices are mapped to special tokens (e.g., `[pad], [bos], [oov]`). Finally, it returns the processed training, testing, and development data, containing 25,000, 12,500 and 12,500 reviews respectively. Subsequently, the training, testing and development sets are passed to `preprocess_data,` which cleans and standardizes text data by removing special characters and single characters, normalizing whitespace, and converting text to lowercase. It then tokenizes the text, tags parts of speech for improved lemmatization, and lemmatizes words. Next, the vocabulary is built from the training data (`build_vocab`) and all datasets are encoded and padded using the vocabulary (`encode_and_pad`). Then, the embedding matrix for the vocabulary is prepared using `word2vec-google-news-300` model from Gensim's API to obtain 300-dimensional word embeddings.

MultikernelCNN:

This class implements multi-kernel convolutional neural network (CNN). It supports dynamic n-gram convolutional layers with residual connections, global max pooling, and optional pre-trained embeddings.

- `__init__` (Constructor): Initializes the model with specified parameters and constructs its layers.
    - **input_dim:** Size of input embedding vectors.
    - **n_classes:** Number of target classes for classification.
    - **filters:** Number of filters (channels) per convolutional layer.
    - **max_words:** Vocabulary size for the embedding layer.
    - **n_values:** List of n-gram sizes for convolutional kernels.
    - **stacks:** Number of stacked convolutional layers for each n-gram size.
    - **dropout_prob:** Dropout probability for regularization.
    - **matrix_embeddings:** Optional pre-trained embedding matrix for the embedding layer.

- `forward` (Forward Pass):
  Implements the forward pass:
    - Takes a batch of text sequences as input.
    - Embeds the input sequences using the embedding layer.
    - Applies dropout for regularization.
    - Processes the embeddings through multiple convolutional layers, each corresponding to a different n-gram size, with residual connections for stacked layers.
    - Pools the convolutional outputs using global max pooling.
    - Concatenates the pooled outputs and applies dropout.
    - Passes the result through the output layer to produce logits for classification.

Classifier_Wrapper Class:

This class provides a training and evaluation framework for the CNN model, as well as for the MLP and BiLSTM models. It integrates the training process, prediction of class labels, and prediction of class probabilities.

- `fit`(Training):
  Trains the model over the specified number of epochs using the provided DataLoader:
    - Performs forward passes, calculates the loss using the criterion, and updates model weights using the optimizer.
    - Tracks and prints the average loss for each epoch.
    - Returns a **history** dictionary containing the loss values for each epoch.
- `predict` (Class Prediction): Predicts class labels for a given input dataset X.
- `predict_proba` (Probability Prediction): Predicts class probabilities for a given input dataset X, using softmax.

Utils Class:

This utility class provides static methods to train, evaluate, and validate a model. It supports detailed metric tracking, model checkpointing, and reporting of accuracy and F1 scores.

- `train_and_evaluate:` This method trains an model instance on a given training dataset and evaluates its performance on a test dataset.
    - The training phase includes iterative optimization of model parameters using the provided optimizer and loss function over a specified number of epochs. During each epoch, the average training loss is computed and displayed to monitor progress.
    - In the evaluation phase, the model's predictions on the test dataset are compared with the true labels to compute accuracy and F1 score. The method collects all predictions and true labels during testing to calculate these metrics comprehensively. The final test accuracy and F1 score are printed.

- `train_and_validate:` This method provides a more detailed training and validation process, tracking metrics for both datasets across epochs and saving the best-performing model. During each epoch, the model's performance is evaluated on the training dataset, with metrics such as loss, accuracy, and F1 score being computed. Simultaneously, the validation set is used to assess how well the model generalizes, using the same metrics.

  The method also implements a checkpoint mechanism, where the model with the lowest validation loss is saved to a file. If a new best model is found, the previous checkpoint is deleted to conserve resources. This ensures that the best model is always available for deployment or further evaluation. Additionally, all training and validation metrics are logged in a history dictionary. At the end of training, the history and path to the best model are returned.

Experiments:

The primary objective of our experiments is to assess and compare the performance of our CNN classifier against four baseline models: the BiLSTM RNN classifier implemented in the previous assignment, the MLP classifier, as implemented in a previous assignment, a majority classifier and the top-performing classifier identified in a previous assignment (i.e. the Logistic Regression with SGD). First, we configure the CNN classifier with initial, arbitrarily chosen hyperparameters. This initial testing allows us to observe baseline performance and helps us identify potential adjustments for further tuning. Subsequent steps involve tuning these hyperparameters for optimal performance.

Hyperparameter tuning our CNN Classifier:

The hyperparameter tuning is performed using a random search approach. A predefined number of random samples (n_iter = 50) is selected from the hyperparameter space.

Hyperparameters Tuned:

- **Filters (`filters`)**: Number of convolutional filters applied per kernel.
- **n-Gram Sizes (`n_values`)**: Specifies the range of n-gram sizes used in convolutional kernels.
- **Number of Stacked Layers (`stacks`)**: Defines how many convolutional layers are stacked for each n-gram size.
- **Dropout Probability (`dropout_prob`)**: Dropout rate applied during training to prevent overfitting.
- **Learning Rate (`learning_rate`)**: Controls the step size for the optimizer during weight updates.

Procedure

1. **Data Preparation**: First, input data (pre-trained word embeddings) is transformed into tensors and then is batched using a data loader to ensure efficient feeding into the model during training.
2. **Model Initialization**: The `MultikernelCNN` model is instantiated with a randomly selected hyperparameter configuration from the defined search space.
3. **Training**: The model is trained for a fixed number of epochs (`epochs=5`) using:
   - **Optimizer:** Adam optimizer.
   - **Loss Function:** Cross-entropy loss.
4. **Evaluation**: After training, the model is evaluated on the validation set, where the macro F1 score is the primary evaluation metric.

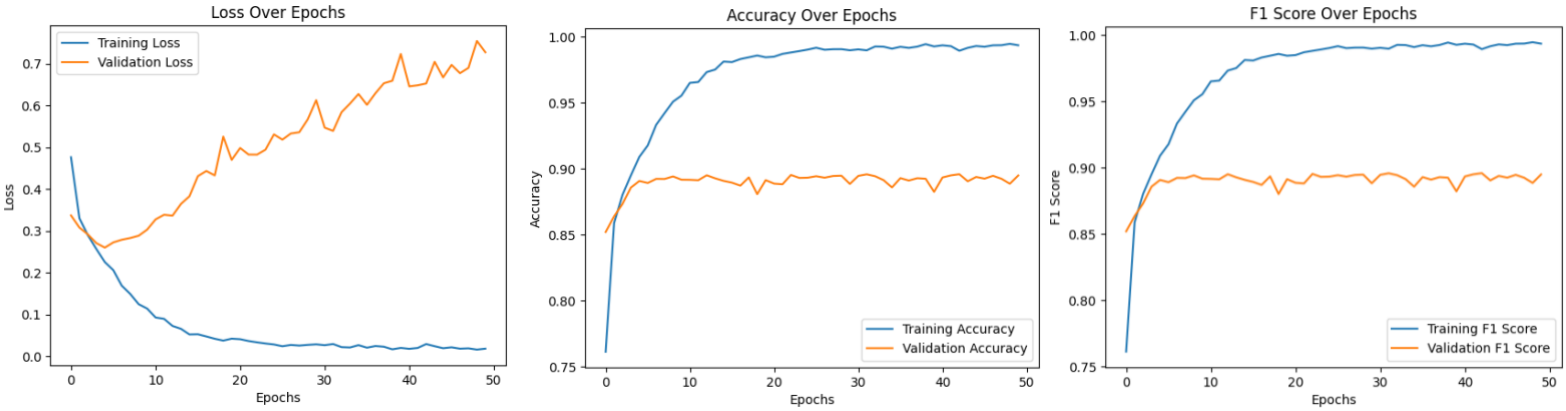
The best-performing parameters are identified based on the macro F1 score on the development set. The final results include the best hyperparameters and their corresponding performance metrics.

```
Testing parameters: {'filters': 300, 'n_values': [4], 'stacks': 1, 'dropout_prob': 0.3, 'learning_rate': 0.001}
Epoch [1/5], Average Loss: 0.5421
Epoch [2/5], Average Loss: 0.3829
Epoch [3/5], Average Loss: 0.3447
Epoch [4/5], Average Loss: 0.3249
Epoch [5/5], Average Loss: 0.3073
Validation Accuracy: 0.8717
Validation F1 Score: 0.8716
Testing parameters: {'filters': 300, 'n_values': [2, 3, 4], 'stacks': 3, 'dropout_prob': 0.4, 'learning_rate': 0.01}
Epoch [1/5], Average Loss: 0.9582
Epoch [2/5], Average Loss: 0.4137
Epoch [3/5], Average Loss: 0.3843
Epoch [4/5], Average Loss: 0.3888
Epoch [5/5], Average Loss: 0.3582
Validation Accuracy: 0.8653
Validation F1 Score: 0.8648
Testing parameters: {'filters': 300, 'n_values': [2, 3], 'stacks': 2, 'dropout_prob': 0.2, 'learning_rate': 0.001}
...
Validation Accuracy: 0.8932
Validation F1 Score: 0.8932
Best hyperparameters: {'filters': 300, 'n_values': [2, 3, 4], 'stacks': 3, 'dropout_prob': 0.3, 'learning_rate': 0.001}
Best development macro f1: 0.8932
```

Best Model

```
MultikernelCNN(
  (embedding_layer): Embedding(100002, 300)
  (conv_blocks): ModuleDict(
    (conv_2): ModuleList(
      (0-2): 3 x Conv1d(300, 300, kernel_size=(2,), stride=(1,), padding=same)
    )
    (conv_3): ModuleList(
      (0-2): 3 x Conv1d(300, 300, kernel_size=(3,), stride=(1,), padding=same)
    )
    (conv_4): ModuleList(
      (0-2): 3 x Conv1d(300, 300, kernel_size=(4,), stride=(1,), padding=same)
    )
  )
  (global_max_pool): AdaptiveMaxPool1d(output_size=1)
  (output): Linear(in_features=900, out_features=2, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Then the classifier is trained using the best hyperparameters identified during the tuning process. The model is trained over 50 epochs using the `Utils.train_and_validate` method, which monitors performance on both the training and the validation set after each epoch. The training process is visualized in three performance plots:



Finally, we evaluate the performance of our **CNN Classifier** (using the best model and the optimal number of epochs according to validation loss, as shown in the plot above) against four baseline models:

1. **BiLSTM RNN Classifier**

2. **MLP Classifier**

3. **Majority Classifier**: A dummy classifier from scikit-learn that always predicts the majority class observed in the training data. This serves as a simple baseline for comparison.

4. **Logistic Regression with SGD**: A logistic regression classifier trained using stochastic gradient descent (SGD). The hyperparameters for this classifier were tuned in a previous assignment and are directly reused here without modification.

Best Params: `{'sgd__alpha': 0.00041127525951368984, 'sgd__eta0': 0.08198575356848872, 'sgd__max_iter': 500, 'sgd__n_iter_no_change': 5, 'sgd__tol': 1e-05`

Results

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	SGDClassifier	Train	Macro Average	0.826204	0.825600	0.825519	0.901780				
1	SGDClassifier	Train	0					0.812164	0.847120	0.829274	0.903547
2	SGDClassifier	Train	1					0.840244	0.804080	0.821764	0.900012
3	DummyClassifier	Train	Macro Average	0.250000	0.500000	0.333333	0.750000				
4	DummyClassifier	Train	0					0.500000	1.000000	0.666667	0.750000
5	DummyClassifier	Train	1					0.000000	0.000000	0.000000	0.750000
6	MLPClassifier	Train	Macro Average	0.844560	0.844560	0.844560	0.919860				
7	MLPClassifier	Train	0					0.844230	0.845040	0.844635	0.922708
8	MLPClassifier	Train	1					0.844891	0.844080	0.844485	0.917012
9	BiLSTM_Attention	Train	Macro Average	0.919770	0.918680	0.918627	0.975135				
10	BiLSTM_Attention	Train	0					0.898379	0.944160	0.920701	0.975507
11	BiLSTM_Attention	Train	1					0.941162	0.893200	0.916554	0.974764
12	CNNClassifier	Train	Macro Average	0.948036	0.947000	0.946969	0.989445				
13	CNNClassifier	Train	0					0.969577	0.922960	0.945694	0.989544
14	CNNClassifier	Train	1					0.926494	0.971040	0.948244	0.989347

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	SGDClassifier	Dev	Macro Average	0.821151	0.819920	0.819747	0.902342				
1	SGDClassifier	Dev	0					0.801266	0.850880	0.825328	0.901449
2	SGDClassifier	Dev	1					0.841037	0.788960	0.814167	0.903235
3	DummyClassifier	Dev	Macro Average	0.250000	0.500000	0.333333	0.750000				
4	DummyClassifier	Dev	0					0.500000	1.000000	0.666667	0.750000
5	DummyClassifier	Dev	1					0.000000	0.000000	0.000000	0.750000
6	MLPClassifier	Dev	Macro Average	0.838596	0.838560	0.838556	0.921411				
7	MLPClassifier	Dev	0					0.835128	0.843680	0.839382	0.922462
8	MLPClassifier	Dev	1					0.842063	0.833440	0.837729	0.920359
9	BiLSTM_Attention	Dev	Macro Average	0.889189	0.887840	0.887743	0.957427				
10	BiLSTM_Attention	Dev	0					0.866274	0.917280	0.891048	0.958969
11	BiLSTM_Attention	Dev	1					0.912105	0.858400	0.884438	0.955884
12	CNNClassifier	Dev	Macro Average	0.891773	0.889840	0.889704	0.959903				
13	CNNClassifier	Dev	0					0.919291	0.854720	0.885830	0.961108
14	CNNClassifier	Dev	1					0.864255	0.924960	0.893578	0.958697

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	SGDClassifier	Test	Macro Average	0.819840	0.818160	0.817921	0.897820				
1	SGDClassifier	Test	0					0.796658	0.854400	0.824519	0.897594
2	SGDClassifier	Test	1					0.843022	0.781920	0.811322	0.898046
3	DummyClassifier	Test	Macro Average	0.250000	0.500000	0.333333	0.750000				
4	DummyClassifier	Test	0					0.500000	1.000000	0.666667	0.750000
5	DummyClassifier	Test	1					0.000000	0.000000	0.000000	0.750000
6	MLPClassifier	Test	Macro Average	0.840399	0.840240	0.840221	0.917299				
7	MLPClassifier	Test	0					0.833046	0.851040	0.841947	0.920119
8	MLPClassifier	Test	1					0.847751	0.829440	0.838496	0.914478
9	BiLSTM_Attention	Test	Macro Average	0.887194	0.886320	0.886256	0.956674				
10	BiLSTM_Attention	Test	0					0.868795	0.910080	0.888958	0.958312
11	BiLSTM_Attention	Test	1					0.905594	0.862560	0.883553	0.955036
12	CNNClassifier	Test	Macro Average	0.884733	0.882640	0.882480	0.957531				
13	CNNClassifier	Test	0					0.913111	0.845760	0.878146	0.959679
14	CNNClassifier	Test	1					0.856355	0.919520	0.886814	0.955383

2. Exercise 3 (POS Tagging using a CNN)

In this exercise, we developed a part-of-speech (POS) tagger for the **English EWT** dataset from the Universal Dependencies treebanks. The objective was to create a system that accurately assigns POS tags to words in sentences, using only the words, sentences, and POS tags available in the dataset. Additional annotations, such as syntactic dependencies, were not included in this task.

Below there are some statistics for the dataset:

```
Training Data Statistics:
  Number of Sentences: 12544
  Number of Words: 204609
  Average Sentence Length: 16.31
```

```
Development Data Statistics:
  Number of Sentences: 2001
  Number of Words: 25152
  Average Sentence Length: 12.57
```

```
Test Data Statistics:
  Number of Sentences: 2077
  Number of Words: 25096
  Average Sentence Length: 12.08
```

UtilsPOSTagging Class:

The UtilsPOSTagging class provides utility methods for preprocessing and feature engineering in part-of-speech (POS) tagging tasks.

- `parse_conllu(path)`:
  Parses .conllu files to extract sentences as lists of (word, POS tag) tuples. Each word is converted to lowercase, and sentences without valid word-POS pairs are excluded.
- `calculate_statistics(tagged_sentences, title)`:
  Computes and displays dataset statistics, including the number of sentences, total words, and average sentence length. This method provides a quick overview of the dataset's characteristics.
- `sliding_window(sentences_as_words, sentences_as_tags, window_size=3, embedding_dim=300)`:
  This method constructs feature vectors and labels for training a POS tagger using a Multi-Layer Perceptron (MLP). It uses a sliding window approach to incorporate contextual information around each word in a sentence. Each window consists of a target word and its neighboring words, ensuring that the model can learn patterns based on context.
  To handle words near the beginning or end of a sentence, the method pads the input with zero vectors for the embeddings and placeholders for the POS tags. This padding ensures that each word, including those at the boundaries, has a complete window of neighbors. The size of the padding depends on the sliding window size, with padding added symmetrically to both ends of the sentence.
  The method then iterates through each word in the padded sentence, extracting a window of embeddings centered on the current word. The embeddings from the window are concatenated into a single flattened vector, which serves as a feature for the model. The corresponding POS tag of the center word is extracted as the label. By iterating through all the words in the dataset, the method generates a set of features and their associated labels, ready for training the MLP.
- `get_word_embedding(word, model, embedding_dim=300):`
  Retrieves the embedding vector for a given word from a pre-trained model. If the word is not found in the model's vocabulary, it returns a zero vector of the specified embedding dimension.
- `tags(sentences):`
  Extracts POS tags from a list of sentences. Each sentence is represented as a list of tags.

- `words(sentences)`:
  Extracts words from a list of sentences. Each sentence is represented as a list of words.
- `map_tags_to_numbers(tags, tag_to_int)`:
  Converts a list of tags into their corresponding numeric representations based on a provided mapping (tag_to_int). This is essential for preparing labels for machine learning models.

The **MultikernelCNN** class is similar to that implemented in Exercise 2, appropriately configured for a multiclass token classification problem. The **Utils_Padded** class functions similarly to **UtilsMLP,** again appropriately configured for a token classification problem. Lastly, the **Adapter** class provides a training and evaluation framework for both the CNN and the BiLSTM RNN model. It integrates the training process, prediction of class labels, and prediction of class probabilities.

MostFrequentTagBaseline Class: It is a baseline model for POS tagging that predicts tags based on their most frequent occurrence per feature or overall. It calculates tag probabilities during training and predicts either the most frequent tag for a feature or the overall most frequent tag when the feature is unseen.

- `fit`: Learns the most frequent tag per feature and overall tag probabilities.
- `predict`: Predicts tags using the learned frequencies.
- `predict_proba`: Outputs tag probabilities.

To begin with, the .conllu files are loaded and parsed using the `pyconll` library to extract useful linguistic annotations (e.g., word forms and POS tags). More specifically, the parsing function (`UtilsPOSTagging.parse_conllu`) is called for each dataset (training, development, and test).

```
Randomly selected examples
1. Sentence: ([('but', 'CCONJ'), ('comfortable', 'ADJ'), ('among', 'ADP'), ('the', 'DET'), ('thugs', 'NOUN'), (',', 'P
2. Sentence: ([('thanks', 'NOUN')],)
3. Sentence: ([('the', 'DET'), ('next', 'ADJ'), ('time', 'NOUN'), ('you', 'PRON'), ('feel', 'VERB'), ('like', 'SCONJ')
4. Sentence: ([('the', 'DET'), ('bear', 'NOUN'), ('dwarfed', 'VERB'), ('the', 'DET'), ('6', 'NUM'), ('-', 'PUNCT'), ('
5. Sentence: ([('you', 'PRON'), ('want', 'VERB'), ('to', 'PART'), ('merge', 'VERB'), ('your', 'PRON'), ('mind', 'NOUN'
```

Preparing Features and Labels (Tags)

We prepare data for training a Bidirectional Long Short-Term Memory (BILSTM) model to predict Part-of-Speech (POS) tags using sliding window word embeddings.

We use the pre-trained `word2vec-google-news-300` model from Gensim's API to obtain 300-dimensional word embeddings for the words in our dataset.

1. **Vocabulary Creation**

Word Vocabulary: Extract all unique words from the training dataset to create a `word2idx` mapping. Special tokens include: UNK (unknown words) and PAD (padding).

Tag Vocabulary: Extract all unique POS tags from the training tag sequences to create a `tag2idx` mapping.

Reverse Tag Mapping: A reverse mapping (`idx2tag`) is created to decode tag indices back into their corresponding tags.

2. **Encoding Sequences**

In this step, word and tag sequences are converted into numerical indices using the `word2id` and `tag2idx` mappings. For words not present in the vocabulary, the UNK token index is used. This step ensures that the text data is compatible with the numerical computations of the model.

3. **Padding Sequences**

Since sentences in the dataset vary in length, padding is applied to make all sequences equal in length. The PAD token is used for both words and tags. A fixed maximum length (`max_len`) is defined to standardize the sequence lengths. Sequences longer than `max_len` are truncated, while shorter ones are padded.

4. **One-Hot Encoding of Tags**

The padded tag sequences are converted to a categorical format (one-hot encoded) to match the output requirements of the model. Each tag is represented as a vector with a length equal to the total number of unique tags.

Experiments:

The primary objective of our experiments is to assess and compare the performance of our CNN POS Tagger against three baselines:

1. Best BiLSTM Model: After performing hyperparameter tuning on the BiLSTM model with attention, we use the configuration that achieved the highest macro F1 score on the validation set. This model utilizes pre-trained word embeddings (e.g., Word2Vec) and incorporates an attention mechanism to improve context understanding.

2. Best MLP Model: After conducting hyperparameter tuning on the MLP model, we use the configuration that achieved the best performance on the validation set. This model utilizes sliding window word embeddings.

2. Baseline Classifier: As a reference point, we implement a baseline classifier that operates as follows:

- For each word in the test set, it assigns the **most frequent tag** that the word had in the training data.
- For words not encountered during training (unseen words), the classifier assigns the **most frequent tag overall** (across all words) from the training data.

First, we configure the model with initial, arbitrarily chosen hyperparameters. This initial testing allows us to observe baseline performance and helps us identify potential adjustments for further tuning. Subsequent steps involve tuning these hyperparameters for optimal performance.

Hyperparameter Tuning for CNN POS Tagger:

Hyperparameters tuned

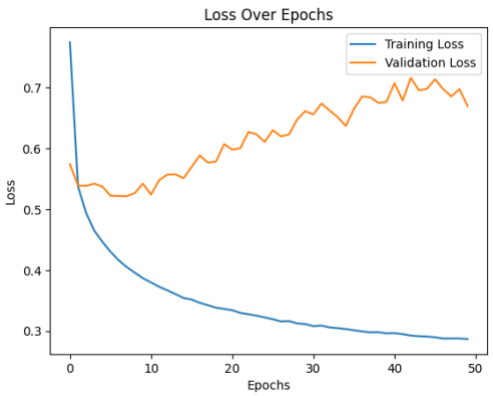
- **Number of Filters per Convolution (`filters`):** Represents the number of filters used in each convolutional layer.
- **n-gram sizes (`n_values`)**: Defines the kernel sizes used for convolutions, representing n-grams to capture.
- **Number of Stacked Layers per n-gram (`stacks`)**: Specifies the number of stacked convolutional layers for each n-gram size.
- **Dropout Probability (`dropout_prob`)**: Specifies the dropout probability.
- **Learning Rate (`learning_rate`)**: Controls the step size for weight updates during optimization.

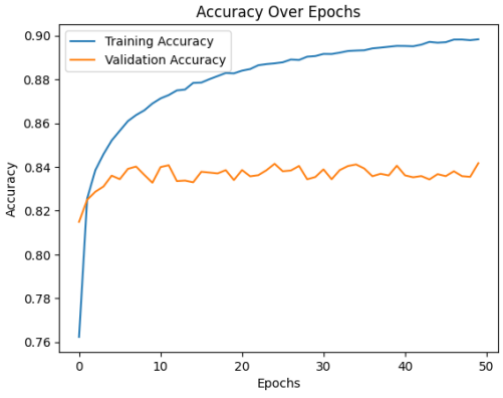
Tuning Procedure: A random sampling of hyperparameter combinations is performed over 50 iterations. Each combination is tested to evaluate its performance on the development dataset. The macro F1 score on the development set is used as the primary metric to assess model performance. The combination of hyperparameters that yields the highest macro F1 score on the development dataset is selected as the best configuration.

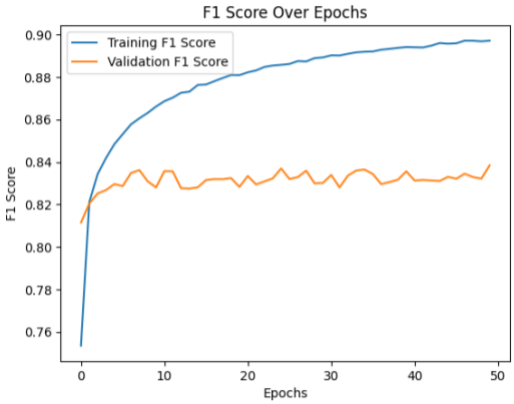
Best Model

```
MultikernelCNN(
  (embedding_layer): Embedding(16656, 300)
  (conv_blocks): ModuleDict(
    (conv_2): ModuleList(
      (0-1): 2 x Conv1d(300, 300, kernel_size=(2,), stride=(1,), padding=same)
    )
  )
  (output): Conv1d(300, 18, kernel_size=(1,), stride=(1,))
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Then the CNN POS Tagger is trained using the best hyperparameters identified during the tuning process. The model is trained over 50 epochs. The training process is visualized in three performance plots:







Lastly, the performance of our **CNN POS Tagger** (using the best model and the optimal number of epochs according to validation loss, as shown in the plot above) is evaluated against the baselines.

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	MLPClassifier	Train	Macro Average	0.905401	0.816698	0.839655	0.901826	nan	nan	nan	nan
1	MLPClassifier	Train	0	nan	nan	nan	nan	0.934697	0.952984	0.943752	0.985258
2	MLPClassifier	Train	1	nan	nan	nan	nan	0.892222	0.907210	0.899653	0.963371
3	MLPClassifier	Train	2	nan	nan	nan	nan	0.970544	0.892359	0.929811	0.982059
4	MLPClassifier	Train	3	nan	nan	nan	nan	0.980784	0.983616	0.982198	0.997863
5	MLPClassifier	Train	4	nan	nan	nan	nan	0.840064	0.395095	0.537429	0.730173
6	MLPClassifier	Train	5	nan	nan	nan	nan	0.951231	0.962145	0.956657	0.991726
7	MLPClassifier	Train	6	nan	nan	nan	nan	0.926619	0.926619	0.926619	0.956934
8	MLPClassifier	Train	7	nan	nan	nan	nan	0.921732	0.963343	0.942079	0.983806
9	MLPClassifier	Train	8	nan	nan	nan	nan	0.867235	0.690090	0.768587	0.857841
10	MLPClassifier	Train	9	nan	nan	nan	nan	0.884342	0.909882	0.896930	0.962906
11	MLPClassifier	Train	10	nan	nan	nan	nan	0.992448	0.992129	0.992289	0.999090
12	MLPClassifier	Train	11	nan	nan	nan	nan	0.922914	0.692661	0.791380	0.897359
13	MLPClassifier	Train	12	nan	nan	nan	nan	0.718364	0.921427	0.807322	0.897149
14	MLPClassifier	Train	13	nan	nan	nan	nan	0.850886	0.904762	0.876997	0.949351
15	MLPClassifier	Train	14	nan	nan	nan	nan	0.879925	0.649584	0.747410	0.749532
16	MLPClassifier	Train	15	nan	nan	nan	nan	0.974084	0.949478	0.961624	0.992296
17	MLPClassifier	Train	16	nan	nan	nan	nan	0.883721	0.190476	0.313402	0.434332
18	MostFrequentTagBaseline	Train	Macro Average	0.964640	0.938694	0.950630	0.981475	nan	nan	nan	nan
19	MostFrequentTagBaseline	Train	0	nan	nan	nan	nan	0.994069	0.991431	0.992748	0.999617
20	MostFrequentTagBaseline	Train	1	nan	nan	nan	nan	0.982353	0.972828	0.977568	0.998242
21	MostFrequentTagBaseline	Train	2	nan	nan	nan	nan	0.992362	0.988831	0.990593	0.999578
22	MostFrequentTagBaseline	Train	3	nan	nan	nan	nan	0.997335	0.992744	0.995034	0.999873
23	MostFrequentTagBaseline	Train	4	nan	nan	nan	nan	0.929619	0.890833	0.909813	0.977373
24	MostFrequentTagBaseline	Train	5	nan	nan	nan	nan	0.993767	0.987975	0.990862	0.999434
25	MostFrequentTagBaseline	Train	6	nan	nan	nan	nan	0.976676	0.964029	0.970311	0.990681
26	MostFrequentTagBaseline	Train	7	nan	nan	nan	nan	0.990079	0.984865	0.987465	0.999235
27	MostFrequentTagBaseline	Train	8	nan	nan	nan	nan	0.908052	0.789678	0.844738	0.930614
28	MostFrequentTagBaseline	Train	9	nan	nan	nan	nan	0.989317	0.982777	0.986036	0.999047
29	MostFrequentTagBaseline	Train	10	nan	nan	nan	nan	0.997861	0.999197	0.998529	0.999988
30	MostFrequentTagBaseline	Train	11	nan	nan	nan	nan	0.960959	0.889523	0.923862	0.982439
31	MostFrequentTagBaseline	Train	12	nan	nan	nan	nan	0.868081	0.962409	0.912815	0.984034
32	MostFrequentTagBaseline	Train	13	nan	nan	nan	nan	0.987569	0.976975	0.982244	0.999082
33	MostFrequentTagBaseline	Train	14	nan	nan	nan	nan	0.931464	0.828255	0.876833	0.924258
34	MostFrequentTagBaseline	Train	15	nan	nan	nan	nan	0.994553	0.993541	0.994047	0.999866
35	MostFrequentTagBaseline	Train	16	nan	nan	nan	nan	0.904762	0.761905	0.827211	0.901711
36	BiLSTM	Train	Macro Average	0.900932	0.785330	0.809152	0.870750	nan	nan	nan	nan
37	BiLSTM	Train	1	nan	nan	nan	nan	0.945017	0.791367	0.861394	0.906077
38	BiLSTM	Train	2	nan	nan	nan	nan	0.981579	0.516620	0.676951	0.630626
39	BiLSTM	Train	3	nan	nan	nan	nan	0.949864	0.733647	0.827871	0.908593
40	BiLSTM	Train	4	nan	nan	nan	nan	0.989883	0.984848	0.987359	0.996925
41	BiLSTM	Train	5	nan	nan	nan	nan	0.862458	0.741459	0.797394	0.870128
42	BiLSTM	Train	6	nan	nan	nan	nan	0.911341	0.973741	0.941508	0.988195
43	BiLSTM	Train	7	nan	nan	nan	nan	0.939106	0.938944	0.939025	0.976979
44	BiLSTM	Train	8	nan	nan	nan	nan	0.919946	0.925457	0.922693	0.959612
45	BiLSTM	Train	9	nan	nan	nan	nan	0.851434	0.736542	0.789832	0.874298
46	BiLSTM	Train	10	nan	nan	nan	nan	0.967050	0.955712	0.961348	0.986053
47	BiLSTM	Train	11	nan	nan	nan	nan	0.874862	0.896268	0.885436	0.951354
48	BiLSTM	Train	12	nan	nan	nan	nan	0.888889	0.020050	0.039216	0.280958
49	BiLSTM	Train	13	nan	nan	nan	nan	0.773956	0.423957	0.547826	0.673774
50	BiLSTM	Train	14	nan	nan	nan	nan	0.983242	0.988765	0.985995	0.995550
51	BiLSTM	Train	15	nan	nan	nan	nan	0.889615	0.928833	0.908801	0.955269
52	BiLSTM	Train	16	nan	nan	nan	nan	0.752607	0.908544	0.823256	0.927304
53	BiLSTM	Train	17	nan	nan	nan	nan	0.835001	0.885853	0.859676	0.921060

54	CNN	Train	Macro Average	0.893926	0.790439	0.812469	0.880827	nan	nan	nan	nan
55	CNN	Train	1	nan	nan	nan	nan	0.937500	0.841727	0.887036	0.935834
56	CNN	Train	2	nan	nan	nan	nan	0.947826	0.603878	0.737733	0.746734
57	CNN	Train	3	nan	nan	nan	nan	0.849575	0.889587	0.869121	0.942036
58	CNN	Train	4	nan	nan	nan	nan	0.989178	0.983670	0.986416	0.998623
59	CNN	Train	5	nan	nan	nan	nan	0.844539	0.672644	0.748854	0.821175
60	CNN	Train	6	nan	nan	nan	nan	0.929050	0.944782	0.936850	0.986511
61	CNN	Train	7	nan	nan	nan	nan	0.926166	0.936239	0.931176	0.976419
62	CNN	Train	8	nan	nan	nan	nan	0.950923	0.937438	0.944133	0.983535
63	CNN	Train	9	nan	nan	nan	nan	0.910722	0.672084	0.773414	0.867123
64	CNN	Train	10	nan	nan	nan	nan	0.950714	0.943058	0.946870	0.985710
65	CNN	Train	11	nan	nan	nan	nan	0.844322	0.863457	0.853782	0.940738
66	CNN	Train	12	nan	nan	nan	nan	0.925000	0.092732	0.168565	0.348349
67	CNN	Train	13	nan	nan	nan	nan	0.799935	0.368925	0.504964	0.659373
68	CNN	Train	14	nan	nan	nan	nan	0.970706	0.966919	0.968809	0.992617
69	CNN	Train	15	nan	nan	nan	nan	0.927906	0.924879	0.926390	0.978415
70	CNN	Train	16	nan	nan	nan	nan	0.692368	0.900025	0.782657	0.889368
71	CNN	Train	17	nan	nan	nan	nan	0.800311	0.895424	0.845200	0.921499

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	MLPClassifier	Dev	Macro Average	0.800241	0.750421	0.765997	0.830905	nan	nan	nan	nan
1	MLPClassifier	Dev	0	nan	nan	nan	nan	0.916225	0.922584	0.919393	0.966605
2	MLPClassifier	Dev	1	nan	nan	nan	nan	0.849048	0.874877	0.861769	0.932835
3	MLPClassifier	Dev	2	nan	nan	nan	nan	0.944598	0.835784	0.886866	0.955609
4	MLPClassifier	Dev	3	nan	nan	nan	nan	0.968987	0.977026	0.972990	0.992863
5	MLPClassifier	Dev	4	nan	nan	nan	nan	0.772989	0.345315	0.477374	0.647908
6	MLPClassifier	Dev	5	nan	nan	nan	nan	0.913244	0.936316	0.924636	0.981065
7	MLPClassifier	Dev	6	nan	nan	nan	nan	0.813084	0.756522	0.783784	0.847112
8	MLPClassifier	Dev	7	nan	nan	nan	nan	0.864590	0.929250	0.895755	0.946402
9	MLPClassifier	Dev	8	nan	nan	nan	nan	0.724590	0.577023	0.642442	0.723311
10	MLPClassifier	Dev	9	nan	nan	nan	nan	0.796760	0.836167	0.815988	0.904094
11	MLPClassifier	Dev	10	nan	nan	nan	nan	0.984719	0.984719	0.984719	0.996299
12	MLPClassifier	Dev	11	nan	nan	nan	nan	0.881204	0.580697	0.700065	0.812476
13	MLPClassifier	Dev	12	nan	nan	nan	nan	0.689294	0.898211	0.780006	0.861911
14	MLPClassifier	Dev	13	nan	nan	nan	nan	0.793349	0.841310	0.816626	0.906750
15	MLPClassifier	Dev	14	nan	nan	nan	nan	0.750000	0.542169	0.629371	0.603961
16	MLPClassifier	Dev	15	nan	nan	nan	nan	0.941421	0.919188	0.930172	0.973182
17	MLPClassifier	Dev	16	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.073009
18	MostFrequentTagBaseline	Dev	Macro Average	0.787014	0.332194	0.419534	0.528552	nan	nan	nan	nan
19	MostFrequentTagBaseline	Dev	0	nan	nan	nan	nan	0.955665	0.207154	0.340500	0.607536
20	MostFrequentTagBaseline	Dev	1	nan	nan	nan	nan	0.835391	0.199215	0.321712	0.499591
21	MostFrequentTagBaseline	Dev	2	nan	nan	nan	nan	0.935933	0.274510	0.424510	0.612346
22	MostFrequentTagBaseline	Dev	3	nan	nan	nan	nan	0.984177	0.396937	0.565712	0.700174
23	MostFrequentTagBaseline	Dev	4	nan	nan	nan	nan	0.670051	0.169448	0.270492	0.251057
24	MostFrequentTagBaseline	Dev	5	nan	nan	nan	nan	0.961872	0.292105	0.448123	0.640033
25	MostFrequentTagBaseline	Dev	6	nan	nan	nan	nan	0.924528	0.426087	0.583333	0.601647
26	MostFrequentTagBaseline	Dev	7	nan	nan	nan	nan	0.236573	0.973647	0.380656	0.637029
27	MostFrequentTagBaseline	Dev	8	nan	nan	nan	nan	0.613333	0.240209	0.345216	0.329990
28	MostFrequentTagBaseline	Dev	9	nan	nan	nan	nan	0.955224	0.296754	0.452830	0.578153
29	MostFrequentTagBaseline	Dev	10	nan	nan	nan	nan	0.989418	0.504270	0.668056	0.779282
30	MostFrequentTagBaseline	Dev	11	nan	nan	nan	nan	0.805430	0.190885	0.308626	0.423616
31	MostFrequentTagBaseline	Dev	12	nan	nan	nan	nan	0.758146	0.537236	0.628854	0.694259
32	MostFrequentTagBaseline	Dev	13	nan	nan	nan	nan	0.944000	0.297229	0.452107	0.561892
33	MostFrequentTagBaseline	Dev	14	nan	nan	nan	nan	0.722222	0.313253	0.436975	0.353550
34	MostFrequentTagBaseline	Dev	15	nan	nan	nan	nan	0.969623	0.294465	0.451741	0.677901
35	MostFrequentTagBaseline	Dev	16	nan	nan	nan	nan	0.117647	0.033898	0.052632	0.037321

36	BiLSTM	Dev	Macro Average	0.816575	0.736860	0.762174	0.807666	nan	nan	nan	nan
37	BiLSTM	Dev	1	nan	nan	nan	nan	0.831579	0.686957	0.752381	0.738400
38	BiLSTM	Dev	2	nan	nan	nan	nan	0.972973	0.433735	0.600000	0.507905
39	BiLSTM	Dev	3	nan	nan	nan	nan	0.944251	0.682620	0.792398	0.852950
40	BiLSTM	Dev	4	nan	nan	nan	nan	0.986030	0.983371	0.984698	0.994347
41	BiLSTM	Dev	5	nan	nan	nan	nan	0.838710	0.543081	0.659271	0.672456
42	BiLSTM	Dev	6	nan	nan	nan	nan	0.905378	0.956842	0.930399	0.982552
43	BiLSTM	Dev	7	nan	nan	nan	nan	0.888833	0.842830	0.865221	0.939344
44	BiLSTM	Dev	8	nan	nan	nan	nan	0.922419	0.863321	0.891892	0.929393
45	BiLSTM	Dev	9	nan	nan	nan	nan	0.590461	0.803217	0.680600	0.743786
46	BiLSTM	Dev	10	nan	nan	nan	nan	0.941292	0.881550	0.910442	0.935497
47	BiLSTM	Dev	11	nan	nan	nan	nan	0.858937	0.872424	0.865628	0.931436
48	BiLSTM	Dev	12	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.093520
49	BiLSTM	Dev	13	nan	nan	nan	nan	0.773869	0.395379	0.523364	0.672974
50	BiLSTM	Dev	14	nan	nan	nan	nan	0.965474	0.981493	0.973418	0.988985
51	BiLSTM	Dev	15	nan	nan	nan	nan	0.877850	0.880719	0.879282	0.931281
52	BiLSTM	Dev	16	nan	nan	nan	nan	0.792915	0.895285	0.840996	0.934002
53	BiLSTM	Dev	17	nan	nan	nan	nan	0.790801	0.823802	0.806964	0.881492
54	CNN	Dev	Macro Average	0.803844	0.712121	0.740386	0.799099	nan	nan	nan	nan
55	CNN	Dev	1	nan	nan	nan	nan	0.865169	0.669565	0.754902	0.752122
56	CNN	Dev	2	nan	nan	nan	nan	0.953488	0.493976	0.650794	0.562972
57	CNN	Dev	3	nan	nan	nan	nan	0.790524	0.798489	0.794486	0.881205
58	CNN	Dev	4	nan	nan	nan	nan	0.980561	0.974831	0.977688	0.996446
59	CNN	Dev	5	nan	nan	nan	nan	0.809302	0.454308	0.581940	0.553260
60	CNN	Dev	6	nan	nan	nan	nan	0.886069	0.900526	0.893239	0.968247
61	CNN	Dev	7	nan	nan	nan	nan	0.721236	0.925689	0.810771	0.911855
62	CNN	Dev	8	nan	nan	nan	nan	0.928116	0.854778	0.889939	0.946895
63	CNN	Dev	9	nan	nan	nan	nan	0.815647	0.486327	0.609338	0.758936
64	CNN	Dev	10	nan	nan	nan	nan	0.922345	0.859041	0.889568	0.944248
65	CNN	Dev	11	nan	nan	nan	nan	0.825158	0.833660	0.829387	0.917873
66	CNN	Dev	12	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.059782
67	CNN	Dev	13	nan	nan	nan	nan	0.811728	0.337612	0.476881	0.622755
68	CNN	Dev	14	nan	nan	nan	nan	0.964493	0.953414	0.958922	0.983734
69	CNN	Dev	15	nan	nan	nan	nan	0.904393	0.857843	0.880503	0.950234
70	CNN	Dev	16	nan	nan	nan	nan	0.744173	0.893008	0.811826	0.917466
71	CNN	Dev	17	nan	nan	nan	nan	0.742938	0.812983	0.776384	0.856660

	Classifier	Subset	Class	Macro Precision	Macro Recall	Macro F1	Macro PR-AUC	Precision	Recall	F1	PR-AUC
0	MLPClassifier	Test	Macro Average	0.823153	0.762456	0.782018	0.837326	nan	nan	nan	nan
1	MLPClassifier	Test	0	nan	nan	nan	nan	0.908493	0.924192	0.916275	0.961572
2	MLPClassifier	Test	1	nan	nan	nan	nan	0.845232	0.868966	0.856935	0.934522
3	MLPClassifier	Test	2	nan	nan	nan	nan	0.958608	0.861369	0.907391	0.961292
4	MLPClassifier	Test	3	nan	nan	nan	nan	0.975420	0.977317	0.976368	0.993322
5	MLPClassifier	Test	4	nan	nan	nan	nan	0.766272	0.351902	0.482309	0.653094
6	MLPClassifier	Test	5	nan	nan	nan	nan	0.920188	0.930380	0.925256	0.978324
7	MLPClassifier	Test	6	nan	nan	nan	nan	0.889831	0.867769	0.878661	0.910088
8	MLPClassifier	Test	7	nan	nan	nan	nan	0.873890	0.931118	0.901597	0.950023
9	MLPClassifier	Test	8	nan	nan	nan	nan	0.762712	0.498155	0.602679	0.689716
10	MLPClassifier	Test	9	nan	nan	nan	nan	0.835098	0.850539	0.842748	0.926083
11	MLPClassifier	Test	10	nan	nan	nan	nan	0.983871	0.985688	0.984779	0.996086
12	MLPClassifier	Test	11	nan	nan	nan	nan	0.874727	0.578516	0.696434	0.806849
13	MLPClassifier	Test	12	nan	nan	nan	nan	0.658962	0.902455	0.761723	0.843524
14	MLPClassifier	Test	13	nan	nan	nan	nan	0.841146	0.841146	0.841146	0.912817
15	MLPClassifier	Test	14	nan	nan	nan	nan	0.947368	0.660550	0.778378	0.716952
16	MLPClassifier	Test	15	nan	nan	nan	nan	0.951784	0.931696	0.941633	0.982084
17	MLPClassifier	Test	16	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.018200

18	MostFrequentTagBaseline	Test	Macro Average	0.803940	0.342006	0.430657	0.541903	nan	nan	nan	nan
19	MostFrequentTagBaseline	Test	0	nan	nan	nan	nan	0.950617	0.214604	0.350159	0.609094
20	MostFrequentTagBaseline	Test	1	nan	nan	nan	nan	0.905039	0.230049	0.366850	0.541895
21	MostFrequentTagBaseline	Test	2	nan	nan	nan	nan	0.961194	0.272189	0.424242	0.622439
22	MostFrequentTagBaseline	Test	3	nan	nan	nan	nan	0.991883	0.395982	0.566003	0.705831
23	MostFrequentTagBaseline	Test	4	nan	nan	nan	nan	0.593458	0.172554	0.267368	0.233083
24	MostFrequentTagBaseline	Test	5	nan	nan	nan	nan	0.967797	0.301160	0.459372	0.640369
25	MostFrequentTagBaseline	Test	6	nan	nan	nan	nan	0.969697	0.528926	0.684492	0.686630
26	MostFrequentTagBaseline	Test	7	nan	nan	nan	nan	0.237366	0.977444	0.381972	0.646695
27	MostFrequentTagBaseline	Test	8	nan	nan	nan	nan	0.603175	0.210332	0.311902	0.397799
28	MostFrequentTagBaseline	Test	9	nan	nan	nan	nan	0.954955	0.326656	0.486797	0.604698
29	MostFrequentTagBaseline	Test	10	nan	nan	nan	nan	0.991877	0.507387	0.671350	0.781896
30	MostFrequentTagBaseline	Test	11	nan	nan	nan	nan	0.800745	0.207129	0.329124	0.442606
31	MostFrequentTagBaseline	Test	12	nan	nan	nan	nan	0.710448	0.538114	0.612387	0.665463
32	MostFrequentTagBaseline	Test	13	nan	nan	nan	nan	0.955752	0.281250	0.434608	0.560531
33	MostFrequentTagBaseline	Test	14	nan	nan	nan	nan	0.878049	0.330275	0.480000	0.393412
34	MostFrequentTagBaseline	Test	15	nan	nan	nan	nan	0.944920	0.296239	0.451066	0.662463
35	MostFrequentTagBaseline	Test	16	nan	nan	nan	nan	0.250000	0.023810	0.043478	0.017446
36	BiLSTM	Test	Macro Average	0.821795	0.741770	0.767832	0.814650	nan	nan	nan	nan
37	BiLSTM	Test	1	nan	nan	nan	nan	0.937500	0.743802	0.829493	0.831538
38	BiLSTM	Test	2	nan	nan	nan	nan	0.952381	0.550459	0.697674	0.631982
39	BiLSTM	Test	3	nan	nan	nan	nan	0.944853	0.669271	0.783537	0.861503
40	BiLSTM	Test	4	nan	nan	nan	nan	0.982399	0.979224	0.980809	0.995768
41	BiLSTM	Test	5	nan	nan	nan	nan	0.778571	0.402214	0.530414	0.586488
42	BiLSTM	Test	6	nan	nan	nan	nan	0.920489	0.952532	0.936236	0.980017
43	BiLSTM	Test	7	nan	nan	nan	nan	0.896856	0.837254	0.866031	0.939166
44	BiLSTM	Test	8	nan	nan	nan	nan	0.910967	0.861204	0.885387	0.928483
45	BiLSTM	Test	9	nan	nan	nan	nan	0.587987	0.806358	0.680073	0.745808
46	BiLSTM	Test	10	nan	nan	nan	nan	0.946210	0.891021	0.917787	0.937996
47	BiLSTM	Test	11	nan	nan	nan	nan	0.857212	0.866502	0.861832	0.931409
48	BiLSTM	Test	12	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.040282
49	BiLSTM	Test	13	nan	nan	nan	nan	0.781081	0.392663	0.522604	0.666328
50	BiLSTM	Test	14	nan	nan	nan	nan	0.973514	0.976669	0.975089	0.991789
51	BiLSTM	Test	15	nan	nan	nan	nan	0.876634	0.907016	0.891566	0.937112
52	BiLSTM	Test	16	nan	nan	nan	nan	0.792501	0.907946	0.846304	0.930859
53	BiLSTM	Test	17	nan	nan	nan	nan	0.831361	0.865948	0.848302	0.912520
54	CNN	Test	Macro Average	0.810897	0.725209	0.751655	0.805913	nan	nan	nan	nan
55	CNN	Test	1	nan	nan	nan	nan	0.927835	0.743802	0.825688	0.844540
56	CNN	Test	2	nan	nan	nan	nan	0.943662	0.614679	0.744444	0.649008
57	CNN	Test	3	nan	nan	nan	nan	0.828496	0.817708	0.823067	0.901333
58	CNN	Test	4	nan	nan	nan	nan	0.982480	0.983841	0.983160	0.995849
59	CNN	Test	5	nan	nan	nan	nan	0.774704	0.361624	0.493082	0.451400
60	CNN	Test	6	nan	nan	nan	nan	0.898835	0.895042	0.896934	0.968251
61	CNN	Test	7	nan	nan	nan	nan	0.714339	0.927965	0.807258	0.915453
62	CNN	Test	8	nan	nan	nan	nan	0.930597	0.859532	0.893654	0.944145
63	CNN	Test	9	nan	nan	nan	nan	0.819543	0.500963	0.621824	0.780568
64	CNN	Test	10	nan	nan	nan	nan	0.928282	0.879125	0.903035	0.954552
65	CNN	Test	11	nan	nan	nan	nan	0.829161	0.832020	0.830588	0.919978
66	CNN	Test	12	nan	nan	nan	nan	0.000000	0.000000	0.000000	0.010939
67	CNN	Test	13	nan	nan	nan	nan	0.787582	0.327446	0.462572	0.614324
68	CNN	Test	14	nan	nan	nan	nan	0.967742	0.952690	0.960157	0.986992
69	CNN	Test	15	nan	nan	nan	nan	0.920385	0.889265	0.904557	0.960346
70	CNN	Test	16	nan	nan	nan	nan	0.735983	0.903101	0.811022	0.914308
71	CNN	Test	17	nan	nan	nan	nan	0.795620	0.839753	0.817091	0.888545