A REPORT

ON

CS F429 Natural Language Processing: Assignment 1

by

Name of the student	ID number
Pavas Garg	2021A7PS2587H
Atharva Dashora	2021A7PS0127H

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
(Hyderabad Campus)

September, 2024

CONTENTS

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

2 Concepts **2**
2.1 Word Embeddings 2
2.2 BiLSTM Model 2
2.2.1 BiLSTM CRF 3
2.3 Word2Vec 4
2.3.1 Skip Gram 5
2.3.2 CBOW 5

3 Apparatus **6**

4 Dataset **7**

5 Methodology for OpenIE **8**
5.1 Pre-Processing Data 8
5.2 BiLSTM CRF Model 10
5.2.1 LSTM Layer 10
5.2.2 Dropout Layer 10
5.2.3 Batch Normalization Layer 10
5.2.4 Fully Connected Layers 10
5.2.5 Conditional Random Field (CRF) 10
5.3 Training the Model 10
5.4 Evaluation on Validation Data 12
5.5 Extracting Relations 12
5.5.1 Enforce a Strict Pattern on the Extractions 13
5.5.2 Assign Confidence Value to Extractions 15
5.6 Error Analysis 16
5.6.1 Examples 17

6 Methodology for Word2Vec **17**
6.1 Data Cleaning and Tokenization 17
6.2 Filtering Sentences and Words 18
6.3 Output 18
6.4 Skip-Gram Model 18
6.4.1 Model Architecture 18
6.4.2 Training Process 19
6.4.3 Prediction Mechanism 19
6.4.4 Output and Embeddings 20
6.5 Skip-Gram Model with Negative Sampling 20
6.5.1 Model Architecture 20
6.5.2 Training Process with Negative Sampling 20
6.5.3 Choosing the Number of Negative Samples 21

6.5.4 Prediction Mechanism 21
6.6 Continuous Bag of Words (CBOW) Model 21
6.6.1 Model Architecture 22
6.6.2 Training Process 22
6.6.3 Prediction Mechanism 22
6.6.4 Advantages of CBOW 23
6.7 Training Methodology for Skip-Gram Model 23
6.7.1 Data Preparation 23
6.7.2 Batch Generation 23
6.7.3 Training Loop 23
6.8 Mean Reciprocal Rank (MRR) Metric 24
6.9 Model Performance and Training Time Comparison 24
6.9.1 Mean Reciprocal Rank (MRR) 24
6.9.2 Training Time 25
6.9.3 Analysis and Findings 25
6.9.4 Conclusions 26
6.10 CBOW Results and Explanation 27
6.10.1 Mean Reciprocal Rank (MRR) 27
6.10.2 Training Time 28
6.10.3 Reasoning Behind the Time Differences 29
6.10.4 Conclusions 29
6.10.5 Average Time and MRR 29

7 References 31

8 Glossary 32

List of Tables

Table 1 Incorrect Extractions 16
Table 2 Missed Extractions 17
Table 3 Mean Reciprocal Rank (MRR) for SkipGram with Softmax and Negative Sampling 25
Table 4 Training Time (in seconds) for SkipGram with Softmax and Negative Sampling 25
Table 5 Glossary 32

List of Figures

Figure 1	Knowledge Graph	1
Figure 2	BERT Model	2
Figure 3	BiLSTM Architecture	3
Figure 4	CRF	4
Figure 5	Skip Gram	5
Figure 6	CBOW	6
Figure 7	BERT-BiLSTM-CRF Model	11
Figure 8	Training Loss	13
Figure 9	MRR Comparison	26
Figure 10	Training Time	27

List of Algorithms

1 LOAD_DATASET 8
2 Get BERT Embeddings 9
3 Training BiLSTM-CRF Model 12
4 Relationship Extraction Algorithm (Strict Pattern) 14
5 Extract Confidence Features 15
6 Confidence Score Calculation for Relationship Extraction 16
7 Training Algorithm for Skip-Gram Model 24

1 Introduction

Open Information Extraction (OpenIE) is a crucial task in natural language processing (NLP) that involves extracting tuples of structured information from unstructured text. These tuples are typically in the form ⟨subject, relation, object, time, location⟩, with the goal of capturing the key relationships and entities within a sentence. OpenIE plays a fundamental role in various downstream applications, such as knowledge graph construction, question answering, and text summarization. Unlike traditional information extraction, which relies on predefined relations, OpenIE extracts relationships in an open-domain manner, making it applicable across a wide range of texts.

This task presents a supervised learning approach to OpenIE, where the goal is to design a model that extracts multiple relation tuples from a sentence. Sentences may yield more than one extraction depending on the presence of multiple relationships or entities. For example, the sentence "Burnham died of heart failure at the age of 86, on September 1, 1947, at his home in Santa Barbara, California" generates multiple tuples that include location and time information.

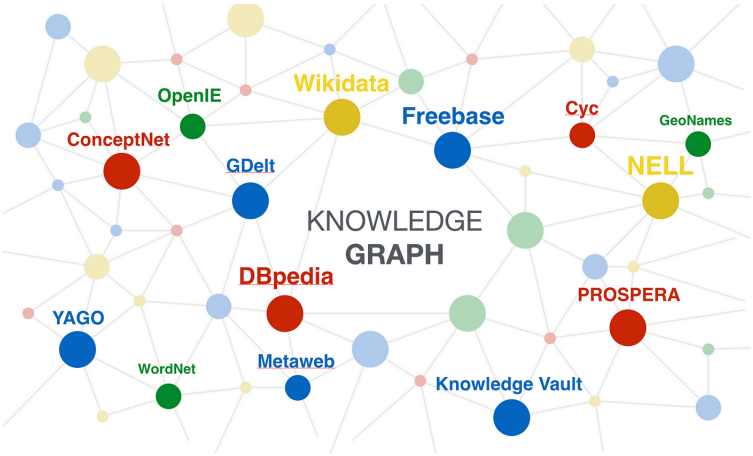

Figure 1: Knowledge Graph

This project also emphasizes error analysis to identify strengths and weaknesses of the model. Drawing inspiration from the REVERB paper, error analysis focuses on understanding where the model performs well and where it fails, providing insights for further improvements. Additionally, two variants of Word2Vec—Skip Gram and CBOW—are implemented from scratch to analyze their efficiency and quality of word embeddings.

2 Concepts

2.1 Word Embeddings

We have used BERT (Bidirectional Encoder Representations from Transformers) for generating embeddings for openIE task as it is a powerful transformer-based model that revolutionized natural language processing by providing deep contextualized word embeddings. Unlike traditional embeddings like Word2Vec or GloVe, which generate a single vector for each word regardless of context, BERT captures the meaning of a word in relation to its surrounding words. This bidirectional approach means BERT processes the entire sentence from both left-to-right and right-to-left, allowing it to generate more nuanced embeddings that reflect the word's meaning in different contexts.

For instance, the word "bank" will have different embeddings in "river bank" versus "financial bank" because BERT considers the context when generating the representation. These contextual embeddings can be directly used for downstream NLP tasks like Open Information Extraction (OpenIE), where each token's embedding can be used to classify it as a subject, relation, object, or other entity.

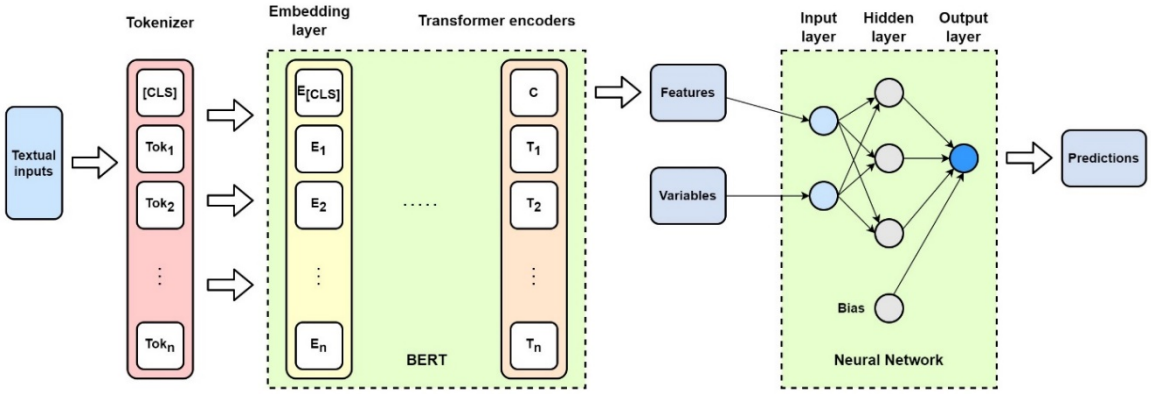

Figure 2: BERT Model

BERT embeddings are often fine-tuned for specific tasks, and in OpenIE, BERT can be used to generate token-level embeddings, which are then passed through models like BiLSTM-CRF to label tokens appropriately. This results in more accurate and context-sensitive extractions.

2.2 BiLSTM Model

A BiLSTM (Bidirectional Long Short-Term Memory) model is an extension of the LSTM architecture designed to capture dependencies in both directions within sequential data. LSTMs, by design, are capable of learning long-range dependencies by using a memory cell and gating mechanisms (input, output, and forget gates) to control the flow of information, making them highly effective in handling sequential data with varying time dependencies, such as text or speech.

In a BiLSTM, two LSTM layers are used: one processes the sequence from left to right

(forward pass), and the other processes it from right to left (backward pass). The outputs of both the forward and backward layers are concatenated at each time step, allowing the model to consider both past and future contexts simultaneously. This bidirectional structure makes BiLSTMs particularly useful for tasks like named entity recognition (NER), sentiment analysis, and relationship extraction, where understanding both preceding and following words enhances prediction accuracy.

For example, in an Open Information Extraction task, a BiLSTM model can leverage the bidirectional flow to better capture the relationship between a subject and object in a sentence, regardless of their positions. The architecture consists of an input layer (usually embeddings like BERT), the BiLSTM layers, and often a final layer (like CRF) for sequence labeling or classification. This combined architecture excels at handling complex dependencies within the data, improving the model's ability to generate accurate predictions.

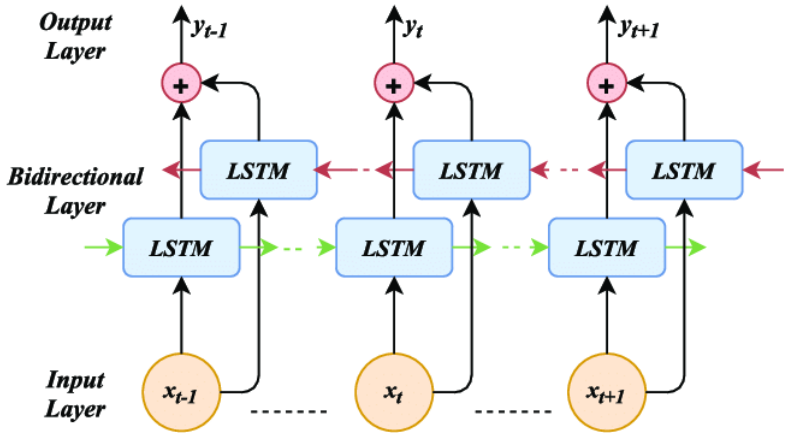

Figure 3: BiLSTM Architecture

2.2.1 BiLSTM CRF

A Conditional Random Field (CRF) layer is often added on top of a BiLSTM model in sequence labeling tasks to improve the overall prediction by considering the dependencies between output labels. While the BiLSTM layer captures context from both directions of the input sequence, the CRF layer helps ensure that the predicted sequence of labels is globally optimal by considering relationships between neighboring labels.

CRFs are particularly useful for structured prediction tasks, like named entity recognition (NER) or relationship extraction, where the prediction of one label depends on the others. For example, in a sentence, if one word is tagged as the start of an entity (e.g., 'ARG1'), it's likely that the following words are also part of the same entity, and CRF can enforce such constraints. By learning the transition probabilities between labels, the CRF layer helps ensure that the model produces consistent and valid label sequences.

In a BiLSTM-CRF model, the BiLSTM generates hidden states or embeddings for each token in a sentence, while the CRF layer takes these hidden states and predicts the

most probable sequence of labels by maximizing the conditional likelihood of the entire sequence, leading to more accurate and coherent predictions.

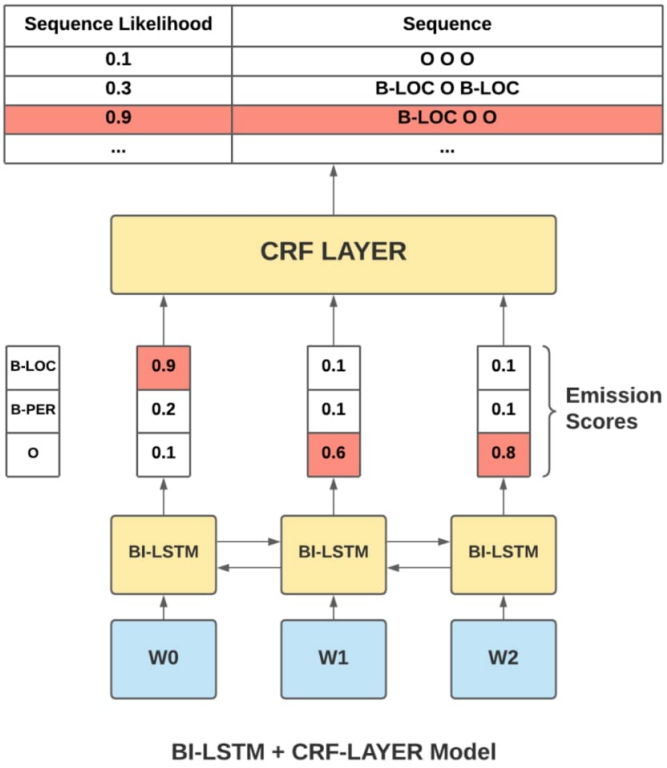

Figure 4: CRF

2.3 Word2Vec

Word2Vec is a popular technique for generating word embeddings, which are dense vector representations of words in a continuous vector space. Developed by Google in 2013, Word2Vec aims to capture semantic and syntactic relationships between words by representing words that appear in similar contexts with similar vectors. It achieves this by training on a large corpus of text, where words with similar meanings end up having close vector representations in the embedding space.

Word2Vec comes in two primary models: Skip-Gram and Continuous Bag of Words (CBOW). In the Skip-Gram model, the task is to predict the surrounding context words given a target word, while in CBOW, the goal is to predict the target word based on its surrounding words. Both models use a neural network with a single hidden layer.

These word embeddings can be used in various downstream tasks such as machine translation, document similarity, and named entity recognition, as they help encode word meaning in a way that captures both semantic similarity and contextual relationships.

2.3.1 Skip Gram

In Skip-Gram, the goal is to predict the surrounding context words (neighbors) for a given target word. For example, given the sentence "The cat sat on the mat," if the target word is "sat," the model would try to predict the words "The," "cat," "on," and "the" as its context.

Skip-Gram works by maximizing the probability of correctly predicting the surrounding words within a certain window size for each target word. A sliding window is moved across the text, and for each word (target), the model attempts to predict nearby words. The training objective is to adjust the word vectors in such a way that words occurring in similar contexts are close to each other in the vector space.

An efficient training approach for Skip-Gram involves negative sampling, which reduces computational complexity by sampling a few incorrect words ("negative samples") for each context prediction instead of updating the weights for all words in the vocabulary. This technique makes Skip-Gram scalable to large datasets. Skip-Gram embeddings are known to perform well in capturing the semantic and syntactic properties of words.

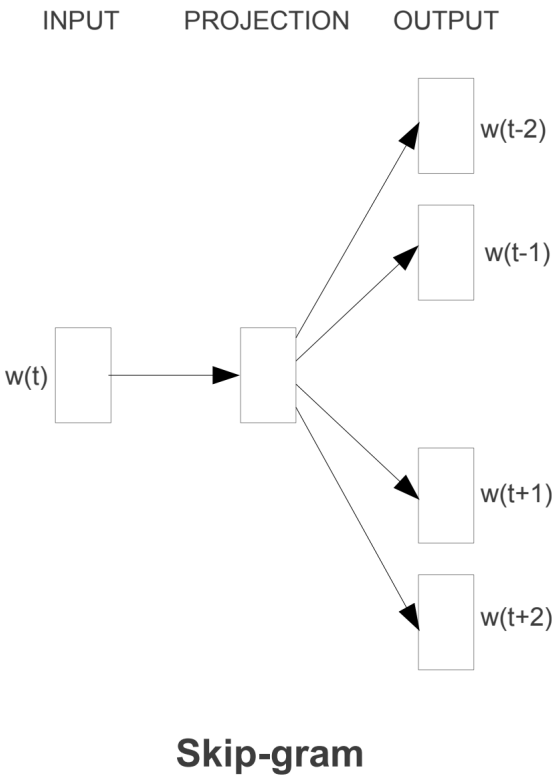

Figure 5: Skip Gram

2.3.2 CBOW

In contrast to Skip-Gram, CBOW predicts a target word based on its surrounding context words. For instance, given the sentence "The cat sat on the mat," and the context words

"The," "cat," "on," and "the," the CBOW model tries to predict the target word "sat."

CBOW works by taking the average of the vectors for the surrounding words (context) and using that to predict the vector for the target word. This method aims to maximize the likelihood of the target word appearing in the given context. Unlike Skip-Gram, which is more focused on predicting multiple context words for a single target, CBOW is designed to predict the target word from the entire context.

CBOW is generally faster to train than Skip-Gram, especially for large corpora, because it predicts one word from multiple context words, while Skip-Gram predicts many context words from a single target word.

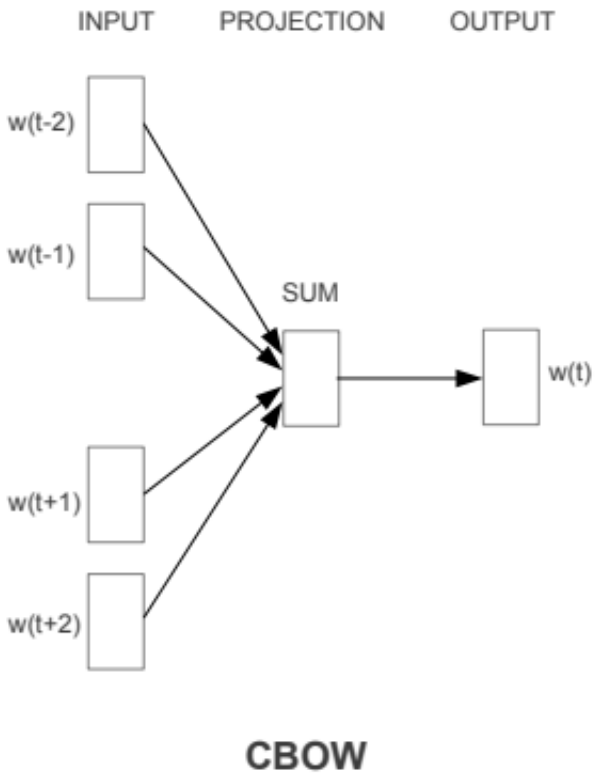

Figure 6: CBOW

3 Apparatus

Machine	Apple MacBook Pro M1 (2021)
RAM	8 GB LPDDR4X 4266MHz
Processor Name	Apple M1
Processor Specs	8 cores, 8 threads, 2.06-3.22 GHz, L1 Cache: 2MB, L2 Cache: 12MB
Python	Version 3.10

4 Dataset

The dataset for openIE task comprises of over 1e5 data points. Each token in the input sentence is labelled with one of six tags: ARG1, ARG2, REL, TIME, LOC, NONE. ARG1 and ARG2 refer to the subject and the object, respectively.

For Word2Vec task this dataset is used.

5 Methodology for OpenIE

5.1 Pre-Processing Data

For preprocessing, first the dataset was loaded from the text file using the below function then we tokenized the sentences given in the dataset using the **spacy** library, and then used the script given for combining the terms so that token and label count matches for each data point.

Algorithm 1 LOAD_DATASET

Input: $file\_path$ (Path to dataset file)
Output: $sentences$ (List of sentences), $labels$ (List of labels)
$sentences \leftarrow []$
$labels \leftarrow []$
$f \leftarrow \text{open}(file\_path, 'r')$
$lines \leftarrow f.readlines()$
$sentence \leftarrow []$
for each $line$ **in** $lines$ **do**
    $line \leftarrow line.strip()$
    if $line \neq \emptyset$ **then**
        if not $line$ **starts with** 'ARG1' **or** 'ARG2' **or** 'REL' **or** 'LOC' **or** 'TIME' **or** 'NONE' **then**
            $sentence \leftarrow line$
        else
            $current\_label \leftarrow line$
            $sentences.append(sentence)$
            $labels.append(current\_label)$
        end if
    end if
end for
return $sentences, labels$

Embeddings were generated using **BERT** and then we also added padding for each sentence to ensure uniformity across dataset as model will be trained in batches.

Tokenization: The input tokens are passed through the BERT tokenizer using the tokenizer() function. This tokenizer is special because it can split words into subword tokens if needed, which allows BERT to handle out-of-vocabulary or rare words by breaking them into smaller components. The tokenizer converts the words into the input format that BERT understands, and includes padding or truncation if necessary, though in this case, no padding or truncation is applied.

Generating BERT Embeddings: The tokenized input is fed into the BERT model to obtain contextual embeddings for each subword. BERT outputs the last_hidden_state, which contains a dense vector (embedding) for each subword token in the sequence. These embeddings capture the context and meaning of the words in their given sentence.

Mapping Subwords to Original Tokens: Since BERT's tokenizer splits some

words into multiple subword tokens, the code retrieves word_ids for each subword. This is crucial for knowing which subwords correspond to the same original word, as they need to be recombined into a single embedding.

Aggregating Subword Embeddings: Once the subword tokens are mapped to their original words using the word_ids, the next step is to aggregate the embeddings. For each original word, all its subword embeddings are averaged to create a single embedding that represents the entire word. The mean of the embeddings is used to ensure that the information from all subwords is combined effectively.

Final Embeddings: After aggregating subword embeddings for each token, the final list of embeddings for all tokens in the sentence is returned.

Algorithm 2 Get BERT Embeddings

Input: List of tokens
Output: Aggregated embeddings for each token
Initialize *inputs* ← `tokenizer`(*tokens*)
Use BERT model to compute *outputs* ← `bert_model`(*inputs*)
Extract *token_embeddings* ← `outputs.last_hidden_state`
Get word IDs *word_ids* ← `inputs.word_ids()`
Initialize empty list *aggregated_embeddings* ← []
Initialize empty list *current_token_embeddings* ← []
for each index i and *word_id* in *word_ids* **do**
  if *word_id* ≠ `None` **then**
    if *current_token_embeddings* is not empty and *word_id* ≠ *word_ids*$[i-1]$ **then**
      Compute mean of *current_token_embeddings* and append to *aggregated_embeddings*
      Reset *current_token_embeddings* ← []
    end if
    Append *token_embeddings*$[i]$ to *current_token_embeddings*
  end if
end for
if *current_token_embeddings* is not empty **then**
  Compute mean of *current_token_embeddings* and append to *aggregated_embeddings*
end if
Return *aggregated_embeddings*

For encoding the labels we used LabelEncoder(), also we added a new label called '**PADDING**' which will be used as a label for padded tokens in the sentence.

Then the dataset was split into train and validation sets using *train_test_split* function from sklearn and validation dataset size was 0.2

5.2 BiLSTM CRF Model

5.2.1 LSTM Layer

- **Input:** BERT embeddings of size 768 for each token.
- **Hidden Dimension:** 256 units. Since the LSTM is bidirectional, the hidden size becomes 512 (doubled).
- **Layers:** 2 layers of stacked LSTMs for deeper learning.
- **Bidirectional:** The LSTM captures context from both past and future directions in the sequence.

5.2.2 Dropout Layer

- A dropout probability of 0.5 is applied to the LSTM outputs to reduce overfitting.

5.2.3 Batch Normalization Layer

- The LSTM outputs are batch-normalized to speed up convergence and make the training process more stable.

5.2.4 Fully Connected Layers

- **FC1:** Linear layer with input size of 512 (from bidirectional LSTM) and output size of 256.
- **FC2:** Linear layer with input size of 256 and output size of 128.
- **FC3:** Final linear layer mapping to the number of output labels.
- **Activation:** ReLU activation function is applied after each linear transformation.

5.2.5 Conditional Random Field (CRF)

- The CRF layer models dependencies between output labels to ensure that valid label sequences are predicted.
- **Training:** The CRF computes the loss if ground truth labels are provided.
- **Inference:** The CRF decodes the most probable sequence during testing.

5.3 Training the Model

1. Hyperparameters

Learning Rate (lr = 0.001): This controls the step size during gradient descent. A smaller learning rate leads to slower convergence, but can potentially avoid overshooting minima.

Number of Epochs (num_epochs = 100): This defines how many times the model will iterate over the entire training dataset.

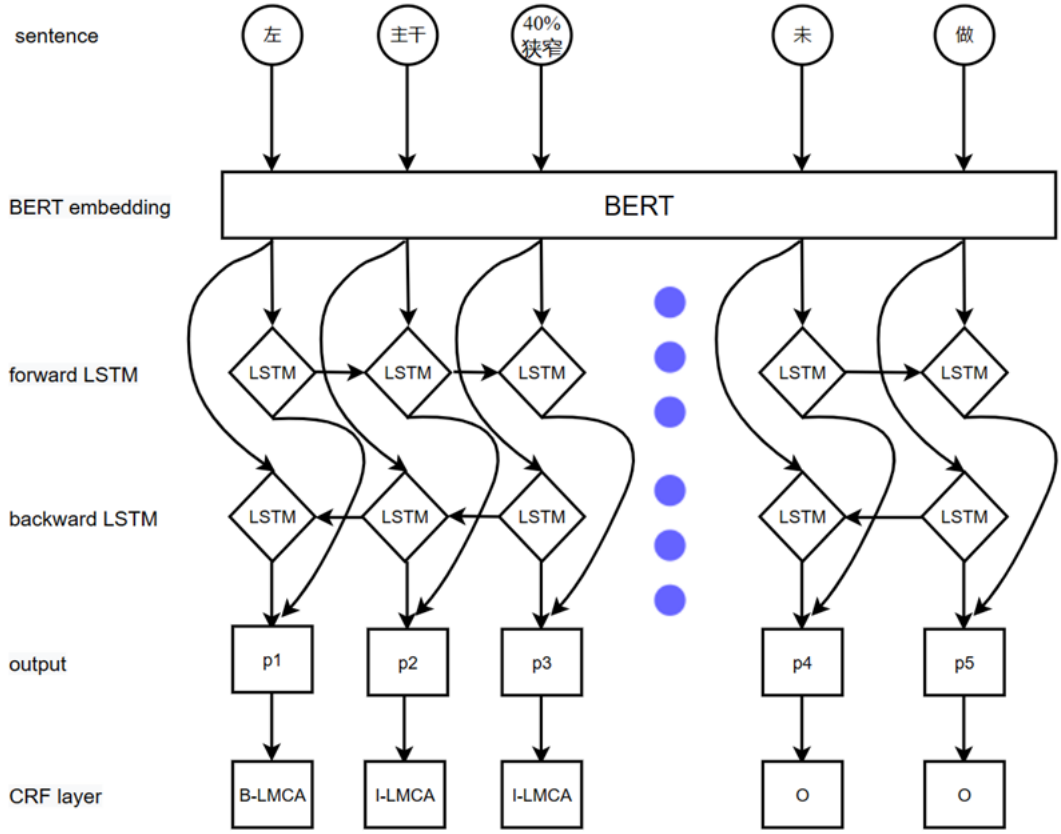

Figure 7: BERT-BiLSTM-CRF Model

Batch Size: Controlled by train_loader, the batch size determines how many samples are passed through the network before an update is made. This is important for stabilizing gradients during training.

2. Loss Function

The loss function used is **CrossEntropyLoss** with an *ignore_index* argument to skip over padding tokens:

CrossEntropyLoss: This is a commonly used loss function for multi-class classification tasks. It calculates the difference between the predicted probability distribution (logits) and the true labels.

Padding Mask: Labels corresponding to padding tokens are ignored using the *ignore_index=label_to_idx['PADDING']*. Padding is used to ensure uniform sequence lengths in batches, and this prevents these artificial tokens from influencing the loss calculation.

3. Optimizer

Adam Optimizer: Adam combines the advantages of both AdaGrad (good for sparse gradients) and RMSProp (good for non-stationary objectives). It adjusts the learning rate for each parameter dynamically, making it a popular choice for deep learning.

Algorithm 3 Training BiLSTM-CRF Model

Input: Train loader, number of epochs num_epochs, learning rate lr
Output: Trained model, loss per epoch
Initialize $optimizer \leftarrow$ `Adam(model.parameters(), lr)`
Initialize $loss\_fn \leftarrow$ `CrossEntropyLoss` with padding index ignored
Initialize $epoch\_losses \leftarrow []$
for each $epoch \in [1, num\_epochs]$ **do**
    Set model to training mode
    Initialize $total\_loss \leftarrow 0$
    for each $batch \in train\_loader$ **do**
        Get $inputs$, $labels$ from $batch$
        Create $mask$ for valid tokens where $labels \neq PADDING$
        Compute $loss \leftarrow model(inputs, tags = labels, mask = mask)$
        Zero the gradients in $optimizer$
        Backpropagate by $loss.backward()$
        Update model parameters using $optimizer.step()$
        Add $loss$ to $total\_loss$
    end for
    Compute $avg\_loss \leftarrow total\_loss/len(train\_loader)$
    Append avg_loss to $epoch\_losses$
    Print "Epoch $epoch/num\_epochs$, Loss: avg_loss"
end for

4. Masking

Masking is applied to ensure that only the valid tokens (i.e., non-padding tokens) are used to calculate the loss. The mask is computed using *labels.ne(label_to_idx['PADDING'])* condition, which checks for tokens that are not padding. This helps in preventing the model from learning patterns based on padding tokens.

5. Training Process

The model is trained over *100* epochs and for *4500* sentences, where in each epoch the model is set to training mode (*model.train()*). For each batch, the loss is computed only for valid tokens (non-padding). Gradients are backpropagated, and the optimizer updates the weights to minimize the loss. The average loss is calculated for each epoch and stored for later analysis.

5.4 Evaluation on Validation Data

The trained model was evaluated on the validation dataset giving accuracy of *61%*. This metric was found by calculating the number of labels that were matching with the ground labels and this was computed for non padding labels.

5.5 Extracting Relations

For the test dataset, pre-processing was done in a similar way by first tokenizing the sentences and then generating embeddings using BERT. The pre-trained model was used

Figure 8: Training Loss

for labelling task. Once we got the labels for each token for the test dataset, next step is to extract multiple relations from this output given by our model.

We followed different approaches:

5.5.1 Enforce a Strict Pattern on the Extractions

Make sure the extractions follow one of the possible combinations.

Standard pattern: ARG1 before REL, ARG2 after REL
Example: ARG1 (subject) is followed by REL (relation) and then ARG2 (object).
Output: "Sentence" — REL — ARG1 — ARG2 — TIME — LOC

Inverted pattern: ARG2 before REL, ARG1 after REL
Example: ARG2 (object) comes before REL (relation), followed by ARG1 (subject).
Output: "Sentence" — REL — ARG2 — ARG1 — TIME — LOC

Both arguments before REL: ARG1 before ARG2, both precede REL
Example: Both ARG1 and ARG2 occur before REL.
Output: "Sentence" — REL — ARG1 — ARG2 — TIME — LOC

Both arguments after REL: REL is followed by ARG1 and then ARG2
Example: REL is followed by ARG1 and then ARG2.
Output: "Sentence" — REL — ARG1 — ARG2 — TIME — LOC

Loosened patterns: If only ARG1 or ARG2 is present with REL, extract a partial relationship. Output includes only the available argument along with REL.

Handling Missing Elements

No relation (REL) found: If neither REL nor ARG1/ARG2 is found but time or location is detected, extract just the time or location information.

Algorithm 4 Relationship Extraction Algorithm (Strict Pattern)

Input: Model *model*, Padded embeddings *padded_embeddings*, Test data *df_test*
Output: Results *results*, Skipped count *skipped_count*
Initialize *results* ← [], *skipped_count* ← 0
for each sentence *i* in *df_test* **do**
    Get predicted labels *predicted_labels*
    Initialize flags and position markers for ARG1, ARG2, REL, TIME, LOC
    for each token in *predicted_labels* **do**
        Identify and group tokens into ARG1, ARG2, REL, TIME, LOC
    end for
    if REL is found **then**
        if Standard pattern ARG1 before REL, ARG2 after REL **then**
            Extract relation with ARG1, REL, ARG2, TIME, LOC
        else if Inverted pattern ARG2 before REL, ARG1 after REL **then**
            Extract inverted relation with ARG2, REL, ARG1, TIME, LOC
        else if Both arguments before REL or after REL **then**
            Extract relation
        end if
    else
        if Only time/location present **then**
            Extract partial information with TIME/LOC
        else
            Skip the sentence
            Increment *skipped_count*
        end if
    end if
    Append sentence results to *results*
end for
Return *results*, *skipped_count*

Output: "Sentence" — 0.5 — [empty] — [empty] — TIME — LOC

Confidence of *0.5* is assigned for these type of extractions. Otherwise confidence value of *1* is assigned for the extractions following any of the above patterns.

Complete skip:

If no meaningful relation, time, or location can be identified, the sentence is skipped, and a warning is logged.

Observations

Using this approach on test dataset, CaRB metric gave a high precision value of $\tilde{0.72}$, as most of the extractions were correct given a strict pattern was enforced, but at the same time recall value was very low $\tilde{0.212}$ indicating that many valid extractions were skipped which did not follow the above pattern, this could be verified by checking the number of generated extractions. There were $\tilde{300}$ extractions generated while test dataset contained *641* sentences. This is because many sentences were being skipped as most of the labels in them were NONE, hence not following any of the above pattern. F1 score value was *.328* and AUC was *.183*.

```
AUC: 0.183       Optimal (precision, recall, F1): [0.72  0.212 0.328]
```

5.5.2 Assign Confidence Value to Extractions

This approach follows the same methodology as the above approach, that is checking all possible patterns. The only difference in this approach is that it assigns a confidence score for each extraction.

We have two functions one for extracting confidence features and other one computes the overall confidence for a given extraction.

Algorithm 5 Extract Confidence Features

function EXTRACT_CONFIDENCE_FEATURES(sentence, rel, arg1, arg2, tokens)
    Initialize $features \leftarrow \{\}$
    $features$['covers_all_words'] $\leftarrow$ (len(arg1.split()) + len(rel.split()) + len(arg2.split()) == len(tokens))
    $last\_token\_rel \leftarrow$ last token in $rel.split()$ or empty string
    $features$['last_prep'] $\leftarrow$ boolean values for preps ['for', 'on', 'of', 'to', 'in']
    $features$['short_sentence'] $\leftarrow$ (len(tokens) ≤ 10)
    $features$['medium_sentence'] $\leftarrow$ ($10 <$ len(tokens) ≤ 20)
    $features$['long_sentence'] $\leftarrow$ (len(tokens) > 20)
    Define $wh\_words \leftarrow$ ['who', 'what', 'where', 'when', 'why', 'how']
    $features$['wh_word_left'] $\leftarrow$ True if any wh_word appears before rel
    $features$['r_matches_vw_p'] $\leftarrow$ match VW*P pattern
    $features$['r_matches_v'] $\leftarrow$ match V pattern
    $features$['sentence_starts_with_x'] $\leftarrow$ ($sentence$ starts with $arg1$)
    $features$['arg1_is_proper_noun'] $\leftarrow$ arg1.istitle()
    $features$['arg2_is_proper_noun'] $\leftarrow$ arg2.istitle()
    return $features$
end function

Algorithm 6 Confidence Score Calculation for Relationship Extraction

Sentence s, Relation r, Argument 1 $arg1$, Argument 2 $arg2$, Tokens $tokens$ Confidence Score CS
$features \leftarrow extract\_confidence\_features(s, r, arg1, arg2, tokens)$
$weights \leftarrow$ {weights for each feature}
$confidence\_score \leftarrow 0$

for each feature f in $features$ **do**

if $f \in weights$ **then** $confidence\_score \leftarrow confidence\_score + weights[f] \cdot features[f]$
$bias \leftarrow 3.0 \quad confidence\_score \leftarrow confidence\_score + bias$
$normalized\_score \leftarrow \frac{(confidence\_score - min\_score)}{(max\_score - min\_score)}$
$CS \leftarrow \max(0, \min(normalized\_score, 1))$
return CS

Features for confidence extraction were taken from the [FSE11] paper, weights for each feature were slightly modified in order to get better results.

Observations

Using this approach we got almost similar results, infact we observed a decrease in precision value resulting in a lower F1 score. AUC value was also slightly less about *.158*.

```
AUC: 0.158      Optimal (precision, recall, F1): [0.701 0.212 0.326]
```

5.6 Error Analysis

The following results have been prepared by comparing our extractions on the test dataset with the gold extractions.

Incorrect Extractions	**Percentage**
Correct relation, incorrect arguments	36.97%
Correct relation, incorrect argument order	0.00%
Correct relation, missing arguments	0.47%
N-ary relation	0.00%
Non-contiguous relation phrase	2.37%
Imperative verb	0.00%
Overspecified relation phrase	11.37%
Other (POS/chunking errors)	48.82%

Table 1: Incorrect Extractions

Most of the incorrect extractions that were classified, given by our model are due to arguments being incorrect. This is mainly because our model was only trained on $\tilde{4}500$ sentences and over 100 epochs, hence could not label the arguments correctly. There are no extractions with n-ary relations because the algorithms we used above have a strict pattern enforcing the argument count to be two. Significant chunk of extraction have overspecified relation phrase too. As there are no such extractions where argument order

is incorrect signifies that CRF layer is effective at preserving sequence order. Also there were no incorrect extractions due to imperative verbs concludes that model is effectively distinguishing between declarative and imperative sentences this could be because BERT is pre-trained on a large dataset, hence able to give appropriate embeddings.

Missed Extractions	**Percentage**
Could not identify correct arguments	100.00%
Relation filtered out by lexical constraint	0.00%
Identified a more specific relation	0.00%
POS/chunking error	0.00%

Table 2: Missed Extractions

All the extractions that were missed are because of incorrect labelling of arguments. We also observed that there were lot of sentences for which no extractions were being produced, this was due to the fact that model predicted all labels as NONE for these sentences, hence there were no valid arguments present. This could be due to model being not properly trained because of computing constraints.

5.6.1 Examples

Model was able to produce accurate extractions for the following sentences:

1. 32.7% of all households were made up of individuals and 15.7 % had someone living alone who was 65 years of age or older.

2. A CEN forms an important but small part of a Local Strategic Partnership.

Some sentences where model *predicted all labels as NONE* are given below:

1. A partial list of turbomachinery that may use one or more centrifugal compressors within the machine are listed here

2. Although under constant attack from kamikazes as well as fighters and dive-bombers , " Hazelwood " came through the invasion untouched and on the night of 25 February sank two small enemy freighters with her guns

6 Methodology for Word2Vec

6.1 Data Cleaning and Tokenization

1. **Corpus Input:** The process begins with a raw text corpus that may contain various forms of textual noise, including punctuation, numbers, and special characters. The corpus is input as a single string. In our case, there were various japanese texts and unknown tokens which had to be taken care of, and we removed roman numerals as well.

2. **Removal of Non-Alphabetic Characters:** To ensure the integrity of the data, a regular expression is employed to remove Roman numerals and any numeric characters. This helps to focus solely on the textual content, which is essential for meaningful word representation.

3. **Punctuation Elimination:** All punctuation marks are stripped from the text using a translation table. This step further cleans the text and avoids introducing any noise that could interfere with word tokenization.

4. **Tokenization:** The cleaned corpus is then tokenized into sentences by splitting it at periods. Each sentence is subsequently converted into a list of words. This hierarchical structure (sentences containing words) is crucial for further processing and analysis.

6.2 Filtering Sentences and Words

1. **Minimum Sentence Length:** To maintain a quality dataset, sentences are filtered based on a minimum length threshold. Sentences shorter than a specified number of words (e.g., 8 words) are discarded. This helps ensure that the remaining sentences contain sufficient contextual information for training.

2. **Word Frequency Analysis:** A frequency count of all words across the entire corpus is performed using the `Counter` class from the `collections` module. This step allows the identification of rare words that may not contribute meaningfully to the model.

3. **Rare Word Removal:** Sentences are further filtered to exclude words that occur less frequently than a predefined threshold (e.g., 10 occurrences). This reduces the vocabulary size and focuses on more commonly used words, which are more likely to yield useful embeddings.

4. **Final Sentence Filtering:** After filtering for rare words, any sentences that fall below the minimum sentence length threshold are removed. This final check ensures that all sentences in the dataset are suitable for training purposes.

6.3 Output

The preprocessing function returns a list of cleaned and filtered sentences, where each sentence is represented as a list of words. This structured output serves as the input for subsequent modeling tasks, such as training the CBOW and Skip-Gram models in Word2Vec.

6.4 Skip-Gram Model

The Skip-Gram model is a neural network-based approach for learning word embeddings by predicting the context words given a target word. The following sections outline the key components of the Skip-Gram model, including the architecture, training process, and methods for generating predictions.

6.4.1 Model Architecture

The Skip-Gram model consists of an input layer, a hidden layer, and an output layer:

1. **Input Layer:** The input layer consists of one-hot encoded vectors representing the target word. Each vector has a dimension equal to the vocabulary size, where only the index corresponding to the target word is activated.

2. **Hidden Layer:** This layer contains a fixed number of neurons (in this implementation, 20) and does not have an activation function. The weights connecting the input layer to the hidden layer are randomly initialized. The output from the hidden layer is a dense representation of the target word.

3. **Output Layer:** The output layer also consists of a one-hot encoded vector representing the context words. The model uses softmax activation to compute the probability distribution over the vocabulary for context words.

6.4.2 Training Process

The training process involves several key steps:

1. **Weight Initialization:** Weights for the connections between the input and hidden layers (W) and the hidden and output layers (W_1) are initialized randomly within a specified range.

2. **Loss Calculation:** The model employs the cross-entropy loss function to quantify the difference between the predicted probabilities and the actual context words. The loss is computed for each training example by aggregating contributions from all context words.

3. **Feedforward Propagation:** For each training instance, the model computes the hidden layer activations (h) by taking the dot product of the input vector with the weight matrix W. The output layer activations (u) are then calculated using the hidden layer output and the weight matrix W_1. The softmax function is applied to obtain the predicted probabilities (y) for context words.

4. **Backpropagation:** The error is computed as the difference between the predicted probabilities and the actual context word vector. Gradients are calculated for both sets of weights (W and W_1) using the chain rule, allowing for updates to be made via gradient descent.

5. **Adaptive Learning Rate:** The learning rate is adjusted dynamically based on the iteration number to improve convergence and optimize training efficiency.

6.4.3 Prediction Mechanism

To generate predictions for context words given a target word:

1. **Input Representation:** A one-hot encoded vector for the target word is created, similar to the training phase.

2. **Feedforward Calculation:** The hidden layer activation is calculated, followed by the output layer activation. The softmax function is applied to produce a probability distribution over the vocabulary.

3. **Top Context Words Retrieval:** The predicted probabilities are sorted, and the top context words are retrieved based on the highest probabilities. This provides insight into the most likely context words for the given target word.

6.4.4 Output and Embeddings

The model provides two key outputs:

1. **Embedding Matrix:** The learned word embeddings can be accessed as a matrix where each row corresponds to a word in the vocabulary.

2. **Individual Word Embedding:** The embedding for a specific word can be retrieved using its index in the vocabulary.

6.5 Skip-Gram Model with Negative Sampling

The Skip-Gram model with negative sampling is an enhancement of the traditional Skip-Gram approach, designed to improve training efficiency and the quality of learned embeddings. This method addresses the challenge of efficiently updating the weights when training on large corpora by focusing on a small number of context words and selected negative samples.

6.5.1 Model Architecture

Similar to the standard Skip-Gram model, the architecture includes an input layer, a hidden layer, and an output layer, with modifications to incorporate negative sampling.

1. **Input Layer:** The input layer remains a one-hot encoded vector for the target word, as in the original Skip-Gram model.

2. **Hidden Layer:** The hidden layer consists of a fixed number of neurons (20 in this implementation) and serves as the representation of the target word.

3. **Output Layer:** The output layer produces probabilities for both positive and negative samples. In this case, softmax activation is still used, but the focus is on a limited set of output neurons that correspond to the positive context words and a selected number of negative samples.

6.5.2 Training Process with Negative Sampling

The training process under negative sampling includes the following steps:

1. **Weight Initialization:** Weights are initialized similarly to the standard Skip-Gram model, creating matrices W and W_1.

2. **Loss Calculation:** Negative sampling alters the loss function. The model calculates the loss for positive samples (actual context words) and negative samples (randomly chosen words not in the context).

3. **Feedforward Propagation:** For each training instance, the hidden layer activations (h) are computed as before. However, instead of calculating the full softmax over the entire vocabulary, we focus on the context words and the negative samples, thus optimizing computation.

4. **Negative Sampling:** A small number of words from the vocabulary, which are not part of the context words, are randomly selected as negative samples. The number of negative samples significantly affects training efficiency and model performance.

5. **Backpropagation:** The error is calculated for both positive and negative samples. Gradients are computed, allowing for the weights W and W_1 to be updated using gradient descent.

6. **Adaptive Learning Rate:** As in the traditional model, the learning rate is adjusted throughout training to enhance convergence.

6.5.3 Choosing the Number of Negative Samples

Selecting the number of negative samples is critical for balancing training speed and the quality of the learned representations. In this implementation, we chose to use 5 negative samples based on several factors:

1. **Empirical Results:** Research and empirical studies suggest that using around 5 negative samples often strikes a good balance between computational efficiency and performance. This number allows the model to learn effectively from a meaningful contrast between positive and negative examples.

2. **Training Time vs. Quality:** Increasing the number of negative samples can improve the model's ability to distinguish between similar words, but it also significantly increases training time. Five samples have been found to provide sufficient learning without excessively prolonging the training process.

3. **Context Size Consideration:** Given the context window size of 2, five negative samples provide a reasonable number of contrasts, allowing the model to learn discriminative features effectively while keeping the computational overhead manageable.

6.5.4 Prediction Mechanism

The prediction mechanism follows similar steps as the traditional Skip-Gram model, with adaptations to incorporate negative samples:

1. **Input Representation:** A one-hot encoded vector for the target word is created.

2. **Feedforward Calculation:** The hidden layer activation and output layer probabilities are calculated, focusing only on the selected context words and negative samples.

3. **Top Context Words Retrieval:** The predicted probabilities are sorted, and the most likely context words are returned based on the computed scores.

6.6 Continuous Bag of Words (CBOW) Model

The Continuous Bag of Words (CBOW) model is an approach for learning word embeddings by predicting a target word based on its surrounding context words. Unlike the Skip-Gram model, which uses a target word to predict its context, CBOW takes multiple context words as input to predict a single target word.

6.6.1 Model Architecture

The architecture of the CBOW model includes an input layer, a hidden layer, and an output layer. The main components are as follows:

1. **Input Layer:** The input layer consists of context words represented as one-hot encoded vectors. For each target word, the model utilizes multiple context words defined by a specified context window.

2. **Hidden Layer:** The hidden layer contains a fixed number of neurons (20 in this implementation) and is responsible for learning the representation of the context words. The context vectors are summed to create a single input vector for the hidden layer.

3. **Output Layer:** The output layer consists of a softmax function that produces a probability distribution over the vocabulary, allowing the model to predict the target word based on the learned context representation.

6.6.2 Training Process

The training process for the CBOW model involves several steps:

1. **Weight Initialization:** Weights are initialized for the input-to-hidden layer and hidden-to-output layer, denoted as W and W_1, respectively.

2. **Feedforward Propagation:** For each training instance, the model computes the context vector by summing the one-hot encoded vectors of the context words. This context vector is then used to compute the hidden layer activations.

3. **Output Calculation:** The output layer computes the probabilities of each word in the vocabulary being the target word. This is achieved using the softmax function on the output layer.

4. **Loss Calculation:** The loss is calculated using the cross-entropy loss function, comparing the predicted probabilities with the actual target word's one-hot encoded vector. This loss guides the optimization process.

5. **Backpropagation:** Gradients of the loss with respect to the weights are computed, and the weights W and W_1 are updated using gradient descent.

6. **Adaptive Learning Rate:** Similar to the Skip-Gram model, the learning rate is adjusted throughout the training process to enhance convergence.

6.6.3 Prediction Mechanism

The prediction mechanism for the CBOW model follows these steps:

1. **Input Representation:** The context words are represented as one-hot encoded vectors, which are summed to form the context vector.

2. **Feedforward Calculation:** The context vector is passed through the hidden layer to obtain the hidden layer activations, which are then used to compute the output layer probabilities.

3. **Target Word Retrieval:** The predicted probabilities are processed to identify the target word with the highest probability, which is returned as the output.

6.6.4 Advantages of CBOW

The CBOW model is particularly advantageous in scenarios where:

1. **Simplicity and Efficiency:** By using context words to predict a single target word, CBOW is computationally efficient, especially in large datasets with extensive vocabularies.

2. **Contextual Learning:** The model effectively captures the semantic meaning of words by leveraging multiple context words, leading to high-quality word embeddings.

6.7 Training Methodology for Skip-Gram Model

The training of the Skip-Gram model is a systematic process aimed at learning meaningful word embeddings from a given corpus. This section details the data preparation, batch generation, training loop, and overall training duration.

6.7.1 Data Preparation

1. **Sentence Tokenization:** The corpus is tokenized into individual sentences.

2. **Vocabulary Construction:** A dictionary is created where each unique word is assigned an index.

   The context window size is defined as 2, meaning that for each target word, the model will consider two words on either side as context. The vocabulary size V is computed based on the unique words in the dataset.

6.7.2 Batch Generation

Given the large size of the dataset, generating word-context pairs for each training iteration can be computationally intensive and can consume a significant amount of memory. To address this challenge, we implement a batch generation strategy. This involves processing the data in smaller, manageable chunks, which is essential for efficient memory utilization and faster training times.

In our implementation, we selected a subset of 5000 sentences from the dataset. This selection resulted in approximately 650 batches, with each batch containing 1000 word-context pairs. The batching approach not only streamlines the training process but also ensures that the model learns effectively from a diverse range of word-context pairs.

6.7.3 Training Loop

The training of the Skip-Gram model is organized into a series of epochs. In our case, we set the number of epochs to 2. The training loop encompasses the following steps:

Algorithm 7 Training Algorithm for Skip-Gram Model

```
Input: Corpus of sentences, Vocabulary, Context Window Size, Batch Size
Initialize: Model with random weights
for each epoch in number of epochs do
    Print "Starting epoch"
    for each batch in batch generator do
        Update model training data with current batch
        model.train(mytol=1e-4, maxepochs=1)
    end for
    Print "Finished epoch"
end for=0
```

6.8 Mean Reciprocal Rank (MRR) Metric

The Mean Reciprocal Rank (MRR) is a statistical measure used to evaluate the effectiveness of systems that return a ranked list of items. It is particularly useful in the context of information retrieval and recommendation systems, where it assesses how well a model can predict relevant items based on a given input.

The MRR is defined as the average of the reciprocal ranks of the first relevant item in the list of results returned by the model. Formally, it can be expressed as:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$$

where $|Q|$ is the total number of queries, and rank_i is the rank position of the first relevant item for the i^{th} query. An MRR value closer to 1 indicates better performance, as it suggests that the relevant items are ranked higher in the returned list.

In the context of the Skip-Gram model, MRR serves as a useful metric for evaluating the quality of word embeddings, particularly how effectively the model captures semantic relationships between words.

6.9 Model Performance and Training Time Comparison

In this subsection, we evaluate the Mean Reciprocal Rank (MRR) and the training time of two variations of the SkipGram model: one using softmax and the other using negative sampling. Both models were trained on datasets of varying sizes, with a batch size of 500 word-context pairs and for 5 epochs. The models were tested using window sizes of 2, 3, and 4, and the results are discussed below.

6.9.1 Mean Reciprocal Rank (MRR)

The Mean Reciprocal Rank (MRR) is a metric that evaluates the quality of the ranked predictions generated by the models. A higher MRR value indicates better performance, as it implies that the model ranks the correct word (target) closer to

the top of its predicted list. Table 3 displays the MRR scores for the SkipGram model trained with softmax and negative sampling across different window sizes.

Table 3: Mean Reciprocal Rank (MRR) for SkipGram with Softmax and Negative Sampling

Window Size	SkipGram with Softmax (MRR)	SkipGram with Negative Sampling (MRR)
2	0.0502	0.0709
3	0.0609	0.0800
4	0.0702	0.0839

From the results, we observe that the SkipGram model using negative sampling consistently outperforms the model using softmax in terms of MRR, regardless of the window size. This pattern suggests that negative sampling is better at capturing relevant semantic relationships between words, especially as the window size increases. In other words, the model with negative sampling is more effective at predicting context words that are meaningful in larger contexts.

6.9.2 Training Time

The training time for each model, corresponding to the same window sizes as in the MRR evaluation, is presented in Table 4. The time is recorded in seconds.

Table 4: Training Time (in seconds) for SkipGram with Softmax and Negative Sampling

Window Size	SkipGram with Softmax (Time)	SkipGram with Negative Sampling (Time)
2	561.90	227.89
3	780.35	339.58
4	1479.84	455.05

On average, the SkipGram model with softmax takes significantly more time to train compared to the model using negative sampling. The average training time for SkipGram with softmax across all window sizes is approximately 940.70 seconds, whereas for SkipGram with negative sampling, the average training time is around 340.84 seconds.

6.9.3 Analysis and Findings

The large discrepancy in training times between the two models can be attributed to the difference in how each method computes the probability distribution over words during training:

- **SkipGram with Softmax:** In the softmax variant, the model computes the probability distribution across all words in the vocabulary to predict the target word for a given context. This involves calculating the dot product between the context word's embedding and the embedding of every word in the vocabulary, followed by normalizing these values using the softmax function. As a result, the time complexity of this method is proportional to the size of the vocabulary, which can be quite large (in the order of tens of thousands or more). This leads to significantly longer training times, especially when training on larger datasets or with larger window sizes.

- **SkipGram with Negative Sampling:** In contrast, the negative sampling variant bypasses the need to compute the probability distribution for every word in the vocabulary. Instead, it samples a small number of negative examples (words that are not the correct prediction) and focuses on updating the model's parameters only for these sampled words, along with the positive examples (the correct target word). This reduces the computational cost to a small, fixed number of operations per training instance, regardless of the size of the vocabulary. Thus, negative sampling has a much lower time complexity, which results in faster training times.

This key difference in computational complexity explains why the SkipGram model with negative sampling is much more efficient than the softmax variant, particularly when training on large datasets or using larger window sizes. Negative sampling allows the model to focus on a subset of words, making it more scalable and less computationally intensive, while softmax requires evaluating every possible word in the vocabulary, leading to higher resource consumption.

6.9.4 Conclusions

The results indicate that negative sampling offers both computational efficiency and superior performance in word prediction tasks. SkipGram with negative sampling consistently achieves higher MRR values across all window sizes and trains significantly faster. This makes negative sampling the preferred method, especially for large-scale training scenarios, where computational resources are a constraint.

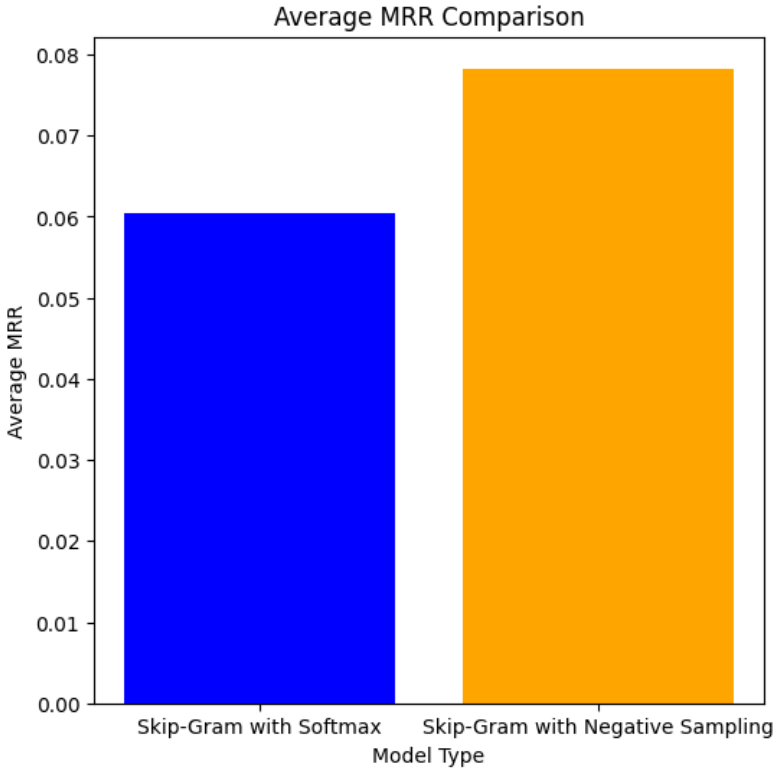

Figure 9: MRR Comparison

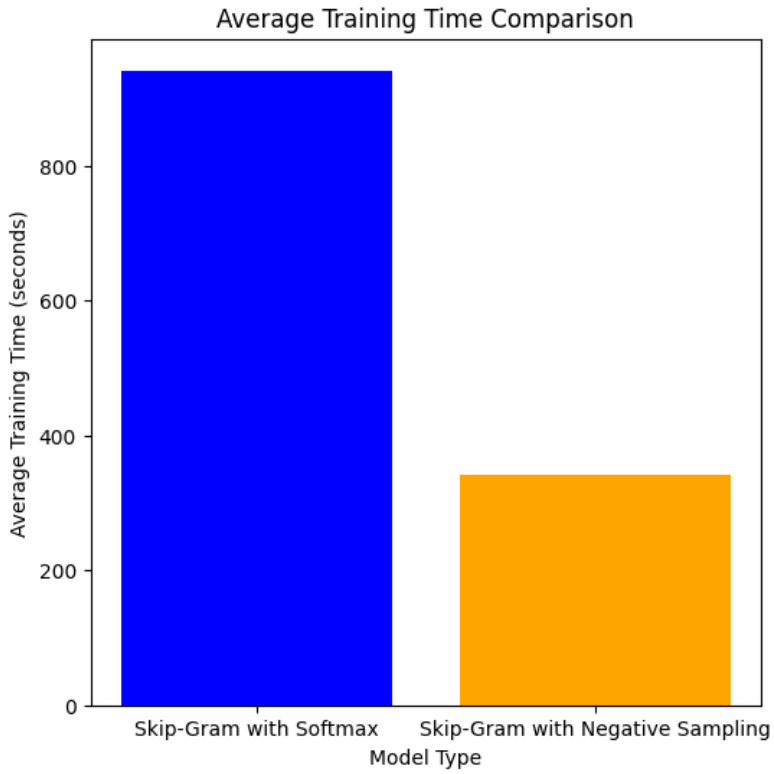

Figure 10: Training Time

6.10 CBOW Results and Explanation

The Continuous Bag of Words (CBOW) model was trained and tested on a dataset with specific configurations to evaluate its performance in terms of Mean Reciprocal Rank (MRR) and training time. The model was trained on **10,000 sentences** for **10 epochs**, with a batch size of **64 word-context pairs**, and each batch was trained for 10 epochs. The model was then tested on **1,000 sentences** using three different window sizes: 2, 3, and 5. This subsection discusses the observed performance, training time, and provides an analysis of the results.

6.10.1 Mean Reciprocal Rank (MRR)

MRR is a metric used to evaluate the ranking performance of models, where a higher score indicates that the model ranks the correct target word closer to the top of the predicted list. The MRR values obtained for different window sizes are as follows:

$$\text{Window size 2: } MRR = 0.4923$$

$$\text{Window size 3: } MRR = 0.2304$$

$$\text{Window size 5: } MRR = 0.4671$$

Analysis of MRR:

- **Window Size 2:** The highest MRR (0.4923) was achieved with a window size of 2. This indicates that the model performs well when the context is small, likely due to the strong association between nearby words. With a smaller context, the model can focus on fewer words, leading to more accurate predictions of the target word.
- **Window Size 3:** For a window size of 3, the MRR dropped significantly to 0.2304. This decline suggests that expanding the context by a single word introduces more noise, making it harder for the model to correctly predict the target word. The addition of extra context words may dilute the importance of the immediate neighbors, reducing the overall prediction accuracy.
- **Window Size 5:** With a larger window size of 5, the MRR improves to 0.4671. Although this score is lower than that for a window size of 2, it is significantly better than the score for window size 3. The larger context allows the model to capture broader semantic relationships between words, which benefits the prediction task in some cases. However, the improvement in MRR for window size 5 may not be as pronounced as with a smaller window because the expanded context could still introduce irrelevant words.

Overall, the MRR results suggest that smaller window sizes are more effective for this particular CBOW model, where local context plays a more crucial role in word prediction accuracy.

6.10.2 Training Time

The training time for each window size was recorded as follows:

Window size 2: 230.87 seconds

Window size 3: 260.55 seconds

Window size 5: 415.78 seconds

Analysis of Training Time:

- **Window Size 2:** The training time for a window size of 2 is the shortest at 230.87 seconds. This is because the model only needs to process a small number of context words for each target word. A smaller window size reduces the number of computations required for each word pair, making the training process faster.
- **Window Size 3:** For a window size of 3, the training time increases to 260.55 seconds. This is expected as the model needs to process a larger context for each word pair, resulting in more computations per training instance. The time increase is not drastic, as only one more word is added to the context.
- **Window Size 5:** The largest window size (5) leads to a significant increase in training time, with 415.78 seconds. The training time scales with the window size because the model must process a much larger number of word pairs, which increases the complexity of each training step. As the window size

grows, the number of potential context words also increases, which results in additional matrix operations and parameter updates, thus lengthening the overall training process.

6.10.3 Reasoning Behind the Time Differences

The increase in training time as the window size grows can be explained by the following factors:

- **Context Complexity:** As the window size increases, the number of context words that the model must consider also grows. This adds more word pairs to be processed in each batch, increasing the number of computations required for the forward pass, the calculation of the loss, and the backward pass during backpropagation.
- **More Parameters to Update:** A larger context window means that more word embeddings are involved in the prediction process, requiring more parameters to be updated during training. This leads to longer gradient computation times and an increase in the time spent optimizing the model.
- **Computational Complexity:** The overall computational complexity of the CBOW model increases as the window size grows. This is because the model must iterate over a larger set of word-context pairs, leading to a higher number of dot products, loss calculations, and parameter updates. The increased number of operations is particularly noticeable with a window size of 5, where training time rises significantly.

6.10.4 Conclusions

The results from the CBOW model show that smaller window sizes tend to result in better performance (higher MRR) and faster training times. A window size of 2 strikes the best balance between model performance and computational efficiency. While increasing the window size allows the model to capture broader semantic relationships, it also introduces more noise and increases the computational load, as evidenced by the longer training times and the dip in MRR for a window size of 3.

The findings suggest that, for tasks where local word context is crucial, smaller window sizes (such as 2) are preferable. On the other hand, if the task benefits from broader contextual information, larger window sizes (such as 5) may still offer a balance between training time and model accuracy, though the added computational complexity must be carefully considered.

6.10.5 Average Time and MRR

To provide a more general perspective on the model's performance and computational efficiency, the average MRR and training time across all window sizes were calculated. The results are as follows:

$$\text{Average MRR: } \frac{0.4923 + 0.2304 + 0.4671}{3} = 0.3966$$

$$\text{Average Training Time: } \frac{230.87 + 260.55 + 415.78}{3} = 302.4 \text{ seconds}$$

Analysis of Averages:

- **Average MRR:** The average MRR of 0.3966 indicates moderate performance across different window sizes. The relatively high performance for window sizes 2 and 5 balanced the drop for window size 3, resulting in a reasonable overall ranking accuracy.
- **Average Training Time:** The average training time of 302.4 seconds reflects the computational burden introduced by larger window sizes. As expected, increasing the window size leads to a corresponding increase in training time due to the greater number of word-context pairs that need to be processed.

This average analysis further emphasizes the trade-off between model performance and computational resources, with smaller window sizes providing better performance in less time, and larger window sizes requiring more time without significant MRR improvements.

7 References

References

[FSE11] Anthony Fader, Stephen Soderland, and Oren Etzioni. "Identifying relations for open information extraction". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. EMNLP '11. Edinburgh, United Kingdom: Association for Computational Linguistics, 2011, pp. 1535–1545. ISBN: 9781937284114.

8 Glossary

Open Information Extraction	Method for extracting structured information from unstructured text, typically focusing on extracting relationships in the form of triples (subject, relation, object) from natural language sentences.
Embedding	A technique to represent words or phrases as vectors in a continuous vector space, capturing semantic meanings and relationships between them.
N-ary Relation	A relation involving more than two entities or arguments, as opposed to binary relations that involve only a subject and object.
POS Tagging	The process of assigning grammatical categories (such as nouns, verbs, adjectives) to individual words in a sentence to understand their roles in the context.
Stemming	Chops off word endings, sometimes creating nonsensical words.
Lemmatization	Takes context into account, finding the true root of a word (e.g., "running" becomes "run").

Table 5: Glossary