Natural Language Processing

Team 40

Word Sense Disambiguation

April 2023

Final Submission

Summary

1 Introduction

Word Sense Disambiguation (WSD) is a crucial task in natural language processing, aimed at automatically determining the meaning of words in context. It involves associating a word with its most appropriate sense from a pre-defined sense inventory, such as WordNet, which is the de-facto inventory for English in WSD.

There are two main approaches to WSD: supervised and knowledge-based. Supervised methods use sense-annotated training data, such as SemCor, to learn the mapping between words and their senses. The most common baseline for supervised methods is the Most Frequent Sense (MFS) heuristic, which selects the most frequent sense for each target word in the training data. Some supervised methods, particularly neural architectures, may also use the SemEval 2007 dataset as a development set. Knowledge-based methods, on the other hand, rely on the properties of lexical resources, such as WordNet or BabelNet, to determine the most appropriate sense for a word in context. The first sense given by the underlying sense inventory (i.e., WordNet 3.0) is often included as a baseline for knowledge-based systems.

The state-of-the-art model for WSD is Contextualized Word Embeddings (ELMo), which is a deep contextualized word representation model trained on large-scale corpora, and it has achieved state-of-the-art performance on multiple WSD benchmark datasets. Another widely used model for WSD is BERT, which is a bidirectional transformer-based language model. However, new models and techniques are constantly being developed, and it is possible that there have been improvements

2 Baseline

2.1 Algorithms

We are using three baseline models , in which two are supervised approaches : K-Nearest Neighbour and Naive Bayes Classifier and one unsupervised approach that is Lesk Algorithm.

1. **KNN :** In the context of WSD, it works by finding the k-nearest labeled instances (neighbors) in a labeled training dataset to a given input word, and then predicting the sense

of the input word based on the most frequent sense among those neighbors. The choice of k, which determines the number of neighbors considered, can have a significant impact on the performance of the algorithm.

2. **Naive Bayes :** Naive Bayes is a probabilistic algorithm based on Bayes' theorem, which calculates the probability of a hypothesis (in this case, the sense of a given word) given the evidence (the context in which the word appears). The algorithm assumes that the features (words in the context) are conditionally independent given the class (sense) of the word, hence the name "naive". In the context of WSD, Naive Bayes works by training a probabilistic model on labeled instances in a training dataset, and then using Bayes' theorem to calculate the probability of each sense given the context of the input word, and choosing the sense with the highest probability.

3. **Lesk Algorithm :**The Lesk algorithm is a knowledge-based algorithm. The algorithm works by using a dictionary to look up the definitions of the candidate senses of a given word, and then selecting the sense whose definition has the most overlap (in terms of shared words) with the context in which the word appears. The algorithm takes advantage of the fact that senses of a word are often defined using words that are also used in other senses of the same word, making it possible to disambiguate the word based on the overlap of those defining words with the context.

2.2 Dataset

2.2.1 Dataset Preparation :

The Brown Corpus is a collection of text samples in American English, compiled by Brown University in the 1960s. It contains 1,014,312 words of running text, sampled from a wide range of sources, including news articles, editorials, interviews, fiction, scientific articles, and academic texts.

The SemCor 3.0 dataset is a sense-tagged subset of the Brown Corpus, where each word in the corpus has been labeled with its sense according to WordNet 2.0. It contains around 235,000 words, and has been widely used as a benchmark dataset for word sense disambiguation tasks. SemCor 3.0 is considered to be a challenging dataset due to its size, diversity of texts and the ambiguity of words in context.

```
<text>
```

```
    <wf id="w2" offset="29" sent="1" para="1">said</wf>
</text>
<terms>
    <term id="t2" pos="VB" lemma="say" type="close">
      <span>
        <target id="w2"/>
      </span>
      <externalReferences>
        <externalRef confidence="1.0" reference="1" resource="WordNet-eng30" reftype="sense_number"/>
        <externalRef confidence="1.0" reference="2:32:00::" resource="WordNet-eng30" reftype="lexical_key"/>
        <externalRef confidence="1.0" resource="WordNet-eng30" reference="eng30-01009240-v" reftype="synset"/>
      </externalReferences>
    </term>
</terms>
```

Listing 1: Dataset

Each .naf file of brown corpus contains complete text in the text, and other information inside terms. We prepare the data in the below format.

```
file , context, context_pos, target_word, gloss, is_proper_gloss, wn_index
```

1. file : The path of the .naf file .
2. context: The context of the word , with 10 words before the target word and 10 words after the target word.
3. context_pos : Same context with parts of speect tag along with it.
4. target_word : The index of target word. e.g w1.
5. gloss : Possible gloss of the word using Wordnet definition.
6. is_proper_gloss: Boolean value to show if gloss is the proper definition according to the context.
7. wn_index:A unique identifier for the sense of the word in WordNet, represented as a string in the form ‘lex_name%ss_type:lex_filenum:lex_id::. For example, the sense identifier for the word ”person” in the context ”a person who plays a musical instrument” is

person%1:03:00::.

2.3 Approach

2.3.1 KNN

In the implementation, BERT embeddings are used to create embeddings of the context of the sentences . The gloss of the word can also be used to concatenate it with the context. These embeddings are used as features for the algorithm where we find the cosine similarity between test embeddings for which wn_index is same for each train embeddings and store them in a list and then check whether any of the top-5 mathes with the gloss of the proper sense of the word.

```
for i,test_embedding in enumerate(test_embeddings):
    test_target_word = XTest.iloc[i]['wn_index']
    train_rows = XTrain[XTrain['wn_index'] == test_target_word ]
    for j,train_row in train_rows.iterrows():
        train_embedding = embeddings[j]
        similarity = np.dot(test_embedding, train_embedding) / (np.linalg.norm(
    test_embedding) * np.linalg.norm(train_embedding))
        cosine_similarities.append(similarity)
    if len(cosine_similarities) == 0:
        zero_cos += 1
    if len(cosine_similarities) > 0 :
        XTest.at[i, 'cosine_similarity'] = max(cosine_similarities)
    else:
        XTest.at[i, 'cosine_similarity'] = -1
    cosine_similarities = []


XTest['correct'] = XTest.apply(lambda row: row['gloss'] in XTrain[XTrain['
    wn_index'] == row['wn_index']].head(5)['gloss'].tolist(), axis=1)
```

Listing 2: KNN

2.3.2 Naive Bayes

For each sentence and target word in the training data, BERT embeddings are computed. The embeddings are computed by encoding the context and target word using the tokenizer and passing the resulting token IDs to the BERT model. The output of the model is the contextualized embeddings of each token in the input. The embeddings of the target word and

the surrounding context words are concatenated to form a single embedding. For each sense in the training data, the prior probability is computed as the ratio of the number of instances of that sense to the total number of instances in the training data. The likelihoods are computed as the average of the BERT embeddings for all instances of that sense. For each instance in the test data, the likelihood of the instance belonging to each sense is computed using the Naive Bayes formula. The sense with the highest likelihood is predicted as the sense of the instance. The accuracy is computed as the ratio of the number of correctly predicted senses to the total number of instances in the test data.

```
for sense in set(y_train):
    sense_count = sum(y_train == sense)
    prior_probs[sense] = sense_count / len(y_train)
    sense_embeddings = [x_train[i] for i in range(len(x_train)) if y_train[i]
    == sense]
    likelihoods[sense] = np.mean(sense_embeddings, axis=0)

y_pred = []
for i in range(0,len(y_test)):
    max_prob = 0
    max_sense = None
    for sense in set(y_train):
        log_prob = prior_probs[sense]
        for j in range(len(x_test[i])):
            log_prob = log_prob + x_test[i][j] * likelihoods[sense][j]
        if log_prob > max_prob:
            max_prob = log_prob
            max_sense = sense
    y_pred.append(max_sense)
accuracy = sum(y_pred[i] == y_test.tolist()[i]
               for i in range(len(y_test)))/len(y_test)
```

Listing 3: Naive Bayes

2.3.3 Lesk Algorithm

There are two methods, one is simple lesk which use context of the sentence and ambiguous word . The function tokenise the sentence and iterates over all the synsets of the ambiguous word in WordnNet. For each synset, the function creates a signature set which is the et of words

that occur in the deginition,lemmans and examples of the synset. It then calculate the overlap between the context word and signature set, and find the maximum overlap.

```
def simple_lesk(cotext_sentence,amb_word):
    max_overlap = 0
    lesk_sense = None
    context_words = nltk.word_tokenize(cotext_sentence)
    context_words = set(context_words)
    for sense in wn.synsets(amb_word):
        signature = set()
        sene_definitions = nltk.word_tokenize(sense.definition())
        signature = signature.union(set(sene_definitions))
        signature = signature.union(set(sense.lemma_names()))
        for example in sense.examples():
            signature = signature.union(set(example.split()))
        overlap = len(context_words.intersection(signature))
        if overlap > max_overlap:
            lesk_sense = sense
            max_overlap = overlap

    return lesk_sense
```

Listing 4: Simple Lesk

In the extended lesk algorithm, weights are calculated for each token in the context based on its occurrence in the deginition and examples of each ense of the target word. For each token, the function checks if it appears in the definition or examples of each sense of the tagrget word. If it does, the weight of that ttoken is icnreased by the total number of senses of the target word. The function then iteretes over each sense of the target word and calculates the overlap between the sense's definition, examples, and lemma names, and the tokens in the context. It then calculates the final weight of each sense by summing up the log of the weights of each token that overlaps with that sense. Finally, the function returns the sense with the highest weight as the disambiguated sense for the target word.

```
def extended_lesk(context, word):
    context = context.lower()
    word = word.lower()
    context_tokens = tokenize(context, word)
    synsets = wordnet.synsets(word)
```

```
    finWeights = [0] * len(synsets)
    N_t = len(synsets)
    weights= {}
    for context_token in context_tokens :
        weights[context_token] = 1
        for sense in synsets:
            if context_token in sense.definition():
                weights[context_token] += N_t
                continue
            for example in sense.examples():
                if context_token in example:
                    weights[context_token] += N_t
                    break
            for lemma in sense.lemma_names():
                if context_token in lemma:
                    weights[context_token] += N_t
                    break
    for ind,sense in enumerate(synsets):
        overlap = set()
        for example in sense.examples():
            for token in tokenize(example, word):
                overlap.add(token)
        for token in tokenize(sense.definition(), word):
            overlap.add(token)
        for token in sense.lemma_names():
            overlap.add(token)
        for token in context_tokens:
            if token in overlap:
                finWeights[ind] += np.log(weights[token] / N_t)
    max_weight = max(finWeights)
    index = finWeights.index(max_weight)
    return synsets[index]
```

Listing 5: Extneded Lesk

2.4 Results

Accuracy			
Algorithms	brown1	brown2	brown1 + brown2
K-Nearest Neighbour	67.132%	67.532%	71.826%
Naive Bayes (context)	51.002%	51.341%	53.112%
Naive Bayes (context + pos)	40.231%	42.541%	44.402%
Simple Lesk	34.358%	34.859%	34.645%
Extended Lesk	51.717%	52.957%	52.485%

3 BiLSTM

3.1 Dataset

The dataset is prepared by converting context words into indices. The senses of the words are gives different indexing with different vocabulary for it. There is a dictionary made to store different senses for the target word(lemma) in the corpus. Words with less than 2 frequency are converted into UNK token and finally padding the sentences. The dataloader has input indices for the context, target sense, and the target word.

3.2 Model

The biLSTMModel class is defined, which inherits from the nn.Module class in PyTorch. The constructor of the class takes several arguments:

- input_size: The size of the input vocabulary.
- hidden_size: The number of hidden units in the LSTM layer.
- sense_vocab: A list of possible sense labels. embedding_dim: The dimensionality of the word embeddings.
- dataset: The dataset object containing the training and test data.
- The constructor initializes several attributes of the class, including the input size, hidden size, sense vocabulary, embedding dimension, embedding layer, LSTM layer, and linear layer. It also initializes the sense-to-index and index-to-target dictionaries.
- The forward method is defined, which takes two arguments:

x: The input sequence of word indices. target_word: The indices of the target words in the input sequence.

- The input sequence is first passed through the embedding layer to get the word embeddings. The dimensions of the embedding tensor are then permuted to match the input format expected by the LSTM layer.
- The embeddings are then passed through the LSTM layer to obtain the output tensor. The LSTM layer is defined as a bidirectional LSTM, which means that it processes the input sequence in both forward and backward directions.
- The final hidden state of the LSTM layer is extracted by concatenating the last hidden states of the forward and backward LSTM layers.
- The concatenated hidden state is passed through the linear layer to obtain the output tensor.
- A loop is then used to apply the softmax function to the sense labels that start with the lemma of the target word. This is done to restrict the output probabilities to only those senses that are relevant to the target word.
- The final output tensor is returned.

```
class biLSTMModel(nn.Module):
def __init__(self,input_size,hidden_size,sense_vocab,embedding_dim,dataset):
    super(biLSTMModel,self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.sense_vocab = sense_vocab
    self.embedding_dim = embedding_dim
    self.dataset = dataset
    self.embedding = nn.Embedding(self.input_size,self.embedding_dim)
    self.lstm = nn.LSTM(self.embedding_dim,self.hidden_size,bidirectional=True)
    self.linear = nn.Linear(self.hidden_size*2,len(self.sense_vocab))
    self.sense2idx = self.dataset.sense2idx
    self.idx2word = idx_to_target

def forward(self,x,target_word):
```

```
        x = self.embedding(x)
        x = x.permute(1,0,2)
        output,(hidden,cell) = self.lstm(x)
        hidden = torch.cat((hidden[-2,:,:],hidden[-1,:,:]),dim=1)
        out = self.linear(hidden)
        for i,target_wo in enumerate(target_word):
            target_wo = idx_to_target[target_wo.item()]
            target_word_sense = lemma_2_sense[target_wo]
            target_word_sense_idx = [self.sense2idx[sense] for sense in
    target_word_sense]
            out[i,target_word_sense_idx] = F.softmax(out[i,
    target_word_sense_idx],dim=0)

        return out
```

Listing 6: BiLSTM model

3.3 Training

The model is trained for 30,000 sentences in the semcor dataset for 10 epochs. For 10 epochs , the losses and accuracy for the training set and validation set are given below:

```
Epoch : 1/10 | Loss : 0.8970 | Accuracy : 0.6899
Epoch : 1/10 | Validation Loss : 0.7616 | Validation Accuracy : 0.7095
Epoch : 2/10 | Loss : 0.7125 | Accuracy : 0.7450
Epoch : 2/10 | Validation Loss : 0.7578 | Validation Accuracy : 0.7108
Epoch : 3/10 | Loss : 0.7048 | Accuracy : 0.7512
Epoch : 3/10 | Validation Loss : 0.7538 | Validation Accuracy : 0.7161
Epoch : 4/10 | Loss : 0.7021 | Accuracy : 0.7534
Epoch : 4/10 | Validation Loss : 0.7518 | Validation Accuracy : 0.7160
Epoch : 5/10 | Loss : 0.7028 | Accuracy : 0.7527
Epoch : 5/10 | Validation Loss : 0.7523 | Validation Accuracy : 0.7150
Epoch : 6/10 | Loss : 0.7012 | Accuracy : 0.7541
Epoch : 6/10 | Validation Loss : 0.7522 | Validation Accuracy : 0.7156
Epoch : 7/10 | Loss : 0.7003 | Accuracy : 0.7546
Epoch : 7/10 | Validation Loss : 0.7550 | Validation Accuracy : 0.7143
Epoch : 8/10 | Loss : 0.6991 | Accuracy : 0.7561
Epoch : 8/10 | Validation Loss : 0.7530 | Validation Accuracy : 0.7135
Epoch : 9/10 | Loss : 0.6992 | Accuracy : 0.7558
Epoch : 9/10 | Validation Loss : 0.7548 | Validation Accuracy : 0.7120
Epoch : 10/10 | Loss : 0.6984 | Accuracy : 0.7566
Epoch : 10/10 | Validation Loss : 0.7549 | Validation Accuracy : 0.7125
```

Figure 1: Training

3.4 Results

Results		
Dataset	Accuracy	Loss
SemEval-2013	64.86%	0.7982
SemEval-2015	57.42%	0.8796
SensEval2	63.54%	0.8299
SensEval3	60.59 %	0.9417
All-merged	61.82%	0.8730

4 BERT Fine Tuned Model

In our work, we are adapting BERT for Word Sense Disambiguation (WSD) using the Gloss Selection Objective and Example Sentences. We are using a similar approach to GlossBERT, where we fine-tune BERT on sequence-pair binary classification tasks. Our training data consists of context-gloss pairs, where each pair contains a sentence with a target word to be disambiguated (context) and a candidate sense definition of the target word (gloss) from a lexical database such as WordNet.

During fine-tuning, we classify each context-gloss pair as either positive or negative depending on whether the sense definition corresponds to the correct sense of the target word in the context. However, instead of treating individual context-gloss pairs as independent training instances, we group related context-gloss pairs as 1 training instance. This allows us to formulate WSD as a ranking/selection problem, where the most probable sense is ranked first. By processing all related candidate senses in one go, our WSD model will be able to learn better discriminating features between positive and negative context-gloss pairs.

4.1 Dataset

The dataset used consists of annotated sentences from SemCor 3.0, which is a part of the Senseval-3 corpus, and sense definitions from WordNet 3.0. SemCor is a large corpus of English text that has been annotated with WordNet senses, and it contains over 226k sense-tagged word occurrences. The corpus covers a diverse range of text genres, including newswire, fiction, and non-fiction. The gold standard annotations for SemCor are provided in the form of .xml files and gold.key.txt file, which contains the mapping between word occurrences and their corresponding

WordNet sense keys.

For testing, datasets from the SemEval-2007, SemEval-2013 and Semeval-2015 shared tasks on Word Sense Disambiguation. These datasets contain sense-annotated instances of target words from various domains, including news, science, and technology. The test sets consist of instances that were not seen during training, allowing for a fair evaluation of the WSD model's performance on unseen data.

4.1.1 Preparation

We create a CSV file containing a dataset for word sense disambiguation. The code reads in an XML file containing sentences and word instances, as well as a gold key file indicating the correct sense for each instance. The resulting dataset is a CSV file with the following columns: 'id', 'sentence', 'sense_keys', 'glosses', and 'target_words'.

The 'id' column is a unique identifier for each word instance, 'sentence' is the sentence containing the word instance with target word enclosed with "[TGT]" , 'sense_keys' contains the sense keys for each possible sense of the target word in the sentence, 'glosses' is a list of glosses (i.e., definitions or examples) for each sense key in the sense_keys column, and 'target_words' contains the index of the correct sense key in the sense_keys column for each instance.

The approach used to create the dataset involves iterating through each sentence in the XML file and extracting the target word instances. For each instance, the corresponding sense keys are obtained using WordNet and glosses are generated for each sense key. The sense keys are then shuffled and a random subset is selected to create a list of sense keys and their corresponding glosses, which are used as the 'sense_keys' and 'glosses' columns in the final dataset. The correct sense key index is determined based on the gold key file and added as the 'target_words' column in the final dataset.

The final dataset looks like :

```
1 uid : d000.s000.t000,
2 sentence: How [TGT] long [TGT] has it been since you reviewed the objectives of
      your benefit and service program ?,
3 sense_keys: "['long%3:00:04::', 'long%3:00:02::', 'long%3:00:05::', 'long
    %5:00:00:abundant:00']",
4 glosses : "['(of speech sounds or syllables) of relatively long duration', '
    primarily temporal sense; being or indicating a relatively great or greater
```

```
    than average duration or passage of time or a duration as specified', '
    holding securities or commodities in expectation of a rise in prices', '
    having or being more than normal or necessary:']",
target_words: "[1]"
```

Listing 7: Dataset

4.1.2 Features Generation

List of records (instances), where each record has a sentence and a list of glosses (definitions) with labels indicating whether each gloss is related to the target word in the sentence. Iterate through each record, and for each record, tokenize the sentence and each gloss using the (BERT) tokenizer. Then combine the sentence and each gloss, truncating the combined sequence if necessary to fit within the maximum sequence length.

Create input features for each truncated sequence by concatenating the tokens for the sentence and the gloss, adding special tokens (such as '[CLS]' and '[SEP]') to mark the beginning and end of the sentence and the gloss, and adding segment IDs to indicate which tokens belong to the sentence and which belong to the gloss. The function also creates a label ID based on whether the gloss is related to the target word in the sentence.

The function returns a list of pairs, where each pair corresponds to a truncated sequence, and consists of an input ID tensor, an input mask tensor, a segment ID tensor, and a label ID tensor. The returned features are then used to train a BERT model for the sentence classification task.

4.2 Model

The **BERT_for_WSD model** architecture has the following layers:

BertModel: This is the base BERT model with pre-trained weights. It consists of 12 transformer layers with attention mechanisms.

Dropout: This layer applies dropout regularization to the input to prevent overfitting.

Linear: This is a fully connected layer with one output neuron that is used for ranking the glosses in order to select the correct sense of the target word.

During the forward pass, the input batch is passed through the BertModel, and the output from the second-to-last transformer layer is extracted (also known as the pooled output). This

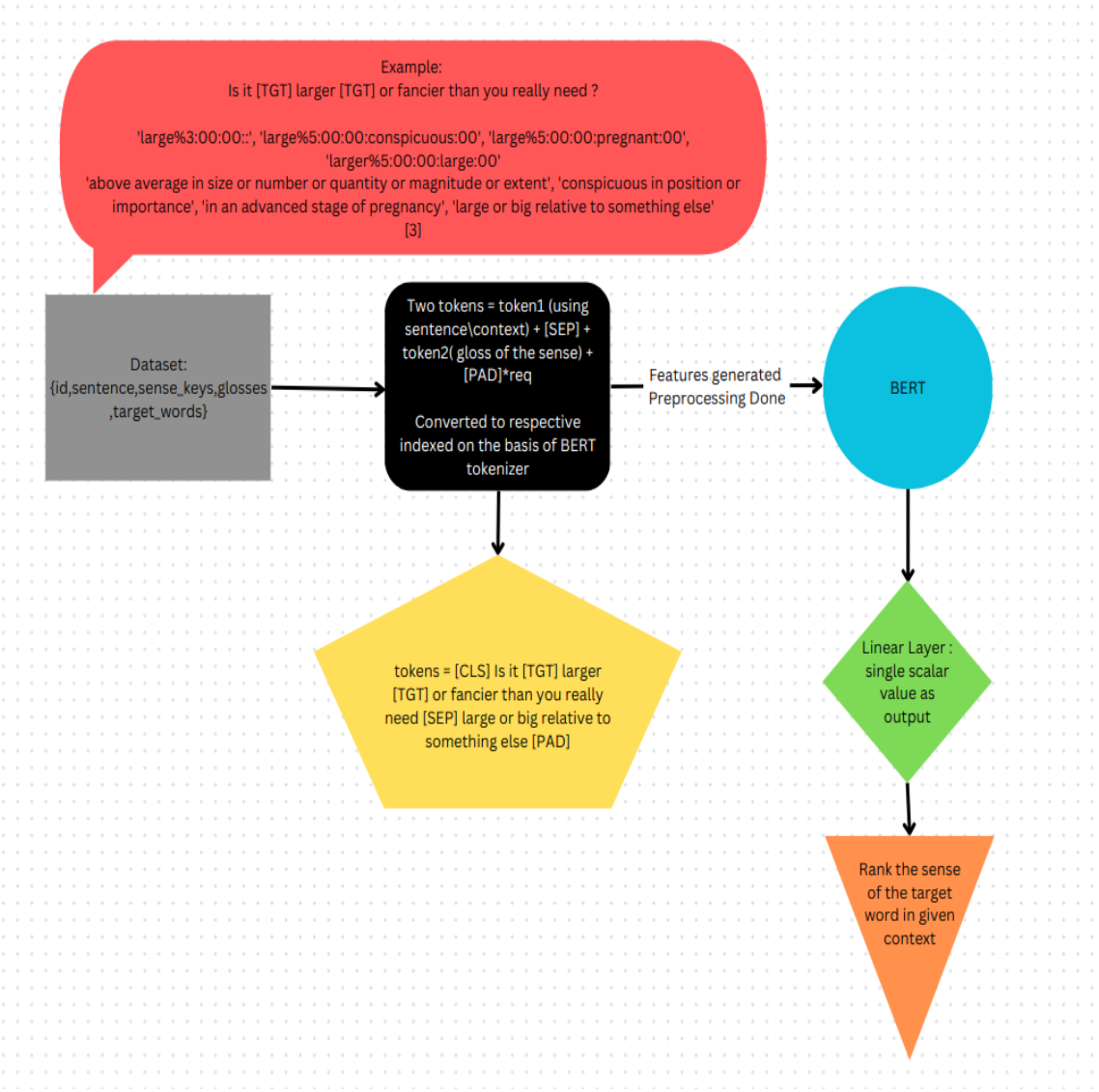

Figure 2: Caption

output is then passed through the Dropout layer, and then through the Linear layer to produce a single output value. This output value is used to rank the glosses in order to select the correct sense of the target word during training.

4.3 Results

We trained our BERT-based model on a dataset of 10,000 records from SemCor 3.0, for a total of 3 epochs. This was due to constraints on both computational resources and time. We then tested our model on multiple datasets, including SemEval-2013, SemEval-2015, SensEval2, SensEval3 as well as a combination of all three.

Results	
Dataset	Accuracy
SemEval-2013	74.8783%
SemEval-2015	78.767%
SensEval2	75.021%
SensEval3	70.054 %
All-merged	73.541%

5 Observation

In non neural models, we used LESK algorithm which is a unsupervised algorithm, whose accuracy is not so good whereas for KNN the accuracy is little high than Naive Bayes (both are supervised). BERT-WSD outperformed all the models and is doing good.

The transformer architecture used in BERT is based on the concept of attention, which allows the model to selectively focus on the relevant parts of the input while ignoring the irrelevant parts. The attention mechanism in BERT is used to compute the importance of each word in the input sentence, based on its relationships with other words in the sentence.

BERT is a bidirectional model, meaning that it can take into account the context of a word both before and after it in a sentence. This is achieved through a process called masked language modeling, where a certain percentage of the input tokens are randomly replaced with a special token [MASK], and the model is trained to predict the original token from the surrounding context.

In addition to masked language modeling, BERT also uses a process called next sentence prediction, where it is trained to predict whether a pair of sentences are consecutive or not. This helps the model capture the relationships between sentences and improve its understanding of the context in which words are used.

Overall, the transformer-based architecture used in BERT, along with its self-supervised pre-training, attention mechanism, and bidirectional modeling, allows it to capture more complex relationships between words and their contexts, making it a highly effective model for tasks like word sense disambiguation.

6 References

- *Adapting BERT for Word Sense Disambiguation with Gloss Selection Objective and Example Sentences -Boon Peng Yap, Andrew Koh, Eng Siong Chng*
- *Rezapour, A., Fakhrahmad, S.M., & Sadreddini, M.H. Applying Weighted KNN to Word Sense Disambiguation*
- *High WSD accuracy using Naive Bayesian classifier with rich features -Cuong Anh Le and Akira Shimazu*