NLP: Word Sense Disambiguation

Nazgul Mamasheva

1 Introduction

Word Sense Disambiguation (WSD) is a natural language processing (NLP) task that aims to determine the correct sense or meaning of a word in a given context. Some words are polysemous, which means they have multiple meanings or senses depending on the context in which they are used. WSD helps in resolving this ambiguity by selecting the most appropriate sense of a word based on the context in which it appears.

There are two levels of granularity in WSD: Coarse-Grained WSD and Fine-Grained WSD. Coarse-Grained WSD aims to classify words into general senses or categories, making it a less detailed but computationally efficient task. Fine-Grained WSD aims to distinguish between more specific word senses, providing more detailed semantic understanding but at a higher complexity.

WSD is essential in various NLP applications, such as machine translation, information retrieval, question answering, sentiment analysis, and more. Resolving word sense ambiguity is critical for these applications to produce accurate and contextually meaningful results.

2 Implementation

As we focus on Coarse-Grained WSD, our approach is providing a solution in a form of Multiclass Token Classification. Multiclass Token Classification refers to the classification of individual words (tokens) into multiple classes or labels. In the case of Coarse-Grained WSD, it involves assigning one of the coarse-grained senses to the corresponding words in a sentence.

2.1 Data processing

In our Coarse-Grained WSD task our main challenge involves dealing with words that have multiple meanings, known as homonyms. To confront this within our Multiclass Token Classification task, we start by creating a list of possible classes, i.e., labels. These labels are retrieved from a mapping of coarse grained candidate and its fine-grained sub-senses. But we can't define all possible labels, since we may encounter new homonymy clusters during testing and they will not be present in our predefined labels vocabulary. For such cases we set a value *0* to the unknown homonymy clusters during homonymy candidate selection.

We are given senses with their corresponding gold homonymy clusters, we treat them as labels. As shown in Figure 1, for each token we assign a label. If the token's instance in the sense, we assign to it a corresponding homonym cluster as a label. If the token's instance is not in the sense, we assign the *"no_homonym"* label to it.

Since model doesn't accept text inputs, we encode our inputs (tokens and labels) to numerical values. For encoding we use HuggingFace's tokenizers, these tokenizers are specifically designed to work with transformer-based models. They break down the text into smaller units or tokens, like words or sub-words, and convert these tokens into unique numerical ids. This allows the models to understand and work with text data. For our task we used HuggingFace's AutoTokenizer, it simplifies the process of working with different pretrained models and ensures that the chosen tokenizer is a suitable match for specific task.

After applying tokenization we get a set of processed tokens, sub-tokens and special tokens. We assign the label only for the first token of each word, for sub-words we assign the label as *-100*. Special tokens are given to us as *None* value and we also assign to them the label as *-100*. The label values that are *-100* automatically ignored in the loss function.

2.2 Model

The structure of our model we can see in Figure 2. The model consists of Transformer Encoder

and Classifier layer. We give tokenized inputs to the Transformer Encoder layer, the outputs from it goes to the Classifier layer as input data. We also compute softmax probabilities on top of classifier layer. Softmax probabilities are commonly used for multiclass classification tasks to make predictions about which class an input belongs to. In order to get softmax probabilities on the output, we apply softmax function on top of classifier's logits output. Thus, the model outputs logits along with the softmax probabilities. During training we deal with logits for computing the loss values and the softmax probabilites we utilize during testing, since we need to output the most probable class for each token in a given input.

For Transformer Encoder we chose a BertModel. BERT (Bidirectional Encoder Representations from Transformers) is pre-trained on large amounts of text data and excels at capturing contextual nuances, making it well-suited for tasks that require a deep understanding of word meanings in context, such as WSD. It can capture patterns and meanings in the text, which is essential for tasks like classifying individual words into different classes. With pre-trained BertModel model we can save a lot of time and effort in training a model from scratch, and it often provides better performance because of its extensive pre-training on a large text corpus. We use pre-trained "bert-base-uncased" BertModel. The "bert-base-uncased" is a well-rounded and a solid starting point for our task.

Since our task is a Multiclass Token Classification, we use a Linear classifier for the Classifier layer thanks to it's simplicity, speed and efficiency.

2.3 Training

In the training phase, we use the Cross-Entropy loss function, which is commonly employed for optimizing classification models. We utilize the Adam optimizer for parameter optimization since it is a widely recognized and effective. On Table 1 we can see the hyper-parameters for the training where the batch size is equal to 32, with training number as 10 and with optimizer choice as Adam optimizer.

For loading the data into training phase we use Dataloader along with it's collate function. The collate function is used to customize samples from a dataset and combine them into batches that can be fed into a model.

The graph of Train and Validating losses are illustrated in Figure 3. As we can see, during first 10 epochs the model has trained faster and better, then it has gradually decreased the learning rate.

3 Results

In order to get better results we utilized the candidates data that we get during testing phase. We retrieve candidate homonyms for each instance, then for each candidate we check if it is in our predefined homonymy labels vocabulary. If candidate homonym isn't in the vocabulary, then we set a value of 0 to it, thus, we don't take unknown homonyms into account, they will not affect the results. Among homonyms candidates that present in label vocabulary we select the most possible one by comparing their softmax probability values, we select the the candidate that has a maximum value. During testing inference we may have scenarios where the prediction value of some homonym label is the highest among all other labels, but this homonym label isn't in candidates list, which means that it's not a correct one certainly. So, selecting a homonym among candidates reduces the possibility of making wrong selections and improves overall results.

As the results shown on Table 2, BertModel gave a better result than a default AutoModel with an accuracy around of 0.95, while AutoModel gave a little less accuracy as around of 0.92.

4 Conclusion

In conclusion, Word Sense Disambiguation (WSD) plays a crucial role in natural language processing by resolving the ambiguity of polysemous words. Our focus on Coarse-Grained WSD, implemented through Multiclass Token Classification, demonstrates the effectiveness of utilizing transformer-based models like BertModel. BertModel captures contextual information and semantic relationships in a given sentence, allowing the model to understand the context in which a word is used. By leveraging pre-trained models and optimizing with techniques such as Cross-Entropy loss and Adam optimizer, we achieved high accuracy in disambiguating word senses. Furthermore, incorporating candidate homonyms during testing improved the robustness of our approach, leading to more accurate results compared to baseline models.

Using Hugging Face Transformer BertModel gives us advantages in our task so that we could achieve an accuracy result up to 95%.

Hyperparameter	Value
Batch size	32
Training number	50
Optimizer	Adam

Table 1: Hyperparameters and their values.

Model	Accuracy
AutoModel	0.92
BertModel	0.9412

Table 2: Models and their accuracy values.

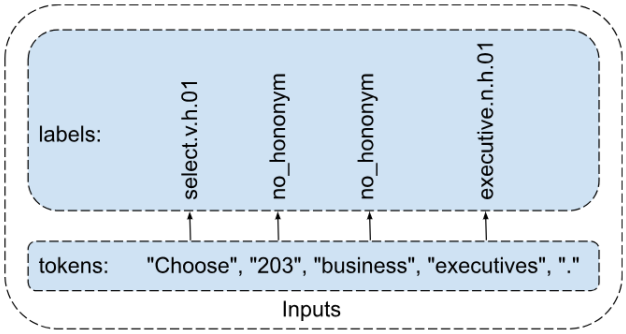

Figure 1: Inputs scheme.

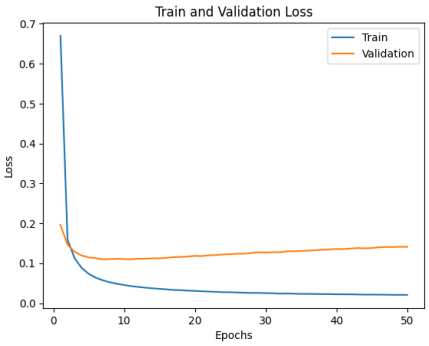

Figure 3: Train and Validation Losses.

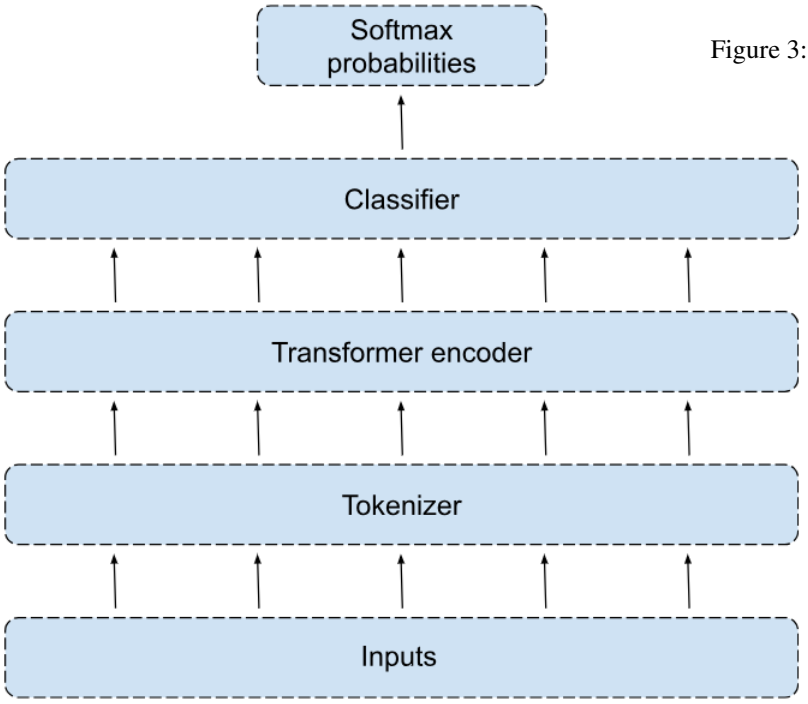

Figure 2: Architecture scheme.