Technische Hochschule Ingolstadt

Fakultät Informatik

Bachelorstudiengang Künstliche Intelligenz

In-depth explanation of POS Tagging with RoBERTa

Seminararbeit

Leong Eu Jinn

Betreuung: Prof. Munir Georges, Mariano Frohnmaier

Datum: June 29, 2023

Abstract

Part-of-speech (POS) tagging plays a crucial role in natural language processing (NLP) tasks, enabling accurate syntactic analysis and facilitating downstream applications. In recent years, transformer-based models have revolutionized the field of NLP, with RoBERTa emerging as a prominent architecture. This seminar paper explores the application of RoBERTa for POS tagging, investigating its performance and analyzing its strengths and weaknesses.

This research contributes to the in-depth understanding of the application of RoBERTa for POS tagging, showcasing its capabilities and highlighting its potential implications in NLP tasks. The findings emphasize RoBERTa's ability to capture complex linguistic patterns.

Contents

1 Introduction

The objective of POS tagging is to assign the most appropriate tag to each word based on its context and grammatical role within the sentenceas showcased in Figure 1. These tags typically represent the word's grammatical category, such as noun, verb, adjective, adverb, or conjunction, among others. By accurately labeling each word, POS tagging facilitates deeper linguistic analysis and enables downstream tasks to better comprehend and interpret textual data [1].

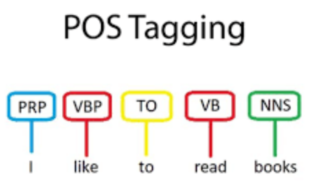


Figure 1: This is an example of POS tagging.

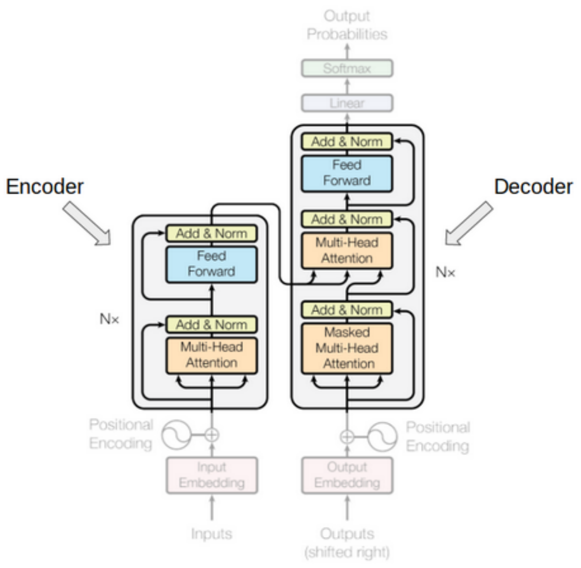
Traditional POS tagging methods relied on linguistic rules, handcrafted feature engineering, and statistical models. These approaches often involved the construction of rule-based systems or the utilization of statistical models. While these methods achieved reasonable accuracy, they were heavily dependent on manual efforts, domain-specific knowledge, and limited coverage of linguistic phenomena.

With the advent of deep learning and the remarkable success of transformer-based models, such as RoBERTa, POS tagging has witnessed a significant advancement. These models utilize neural networks with self-attention mechanisms, allowing them to capture complex contextual information and learn from large-scale corpora without explicit feature engineering [2]. By leveraging vast amounts of unlabeled data during pretraining and fine-tuning on task-specific datasets, transformer-based models have demonstrated superior performance in a wide range of NLP tasks, including POS tagging [3].

2 Transformers

Transformers have revolutionized the field of natural language processing (NLP) with their ability to capture long-range dependencies and contextual information in textual data. Unlike traditional sequential models, transformers employ self-attention mechanisms to efficiently process input sequences in parallel, enabling them to capture global dependencies and overcome the limitations of sequential processing [4].

At the heart of the transformer architecture are self-attention layers. These layers allow the model to assign different weights or attention scores to different words in the input sequence based on their relevance to each other, capturing the contextual relationships between words. By attending to the entire input sequence simultaneously, transformers can capture both local and global dependencies, leading to more robust and accurate representations [5].



The transformer architecture consists of encoder and decoder modules (look at Figure 2). The encoder processes the input sequence, while the decoder generates the output sequence. Each module comprises multiple layers of self-attention and feed-forward neural networks, facilitating the modeling of complex relationships and hierarchical representations.

2.1 Encoder layers

For the purposes of our paper, we are only focused on the encoder layers in the Transformer. As RoBERTa is primarily composed of only these layers.

2.1.1 Input Embeddings

The first step in the encoder layer is to convert the input sequence, which is a sequence of words or tokens, into vector representations called word embeddings. Word embeddings capture the semantic and syntactic information of each word in a lower-dimensional space. These embeddings can be either pre-trained or learned from scratch during the training process [6].

2.1.2 Positional Encodings

Since transformers do not inherently have positional information, positional encoding is added to the word embeddings to convey the order or position of each word in the sequence. This positional encoding is typically a set of fixed vectors that are added element-wise to the word embeddings, allowing the model to distinguish between words based on their positions [4].

The way RoBERTa positional encodes words is to use trigonometric functions.

$$P(k, 2i) = sin(\frac{k}{n^{2i/d}}) \tag{1}$$

$$P(k, 2i+1) = cos(\frac{k}{n^{2i/d}}) \tag{2}$$

where

- d: Dimension of the output embedding space
- k: Position of an object in the input sequence
- n: User-defined scalar, set to 10,000 by the authors of Attention Is All You Need.
- P(k.j): Position function for mapping a position in the input sequence to index of the positional matrix
- i: Used for mapping to column indices, with a single value of maps to both sine and cosine functions

A given worked example would be the phrase “I am a robot,” with n=100 and d=2. The following table 1 shows the positional encoding matrix for this phrase. In fact, the positional encoding matrix would be the same for any four-letter phrase with n=100 and d=2. To further clarify, the code listing 1 is show below.

Table 1: Positional Encoding Matrix for the phrase "I am a robot"

	sequence,k	i=0	i=0
I	k=0	$P_{00} = \sin(0)$	$P_{01} = \cos(0)$
am	k=1	$P_{10} = \sin(1)$	$P_{11} = \cos(\frac{1}{2^0}) = \cos(1)$
a	k=2	$P_{20} = \sin(2)$	$P_{21} = \cos(2)$
robot	k=3	$P_{30} = \cos(3)$	$P_{31} = \cos(3)$

Listing 1: Code for positional embedding

```
import numpy as np
import matplotlib.pyplot as plt

def getPositionEncoding(seq_len, d, n=10000):
    P = np.zeros((seq_len, d))
    for k in range(seq_len):
        for i in np.arange(int(d/2)):
            denominator = np.power(n, 2*i/d)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P
```

2.1.3 Feed Forward

The position-wise feed-forward network (FFN) has a linear layer, ReLU, and another linear layer, which process each embedding vector independently with identical weights [4]. To determine the output of the FFN for the embedding vector,x, is given as following: $Output = ReLU(xW1 + b1)W2 + b2$. To conclude the feed forward network, dropout [7] is also applied after this step.

2.1.4 Layer Normalisation and Residual Connections

The Add and normalisation layer is categorised by the following equation:
$LayerNorm(x+SubLayer(x))$.The sublayer referred is usually the residual connection that is obtained from the previous embeddings to the subsequent layers as can be seen in Figure 2. These sublayers are then added on to the input to retain information from previous layers and then outputted to the subsequent layer. Layer normalisation can be explained

in the following the following equation 3 or code listing 2:

$$x_{norm} = \frac{x - avg(x)}{\sqrt{var(x)} + \mu}[8] \tag{3}$$

where μ is a small constant for numerical stability

Listing 2: Code for positional embedding

```python
import numpy as np

sentence1 = np.array([[0.65,0.14,0.42],[0.98,0.76,0.32]])

def LayerNorm(sentence):
    average = sentence.mean()
    variance = sentence.var()
    return (sentence1 - average1)/(np.sqrt(variance1))
```

2.2 Self Attention Mechanism

The self-attention mechanism is a key component of transformer models that allows them to capture dependencies and relationships between words or tokens in an input sequence. It enables the model to attend to different parts of the sequence while considering the relevance or importance of each token with respect to others. The self-attention mechanism forms the foundation of the transformer architecture, providing a powerful mechanism for learning contextual representations. This paper will be very shallow in the explanation of the Self Attention Mechanism and recommend readers to read the following paper: Attention is all you need [4].

3 RoBERTa (and comparisons to BERT)

RoBERTa, short for "A Robustly Optimized BERT Pretraining Approach," is a transformer-based language model released in July 2019 by Facebook AI and builds upon the success of BERT (Bidirectional Encoder Representations from Transformers). RoBERTa introduces various modifications to the original BERT architecture, training methodology, and data setup, resulting in improved performance on a wide range of natural language processing (NLP) tasks [9].

3.1 Transformer Encoder Layers

RoBERTa consists of a stack of transformer encoder layers, each identical in structure [10]. The inputs to each transformer layer can be processed in parallel, as the computations within each layer are independent of one another. This layer-wise parallelization allows for efficient processing of the entire sequence through multiple layers simultaneously.

3.2 Pre-training

Although RoBERTa uses fundamentally the same architecture. The pre-training in RoBERTa is what distinguishes it from BERT. Roberta was trained on a much larger corpus of text data compared to BERT. It was trained on 160 GB of publicly available text from sources like BooksCorpus and English Wikipedia. BERT is optimized with Adam using the following parameters: $\beta 1 = 0.9$, $\beta 2 = 0.98$, $\epsilon = 1\text{e-}6$ and L2 weight decay of 0.01. The learning rate is warmed up over the first 10,000 steps to a peak value of 1e-4, and then linearly decayed. BERT trains with a dropout of 0.1 on all layers and attention weights, and a GELU activation function instead of the ReLU function in the transformer model. Models are pretrained for S = 1,000,000 updates, with minibatches containing B = 256 sequences of maximum length T = 512 tokens. RoBERTa was trained for a longer duration compared to BERT. RoBERTa was trained for 10 iterations, each consisting of 100,000 training steps, which amounts to a total of 1 million training steps. In conclusion, RoBERTa utilized larger batch sizes, more training data, better strategy and more training steps compared to BERT and that is the key takeaway from RoBERTa [9].

3.2.1 Masked Language Model (MLM)

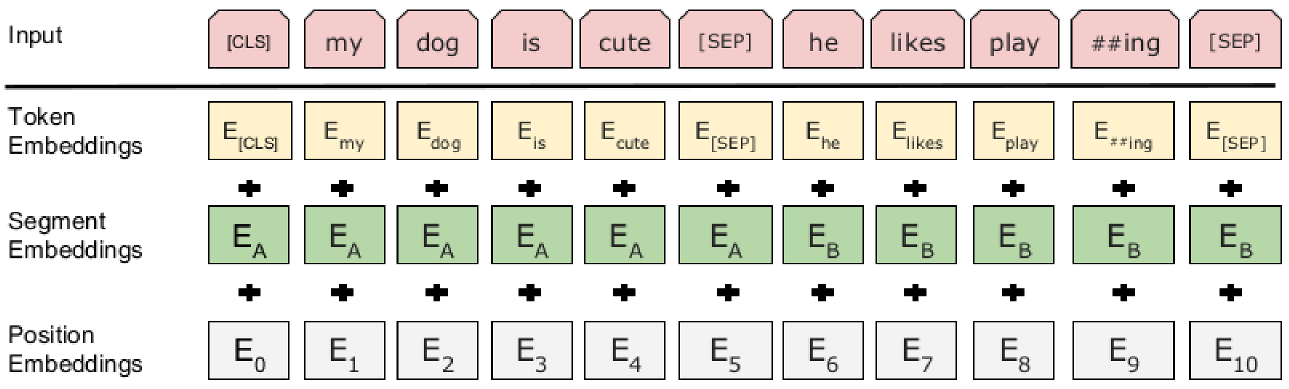
RoBERTa employs only a masked language modeling (MLM) objective, where a fraction of the input tokens are randomly masked, and the model is trained to predict the original masked tokens.It employed dynamic masking, which means that during each training epoch, different parts of the input text are randomly masked. Dynamic masking allows for increased diversity in training and helps the model to learn contextual representations of language from various perspectives.

4 POS Tagging with RoBERTa

To fine-tune RoBERTa for the POS tagging objective, we utilise a technique called Transfer Learning [11]. Transfer Learning leverages the pre-trained knowledge of RoBERTa on a large corpus to initialize the model's parameters, which captures general language patterns. By starting with this strong foundation, we can then fine-tune the model on a smaller, task-specific labeled dataset for POS tagging.

4.1 Methodology

The input text to RoBERTa must be first split into words(tokens). Each set or paragraph of sentences that belong together are part of one sequence which begins with a [CLS] token. The sentence endings are marked by a [SEP] token. The following figure 4.1 shows an example I/O representation:



In order to Fine-tune the model, the dataset used must first be compatible with roberta's word piece tokenization. RoBERTa's word piece tokenization involves splitting words into subword units using the WordPiece algorithm, which allows for better handling of out-of-vocabulary words [12]. An example of how the word piece tokenizer works would be splitting the token "sheepmeat" into "sheep" and "##meat".

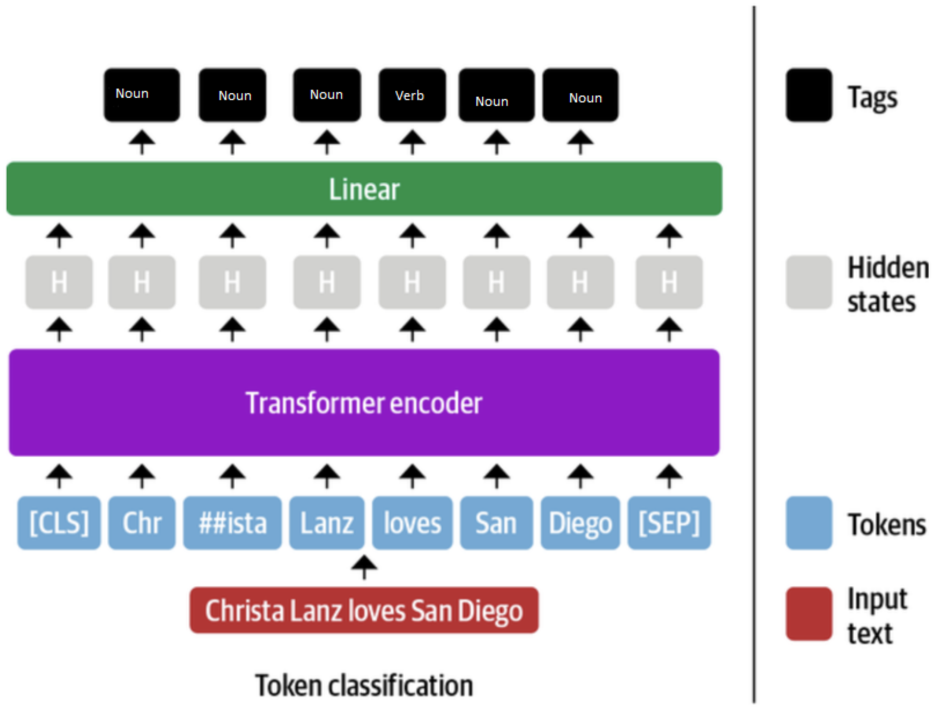
Subsequently, the dataset-labels and roberta-tokens must be aligned. Roberta-tokens that originate from the same dataset-token have the same word index after passing through the word piece tokenizer and so will share the same dataset-label.2.

In addition, padding is applied to the sentences using special tokens, ensuring that all sentences have the same length and are uniform [[13]. This is important for efficient batch processing during training.

Table 2: Alignment of labels and sub-tokens

DT	DT	NNS	VBD	IN	JJ	JJ	JJ
All	The	Contracts	were	for	computer	-	system

Before we pass the POSdataset to our model, we call upon a pre-trained model checkpoint of RoBERTa using the Huggingface modules [14] so we do not need to train it from scratch and then to put a classification head at the top of our RoBERTa model and now our model looks like in Figure 4.1.



The pseudo code so far before passing it onto the training and evaluation loop should look something like in listing 3.

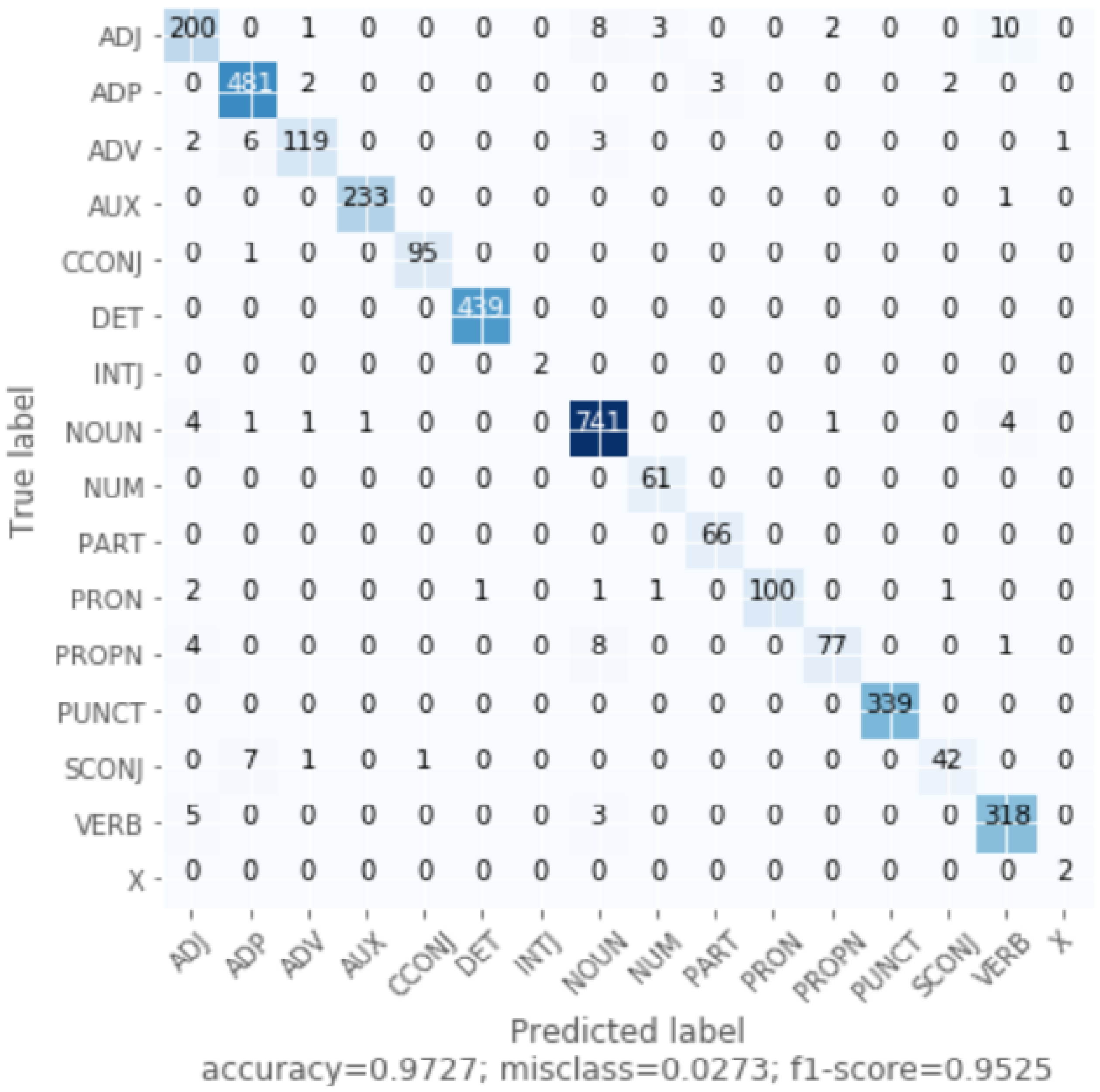
Listing 3: Code for positional embedding

```
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import (AutoTokenizer, TFAutoModelForTokenClassification,
    AdamW,RobertaTokenizerFast
    DataCollatorForTokenClassification,TFRobertaForTokenClassification,
    RobertaForTokenClassification)
import nltk
from sklearn.model_selection import train_test_split

data= nltk.corpus.treebank.tagged_sents()
training_data, testing_data = train_test_split(data, test_size=0.2,
    random_state=25)
x_train = list(map(lambda x: [i[0] for i in x],training_data))
x_test = list(map(lambda x: [i[0] for i in x],testing_data))
y= list(map(lambda x: [i[1] for i in x],data))
label_list = set([item for sublist in y for item in sublist])
id2label = {i: label for i, label in enumerate(label_list)}
label2id = {label: i for i, label in enumerate(label_list)}
y_test = list(map(lambda x: [label2id[i[1]] for i in x],testing_data))
y_train = list(map(lambda x: [label2id[i[1]] for i in x],training_data)

class POSDataset(dataset):
def __init__(self,x,y, tokenizer):
        self.tokenizer = tokenizer
        self.sentences = x
        self.labels = y
        self.pos_labels = self.get_pos_labels()
        self.encoded_inputs = self.tokenize_dataset()

    def tokenize_dataset(self):
        tokenized_inputs = tokenizer(self.sentences, padding=True,
    is_split_into_words=True)
        labels = []
        for i, label in enumerate(self.labels):
            word_ids = tokenized_inputs.word_ids(batch_index=i)
            previous_word_idx = None
            label_ids = []
            for word_idx in word_ids:
                # Special tokens have a word id that is None. We set the
    label to -100 so they are automatically
                # ignored in the loss function.
                if word_idx is None:
                    label_ids.append(-100)
```

```python
                # We set the label for the first token of each word.
                elif word_idx != previous_word_idx:
                    label_ids.append(label[word_idx])
                # For the other tokens in a word, we set the label to either the current label or -100, depending on
                # the label_all_tokens flag.
                else:
                    label_ids.append(label[word_idx])
                previous_word_idx = word_idx
            labels.append(label_ids)
        tokenized_inputs["label_ids"] = labels
        return tokenized_inputs

    def get_pos_labels(self):
        unique_labels = set()
        for labels in self.labels:
            unique_labels.update(labels)
        return list(unique_labels)

    def __len__(self):
        return len(self.sentences)

    def __getitem__(self, idx):
        return {
            "input_ids": self.encoded_inputs["input_ids"][idx],
            "attention_mask": self.encoded_inputs["attention_mask"][idx],
            "label_ids": self.encoded_inputs["label_ids"][idx]
            #,'test': self.tokenizer.convert_ids_to_tokens(self.encoded_inputs["input_ids"][idx])
        }

model_checkpoint = "roberta-base"
batch_size = 16
model = RobertaForTokenClassification.from_pretrained(model_checkpoint, num_labels=len(label_list), id2label=id2label, label2id=label2id)
tokenizer = RobertaTokenizerFast.from_pretrained(model_checkpoint, add_prefix_space=True)
test_ds = POS_dataset(x_train,y_train,tokenizer)
train_ds = POS_dataset(x_test,y_test,tokenizer)
```

4.1.1 Confusion Matrix

After passing our dataset and model onto a standard training and evaluation loop. This is the CM that's obtained.



	ADJ	ADP	ADV	AUX	CCONJ	DET	INTJ	NOUN	NUM	PART	PRON	PROPN	PUNCT	SCONJ	VERB	X
ADJ	200	0	1	0	0	0	0	8	3	0	0	2	0	0	10	0
ADP	0	481	2	0	0	0	0	0	0	3	0	0	0	2	0	0
ADV	2	6	119	0	0	0	0	3	0	0	0	0	0	0	0	1
AUX	0	0	0	233	0	0	0	0	0	0	0	0	0	0	1	0
CCONJ	0	1	0	0	95	0	0	0	0	0	0	0	0	0	0	0
DET	0	0	0	0	0	439	0	0	0	0	0	0	0	0	0	0
INTJ	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
NOUN	4	1	1	1	0	0	0	741	0	0	0	1	0	0	4	0
NUM	0	0	0	0	0	0	0	0	61	0	0	0	0	0	0	0
PART	0	0	0	0	0	0	0	0	0	66	0	0	0	0	0	0
PRON	2	0	0	0	0	1	0	1	1	0	100	0	0	1	0	0
PROPN	4	0	0	0	0	0	0	8	0	0	0	77	0	0	1	0
PUNCT	0	0	0	0	0	0	0	0	0	0	0	0	339	0	0	0
SCONJ	0	7	1	0	1	0	0	0	0	0	0	0	0	42	0	0
VERB	5	0	0	0	0	0	0	3	0	0	0	0	0	0	318	0
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2

References

[1] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 282–289.

[2] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, “Natural language processing (almost) from scratch,” *CoRR*, vol. abs/1103.0398, 2011. [Online]. Available: http://arxiv.org/abs/1103.0398

[3] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *CoRR*, vol. abs/1706.03762, 2017.

[5] S. Latif, A. Zaidi, H. Cuayahuitl, F. Shamshad, M. Shoukat, and J. Qadir, “Transformers in speech processing: A survey,” 2023.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, jan 2014.

[8] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016.

[9] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” 2019.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.

[11] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, "Transfer learning in natural language processing," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 15–18. [Online]. Available: https://aclanthology.org/N19-5004

[12] S. Chi, X. Qiu, Y. Xu, and X. Huang, *How to Fine-Tune BERT for Text Classification?*, 10 2019, pp. 194–206.

[13] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[14] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," 2020.