OmniEvent

Release 0.1

THU-KEG

Aug 24, 2023

TUTORIALS


OmniEvent
A comprehensive, unified and modular
event extraction toolkit.

CHAPTER

ONE

OVERVIEW

OmniEvent is a powerful open-source toolkit for **event extraction**, including **event detection** and **event argument extraction**. We comprehensively cover various paradigms and provide fair and unified evaluations on widely-used **English** and **Chinese** datasets. Modular implementations make OmniEvent highly extensible.

1.1 Highlights

- **Comprehensive Capability**
  - Support to do **Event Extraction** at once, and also to independently do its two subtasks: **Event Detection**, **Event Argument Extraction**.
  - Cover various paradigms: **Token Classification**, **Sequence Labeling**, **MRC (QA)** and **Seq2Seq**, are deployed.
  - Implement **Transformers-based** (BERT, T5, etc.) and **classical** models (CNN, LSTM, CRF, etc.) are implemented.
  - Both **Chinese** and **English** are supported for all event extraction sub-tasks, paradigms and models.
- **Modular Implementation**
  - **All models are decomposed into four modules:**
    - **Input Engineering:** Prepare inputs and support various input engineering methods like prompting.
    - **Backbone:** Encode text into hidden states.
    - **Aggregation:** Fuse hidden states (e.g., select [CLS], pooling, GCN) to the final event representation.
    - **Output Head:** Map the event representation to the final outputs, such as Linear, CRF, MRC head, etc.
- **Unified Benchmark & Evaluation**
  - Various datasets are processed into a unified format.
  - Predictions of different paradigms are all converted into a unified candidate set for fair evaluations.
  - Four evaluation modes (**gold**, **loose**, **default**, **strict**) well cover different previous evaluation settings.
- **Big Model Training & Inference**
  - Efficient training and inference of big models for event extraction are supported with BMTrain.
- **Easy to Use & Highly Extensible**

- Datasets can be downloaded and processed with a single command.
- Fully compatible with Transformers and its Trainer).
- Users can easily reproduce existing models and build customized models with OmniEvent.

CHAPTER

TWO

INSTALLATION

2.1 With pip

This repository is tested on Python 3.9+, Pytorch 1.12.1+. OmniEvent can be installed with pip as follows:

```
pip install OmniEvent
```

CHAPTER

THREE

EASY START

OmniEvent provides ready-to-use models for the users. Examples are shown below.

Make sure you have installed OmniEvent as instructed above. Note that it may take a few minutes to download checkpoint for the first time.

CHAPTER

FOUR

TRAIN YOUR OWN MODEL WITH OMNIEVENT

OmniEvent can help users easily train and evaluate their customized models on a specific dataset.

We show a step-by-step example of using OmniEvent to train and evaluate an **Event Detection** model on **ACE-EN** dataset in the **Seq2Seq** paradigm. More examples are shown in examples.

4.1 Step 1: Process the dataset into the unified format

We provide standard data processing scripts for commonly-adopted datasets. Checkout the details in scripts/data_processing.

```
dataset=ace2005-en  # the dataset name
cd scripts/data_processing/$dataset
bash run.sh
```

4.2 Step 2: Set up the customized configurations

We keep track of the configurations of dataset, model and training parameters via a single `*.yaml` file. See /configs for details.

```
>>> from OmniEvent.arguments import DataArguments, ModelArguments, TrainingArguments,␣
↪ArgumentParser
>>> from OmniEvent.input_engineering.seq2seq_processor import type_start, type_end

>>> parser = ArgumentParser((ModelArguments, DataArguments, TrainingArguments))
>>> model_args, data_args, training_args = parser.parse_yaml_file(yaml_file="config/all-
↪datasets/ed/s2s/ace-en.yaml")

>>> training_args.output_dir = 'output/ACE2005-EN/ED/seq2seq/t5-base/'
>>> data_args.markers = ["<event>", "</event>", type_start, type_end]
```

4.3 Step 3: Initialize the model and tokenizer

OmniEvent supports various backbones. The users can specify the model and tokenizer in the config file and initialize them as follows.

```
>>> from OmniEvent.backbone.backbone import get_backbone
>>> from OmniEvent.model.model import get_model

>>> backbone, tokenizer, config = get_backbone(model_type=model_args.model_type,
                                               model_name_or_path=model_args.model_name_
↪or_path,
                                               tokenizer_name=model_args.model_name_or_
↪path,
                                               markers=data_args.markers,
                                               new_tokens=data_args.markers)
>>> model = get_model(model_args, backbone)
```

4.4 Step 4: Initialize dataset and evaluation metric

OmniEvent prepares the `DataProcessor` and the corresponding evaluation metrics for different task and paradigms.

Note: **Note that** the metrics here are paradigm-dependent and are **not** used for the final unified evaluation.

```
>>> from OmniEvent.input_engineering.seq2seq_processor import EDSeq2SeqProcessor
>>> from OmniEvent.evaluation.metric import compute_seq_F1

>>> train_dataset = EDSeq2SeqProcessor(data_args, tokenizer, data_args.train_file)
>>> eval_dataset = EDSeq2SeqProcessor(data_args, tokenizer, data_args.validation_file)
>>> metric_fn = compute_seq_F1
```

4.5 Step 5: Define Trainer and train

OmniEvent adopts Trainer from Transformers) for training and evaluation.

```
>>> from OmniEvent.trainer_seq2seq import Seq2SeqTrainer

>>> trainer = Seq2SeqTrainer(
        args=training_args,
        model=model,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        compute_metrics=metric_fn,
        data_collator=train_dataset.collate_fn,
        tokenizer=tokenizer,
    )
>>> trainer.train()
```

4.6 Step 6: Unified Evaluation

Since the metrics in Step 4 depend on the paradigm, it is not fair to directly compare the performance of different paradigms.

OmniEvent evaluates models of different paradigms in a unifed manner, where the predictions of different models are converted to word-level and then evaluated.

```
>>> from OmniEvent.evaluation.utils import predict, get_pred_s2s
>>> from OmniEvent.evaluation.convert_format import get_trigger_detection_s2s

>>> logits, labels, metrics, test_dataset = predict(trainer=trainer, tokenizer=tokenizer,
↪ data_class=data_class,
                                                    data_args=data_args, data_file=data_
↪args.test_file,
                                                    training_args=training_args)
>>> # paradigm-dependent metrics
>>> print("{} test performance before converting: {}".formate(test_dataset.dataset_name,␣
↪metrics["test_micro_f1"]))
ACE2005-EN test performance before converting: 66.4215686224377

>>> preds = get_pred_s2s(logits, tokenizer)
>>> # convert to the unified prediction and evaluate
>>> pred_labels = get_trigger_detection_s2s(preds, labels, data_args.test_file, data_
↪args, None)
ACE2005-EN test performance after converting: 67.41016109045849
```

For those datasets whose test set annotations are not given, such as MAVEN and LEVEN, OmniEvent provide APIs to generate submission files. See dump_result.py) for details.

CHAPTER

FIVE

SUPPORTED DATASETS & MODELS & CONTESTS

Continually updated. Welcome to add more!

5.1 Datasets

Language	Domain	Task	Dataset
English	General	ED	MAVEN
English	General	ED EAE	ACE-EN
English	General	ED EAE	ACE-DYGIE
English	General	ED EAE	RichERE (KBP + ERE)
Chinese	Legal	ED	LEVEN
Chinese	General	ED EAE	DuEE
Chinese	General	ED EAE	ACE-ZH
Chinese	Financial	ED EAE	FewFC

5.2 Models

- **Paradigm**
  - Token Classification (TC)
  - Sequence Labeling (SL)
  - Sequence to Sequence (Seq2Seq)
  - Machine Reading Comprehension (MRC)
- **Backbone**
  - CNN / LSTM
  - Transformers (BERT, T5, etc.)
- **Aggregation**
  - Select [CLS]
  - Dynamic/Max Pooling
  - Marker
  - GCN
- **Head**

  - Linear / CRF / MRC heads

5.3 Contests

OmniEvent plans to support various event extraction contest. Currently, we support the following contests and the list is continually updated!

- MAVEN Event Detection Challenge
- CAIL 2022: Event Detection Track
- LUGE: Information Extraction Track

CHAPTER

SIX

EXPERIMENTS

We implement and evaluate state-of-the-art methods on some popular benchmarks using OmniEvent. The results of all Event Detection experiments are shown in the table below. The full results can be accessed via the links below.

- Experiments of base models on All ED Benchmarks
- Experiments of base models on All EAE Benchmarks
- Experiments of All ED Models on ACE-EN+
- Experiments of All EAE Models on ACE-EN+

Language	Domain	Benchmark	Paradigm	Dev F1-score		Test F1-score	
				Paradigm-based	Unified	Paradigm-based	Unified
English	General	MAVEN	TC	--	68.80	--	68.64
			SL	66.75	67.90	--	68.64
			S2S	61.23	61.86	--	61.86
	General	ACE-EN	TC	--	80.47	--	74.13
			SL	77.72	79.44	74.86	75.63
			S2S	75.88	76.73	73.09	72.97
	General	ACE-dygie	TC	--	73.61	--	68.63
			SL	71.58	71.75	68.63	68.63
			S2S	71.61	72.08	65.41	65.99
	General	RichERE	TC	--	68.75	--	51.43
			SL	68.46	66.05	50.13	50.77
			S2S	63.21	62.74	50.07	51.35

Chinese	General	ACE-ZH	TC	--	79.76	--	75.77
			SL	75.41	75.88	72.23	75.93
			S2S	69.45	73.17	63.37	71.61
	General	DuEE	TC	--	92.20	--	--
			SL	85.95	89.62	--	--
			S2S	81.61	85.85	--	--
	Legal	LEVEN	TC	--	85.18	--	85.23
			SL	81.09	84.16	--	84.66
			S2S	78.14	81.29	--	81.41
	Financial	FewFC	TC	--	69.28	--	67.15
			SL	71.13	63.75	68.99	62.31
			S2S	69.89	74.46	69.16	71.33

6.1 Convert the Dataset into Unified OmniEvent Format

To simplify subsequent data loading and modeling, we provide pre-processing scripts for commonly-used Event Extraction datasets. Users can download the dataset and convert it to the unified OmniEvent format by configuring the data path defined in the `run.sh` file under the scripts/data_preprocessing folder with the same name as the dataset.

6.1.1 Unified OmniEvent Format

A unified OmniEvent dataset is a JSON Line file with the extension *.unified.jsonl* (such as, `train.unified.jsonl`, `valid.unified.jsonl`, and `test.unified.jsonl`), which is a convenient format for storing structured data that enables processing one record, in one line, at a time. Taking a record from TAC KBP 2016 as an example, a piece of data in the unified OmniEvent format could be demonstrated as follows:

```
{
    "id": "NYT_ENG_20130910.0002-6",
    "text": "In 1997 , Chun was sentenced to life in prison and Roh to 17 years .",
    "events": [{
        "type": "sentence",
        "triggers": [{
            "id": "em-2342",
            "trigger_word": "sentenced",
            "position": [19, 28],
            "arguments": [{
                "role": "defendant",
                "mentions": [{
                    "id": "m-291",
                    "mention": "Chun",
                    "position": [10, 14]}]}, ... ]}, ... ]} ... ],
    "negative_triggers": [{
```

(continues on next page)

(continued from previous page)

```
        "id": 0,
        "trigger_word": "In",
        "position": [0, 2]}, ... ],
    "entities":  [{
        "type": "PER",
        "mentions": [{
            "id": "m-291",
            "mention": "Chun",
            "position": [10, 14]}, ... ]}, ... ]}
```

6.1.2 Supported Datasets

The pre-processing scripts support almost all commonly-used Event Extraction datasets, so as to minimize the data conversion difficulties. Additional pre-processing scripts are still being developed, and you can submit datasets for which you wish us to complete in "Pull requests". Currently, we have developed pre-processing scripts for the following datasets:

- **ACE2005:** ACE2005-EN, ACE2005-DyGIE, ACE2005-OneIE, ACE2005-ZH
- **DuEE:** DuEE1.0, DuEE-fin
- **ERE:** LDC2015E29, LDC2015E68, LDC2015E78
- **FewFC**
- **TAC KBP:** TAC KBP 2014, TAC KBP 2015, TAC KBP 2016, TAC KBP 2017
- **LEVEN**
- **MAVEN**

6.1.3 Dataset Conversion

Step 1: Download the Dataset

The first step of data conversion is to download the proposed dataset from its corresponding website. For example, for the DuEE 1.0 dataset, it could be downloaded from here.

Step 2: Configure the Dataset Path

After downloading the dataset from the Internet, the `run.sh` file under the folder with the same name as the dataset should be configured. For example, for the DuEE 1.0 dataset, the `run.sh` file under the path scripts/data_preprocessing/duee should be configured, in which the `data_dir` path should be the same as the path of placing the downloaded dataset, you can also modify the path of the processed dataset by configuring the `save_dir` path:

```
python duee.py \
    --data_dir ../../../data/original/DuEE1.0 \
    --save_dir ../../../data/processed/DuEE1.0
```

Step 3: Execute the `run.sh` File

After downloading the dataset and configuring the corresponding `run.sh` file, finally, the dataset could finally be converted to the unified OmniEvent format by executing the configured `run.sh` file. For example, for the DuEE1.0 dataset, we could execute the `run.sh` file as follows:

```
bash run.sh
```

6.2 Examples

Note: To make sure you run the lastest versions of example scirpts, you need to install the repository from source as follows:

```
git clone https://github.com/THU-KEG/OmniEvent.git
cd OmniEvent
pip install .
```

6.2.1 BigModel

The `BigModel` directory contains tuning code for large PLMs. The tuning code is supported by BMTrain engine.

6.2.2 ED

The *ED* directory contains examples of event detection.

6.2.3 EAE

The `EAE` directory contains examples of event argument extraction. You can conduct `EAE` independently using golden event triggers or you can use the predictions of `ED` to do event extraction.

6.3 Tuning Large PLMs for Event Extraction

We provide an example script for tuning large pre-trained language models (PLMs) on event extraction tasks. We use BMTrain as the distributed training engine. BMTrain is an efficient large model training toolkit, see BMTrain and ModelCenter for more details. We adapt the code of ModelCenter for event extraction and place the code in *OmniEvent/utils*.

6.3.1 Setup

Install the code in `OmniEvent/utils/ModelCenter`:

```
cd utils/ModelCenter
pip install .
```

6.3.2 Easy Start

Run `bash train.sh` to train MT5-xxl. You can modify the config and the important hyper-parameters are as follows:

```
NNODES # number of nodes
GPUS_PER_NODE # gpus use on one node
model-config # We only support T5 and MT5
```

The original ModelCenter repo doesn't support inference method (i.e. `generate`) for decoder PLMs. We provide `beam_search.py` for inference.

6.4 Tokenizer

```
import collections
import logging
import numpy as np
import os
import pdb

from transformers import PreTrainedTokenizer
from typing import Dict, Iterable, List, Optional, Tuple, Union

logger = logging.getLogger(__name__)
```

6.4.1 load_vocab

Loads a vocabulary file, allocates a unique id for each word within the vocabulary and saves the correspondence between words and ids into a dictionary. Generates and returns word embeddings if it is required.

Args:

- `vocab_file`: The path of the vocabulary file.
- `return_embeddings`: Whether or not to return the word embeddings.

Returns:

- `word_embeddings`: An numpy array represents each word's embedding within the vocabulary, with the size of (number of words) * (embedding dimension). Returns word embeddings if `return_embeddings` is set as `True`.
- `vocab`: A dictionary indicates the unique id of each word within the vocabulary.

```
def load_vocab(vocab_file: str,
               return_embeddings: bool = False) -> Union[Dict[str, int], np.ndarray]:
    """Loads a vocabulary file into a dictionary.
```

(continues on next page)

(continued from previous page)

```
    Loads a vocabulary file, allocates a unique id for each word within the vocabulary␣
↪and saves the correspondence
    between words and ids into a dictionary. Generates and returns word embeddings if it␣
↪is required.

    Args:
        vocab_file (`str`):
            The path of the vocabulary file.
        return_embeddings (`bool`, `optional`, defaults to `False`):
            Whether or not to return the word embeddings.

    Returns:
        word_embeddings (`np.ndarray`):
            An numpy array represents each word's embedding within the vocabulary, with␣
↪the size of (number of words) *
            (embedding dimension). Returns word embeddings if `return_embeddings` is set␣
↪as True.
        vocab (`Dict[str, int]`):
            A dictionary indicates the unique id of each word within the vocabulary.
    """
    vocab = collections.OrderedDict()
    vocab["[PAD]"] = 0
    with open(vocab_file, "r", encoding="utf-8") as reader:
        lines = reader.readlines()
    num_embeddings = len(lines) + 1
    embedding_dim = len(lines[0].split()) - 1
    for index, line in enumerate(lines):
        token = " ".join(line.split()[:-embedding_dim])
        if token in vocab:
            token = f"{token}_{index+1}"
        vocab[token] = index + 1
    if return_embeddings:
        word_embeddings = np.zeros((num_embeddings, embedding_dim), dtype=np.float32)
        for index, line in enumerate(lines):
            embedding = [float(value) for value in line.strip().split()[-embedding_dim:]]
            word_embeddings[index+1] = embedding
        return word_embeddings
    return vocab
```

6.4.2 whitespace_tokenize()

Cleans the whitespace at the beginning and end of the text and splits the text into a list based on whitespaces.

Args: - `tex`: A string representing the input text to be processed.

Returns:

- `tokens`: A list of strings in which each element represents a word within the input text.

```
def whitespace_tokenize(text: str) -> List[str]:
    """Runs basic whitespace cleaning and splitting on a piece of text.
```

(continues on next page)

(continued from previous page)

```
    Cleans the whitespace at the beginning and end of the text and splits the text into␣
↪a list based on whitespaces.

    Args:
        text (`str`):
            A string representing the input text to be processed.

    Returns:
        tokens (`List[str]`):
            A list of strings in which each element represents a word within the input␣
↪text.
    """
    text = text.strip()
    if not text:
        return []
    tokens = text.split()
    return tokens
```

6.4.3 WordLevelTokenizer

This tokenizer inherits from `PreTrainedTokenizer` which contains most of the main methods. Users should refer to this superclass for more information regarding those methods.

Attributes:

- `vocab`: A dictionary indicating the correspondence between words and ids within the vocabulary.
- `ids_to_tokens`: A dictionary indicating the correspondence between ids and words within the vocabulary.
- `whitespace_tokenizer`: A `WhitespaceTokenizer` instance for word piece tokenization.

```
VOCAB_FILES_NAMES = {"vocab_file": "vec.txt"}

PRETRAINED_VOCAB_FILES_MAP = {}

PRETRAINED_POSITIONAL_EMBEDDINGS_SIZES = {}

PRETRAINED_INIT_CONFIGURATION = {}
```

```
class WordLevelTokenizer(PreTrainedTokenizer):
    """Construct a BERT tokenizer. Based on WordPiece.

    This tokenizer inherits from `PreTrainedTokenizer` which contains most of the main␣
↪methods. Users should refer to
    this superclass for more information regarding those methods.

    Attributes:
        vocab (`Dict[str, int]`):
            A dictionary indicating the correspondence between words and ids within the␣
↪vocabulary.
        ids_to_tokens (`Dict[int, str]`):
```

(continues on next page)

(continued from previous page)

```
            A dictionary indicating the correspondence between ids and words within the␣
↪vocabulary.
        whitespace_tokenizer (`WhitespaceTokenizer`):
            A `WhitespaceTokenizer` instance for word piece tokenization.
    """

    vocab_files_names = VOCAB_FILES_NAMES
    pretrained_vocab_files_map = PRETRAINED_VOCAB_FILES_MAP
    pretrained_init_configuration = PRETRAINED_INIT_CONFIGURATION
    max_model_input_sizes = PRETRAINED_POSITIONAL_EMBEDDINGS_SIZES

    def __init__(self,
                 vocab_file: str,
                 do_lower_case: bool = True,
                 never_split: Iterable = None,
                 unk_token: str = "[UNK]",
                 sep_token: str = "[SEP]",
                 pad_token: str = "[PAD]",
                 cls_token: str = "[CLS]",
                 strip_accents: bool = None,
                 model_max_length: int = 512,
                 **kwargs):
        """Construct a WordLevelTokenizer."""
        kwargs["model_max_length"] = model_max_length
        super().__init__(
            do_lower_case=do_lower_case,
            never_split=never_split,
            unk_token=unk_token,
            sep_token=sep_token,
            pad_token=pad_token,
            cls_token=cls_token,
            strip_accents=strip_accents,
            **kwargs,
        )

        if not os.path.isfile(vocab_file):
            raise ValueError(
                f"Can't find a vocabulary file at path '{vocab_file}'. To load the␣
↪vocabulary from a Google pretrained"
                " model use `tokenizer = BertTokenizer.from_pretrained(PRETRAINED_MODEL_
↪NAME)`"
            )
        self.vocab = load_vocab(vocab_file)
        # insert special token
        for token in [unk_token, sep_token, pad_token, cls_token]:
            if token not in self.vocab:
                self.vocab[token] = len(self.vocab)
        self.ids_to_tokens = collections.OrderedDict([(ids, tok) for tok, ids in self.
↪vocab.items()])
        self.whitespace_tokenizer = WhitespaceTokenizer(vocab=self.vocab, do_lower_
↪case=do_lower_case,
                                                        unk_token=self.unk_token)
```

(continues on next page)

(continued from previous page)

```
    @property
    def do_lower_case(self):
        """Returns whether or not to lowercase the input when tokenizing."""
        return self.whitespace_tokenizer.do_lower_case

    @property
    def vocab_size(self):
        """Returns the length of the vocabulary"""
        return len(self.vocab)

    def get_vocab(self):
        """Returns the vocabulary in a dictionary."""
        return dict(self.vocab, **self.added_tokens_encoder)

    def _tokenize(self,
                  text: str):
        """Tokenizes the input text into tokens."""
        if self.do_lower_case:
            text = text.lower()
        split_tokens = self.whitespace_tokenizer.tokenize(text)
        return split_tokens

    def _convert_token_to_id(self,
                             token: str):
        """Converts a token (`str`) in an id using the vocab."""
        return self.vocab.get(token, self.vocab.get(self.unk_token))

    def _convert_id_to_token(self,
                             index: int):
        """Converts an index (`int`) in a token (`str`) using the vocab."""
        return self.ids_to_tokens.get(index, self.unk_token)

    def convert_tokens_to_string(self,
                                 tokens: str):
        """Converts a sequence of tokens (`str`) in a single string."""
        out_string = " ".join(tokens).replace(" ##", "").strip()
        return out_string

    def build_inputs_with_special_tokens(self,
                                         token_ids_0: List[int],
                                         token_ids_1: Optional[List[int]] = None) ->␣
↪List[int]:
        """Builds model inputs from a sequence or a pair of sequence.
        Builds model inputs from a sequence or a pair of sequence for sequence␣
↪classification tasks by concatenating and
        adding special tokens. A BERT sequence has the following format:

        - single sequence: `[CLS] X [SEP]`
        - pair of sequences: `[CLS] A [SEP] B [SEP]`

        Args:
```

(continues on next page)

(continued from previous page)

```
            token_ids_0 (`List[int]`):
                List of ids to which the special tokens will be added.
            token_ids_1 (`List[int]`, `optional`):
                Optional second list of ids for sequence pairs.

        Returns:
            `List[int]`: List of [input ids](../glossary#input-ids) with the appropriate␣
↪special tokens.
        """
        if token_ids_1 is None:
            return [self.cls_token_id] + token_ids_0 + [self.sep_token_id]
        cls = [self.cls_token_id]
        sep = [self.sep_token_id]
        return cls + token_ids_0 + sep + token_ids_1 + sep

    def get_special_tokens_mask(self,
                                token_ids_0: List[int],
                                token_ids_1: Optional[List[int]] = None,
                                already_has_special_tokens: bool = False) -> List[int]:
        """Retrieve sequence ids from a token list that has no special tokens added."""

        if already_has_special_tokens:
            return super().get_special_tokens_mask(
                token_ids_0=token_ids_0, token_ids_1=token_ids_1, already_has_special_
↪tokens=True
            )

        if token_ids_1 is not None:
            return [1] + ([0] * len(token_ids_0)) + [1] + ([0] * len(token_ids_1)) + [1]
        return [1] + ([0] * len(token_ids_0)) + [1]

    def create_token_type_ids_from_sequences(self,
                                             token_ids_0: List[int],
                                             token_ids_1: Optional[List[int]] = None) ->␣
↪List[int]:
        """Create a mask from the two sequences passed to be used in a sequence-pair␣
↪classification task."""
        sep = [self.sep_token_id]
        cls = [self.cls_token_id]
        if token_ids_1 is None:
            return len(cls + token_ids_0 + sep) * [0]
        return len(cls + token_ids_0 + sep) * [0] + len(token_ids_1 + sep) * [1]

    def save_vocabulary(self,
                        save_directory: str,
                        filename_prefix: Optional[str] = None) -> Tuple[str]:
        """Saves the vocabulary (copy original file) and special tokens file to a␣
↪directory."""
        index = 0
        if os.path.isdir(save_directory):
            vocab_file = os.path.join(
                save_directory, (filename_prefix + "-" if filename_prefix else "") +␣
```

(continues on next page)

(continued from previous page)

```
↪VOCAB_FILES_NAMES["vocab_file"]
            )
        else:
            vocab_file = (filename_prefix + "-" if filename_prefix else "") + save_
↪directory
        with open(vocab_file, "w", encoding="utf-8") as writer:
            for token, token_index in sorted(self.vocab.items(), key=lambda kv: kv[1]):
                if index != token_index:
                    logger.warning(
                        f"Saving vocabulary to {vocab_file}: vocabulary indices are not␣
↪consecutive."
                        " Please check that the vocabulary is not corrupted!"
                    )
                    index = token_index
                writer.write(token + "\n")
                index += 1
        return (vocab_file,)
```

6.4.4 WhitespaceTokenizer

Tokenizes a piece of text into its word pieces by matching whether the token is in the vocabulary.

Attributes:

- vocab: A dictionary indicates the correspondence between words and ids within the vocabulary.
- do_lower_case: A boolean variable indicating Whether or not to lowercase the input when tokenizing.
- unk_token: A string representing the unknown token.

```
class WhitespaceTokenizer(object):
    """A tokenizer to conduct word piece tokenization.

    Tokenizes a piece of text into its word pieces by matching whether the token is in␣
↪the vocabulary.

    Attributes:
        vocab (`Dict[str, int]`):
            A dictionary indicates the correspondence between words and ids within the␣
↪vocabulary.
        do_lower_case (`bool`):
            A boolean variable indicating Whether or not to lowercase the input when␣
↪tokenizing.
        unk_token (`str`):
            A string representing the unknown token.
    """

    def __init__(self,
                 vocab: Dict[str, int],
                 do_lower_case: bool,
                 unk_token: str):
        """Constructs a `WhitespaceTokenizer`."""
        self.vocab = vocab
```

(continues on next page)

(continued from previous page)

```
        self.do_lower_case = do_lower_case
        self.unk_token = unk_token

    def tokenize(self,
                 text: str) -> List[str]:
        """Tokenizes a piece of text into its word pieces."""

        output_tokens = []
        for token in whitespace_tokenize(text):
            if token in self.vocab:
                output_tokens.append(token)
            else:
                output_tokens.append(self.unk_token)
        return output_tokens
```

6.5 Whitespace Tokenizer

```
import collections
import os
import pdb
import logging
import numpy as np
from typing import List, Optional, Tuple
from transformers import PreTrainedTokenizer

logger = logging.getLogger(__name__)
```

6.5.1 load_vocab

Loads a vocabulary file into a dictionary.

```
def load_vocab(vocab_file, return_embeddings=False):
    """Loads a vocabulary file into a dictionary."""
    vocab = collections.OrderedDict()
    vocab["[PAD]"] = 0
    with open(vocab_file, "r", encoding="utf-8") as reader:
        lines = reader.readlines()
    num_embeddings = len(lines) + 1
    embedding_dim = len(lines[0].split()) - 1
    for index, line in enumerate(lines):
        token = " ".join(line.split()[:-embedding_dim])
        if token in vocab:
            token = f"{token}_{index+1}"
        vocab[token] = index + 1
    if return_embeddings:
        word_embeddings = np.zeros((num_embeddings, embedding_dim), dtype=np.float32)
        for index, line in enumerate(lines):
            embedding = [float(value) for value in line.strip().split()[-embedding_dim:]]
```

(continues on next page)

(continued from previous page)

```
            word_embeddings[index+1] = embedding
        return word_embeddings
    return vocab
```

6.5.2 whitespace_tokenize

Runs basic whitespace cleaning and splitting on a piece of text.

```
def whitespace_tokenize(text):
    """Runs basic whitespace cleaning and splitting on a piece of text."""
    text = text.strip()
    if not text:
        return []
    tokens = text.split()
    return tokens
```

```
VOCAB_FILES_NAMES = {"vocab_file": "vec.txt"}

PRETRAINED_VOCAB_FILES_MAP = {}

PRETRAINED_POSITIONAL_EMBEDDINGS_SIZES = {}

PRETRAINED_INIT_CONFIGURATION = {}
```

6.5.3 WordLevelTokenizer

Construct a BERT tokenizer. Based on WordPiece.

This tokenizer inherits from `PreTrainedTokenizer` which contains most of the main methods. Users should refer to this superclass for more information regarding those methods.

```
class WordLevelTokenizer(PreTrainedTokenizer):
    r"""
    Construct a BERT tokenizer. Based on WordPiece.
    This tokenizer inherits from [`PreTrainedTokenizer`] which contains most of the main␣
↪methods. Users should refer to
    this superclass for more information regarding those methods.
    Args:
        vocab_file (`str`):
            File containing the vocabulary.
        do_lower_case (`bool`, *optional*, defaults to `True`):
            Whether or not to lowercase the input when tokenizing.
        do_basic_tokenize (`bool`, *optional*, defaults to `True`):
            Whether or not to do basic tokenization before WordPiece.
        never_split (`Iterable`, *optional*):
            Collection of tokens which will never be split during tokenization. Only has␣
↪an effect when
            `do_basic_tokenize=True`
        unk_token (`str`, *optional*, defaults to `"[UNK]"`):
            The unknown token. A token that is not in the vocabulary cannot be converted␣
```

(continues on next page)

(continued from previous page)

```
↪to an ID and is set to be this
            token instead.
        sep_token (`str`, *optional*, defaults to `"[SEP]"`):
            The separator token, which is used when building a sequence from multiple␣
↪sequences, e.g. two sequences for
            sequence classification or for a text and a question for question answering.␣
↪It is also used as the last
            token of a sequence built with special tokens.
        pad_token (`str`, *optional*, defaults to `"[PAD]"`):
            The token used for padding, for example when batching sequences of different␣
↪lengths.
        cls_token (`str`, *optional*, defaults to `"[CLS]"`):
            The classifier token which is used when doing sequence classification␣
↪(classification of the whole sequence
            instead of per-token classification). It is the first token of the sequence␣
↪when built with special tokens.
        mask_token (`str`, *optional*, defaults to `"[MASK]"`):
            The token used for masking values. This is the token used when training this␣
↪model with masked language
            modeling. This is the token which the model will try to predict.
        tokenize_chinese_chars (`bool`, *optional*, defaults to `True`):
            Whether or not to tokenize Chinese characters.
            This should likely be deactivated for Japanese (see this
            [issue](https://github.com/huggingface/transformers/issues/328)).
        strip_accents (`bool`, *optional*):
            Whether or not to strip all accents. If this option is not specified, then␣
↪it will be determined by the
            value for `lowercase` (as in the original BERT).
    """

    vocab_files_names = VOCAB_FILES_NAMES
    pretrained_vocab_files_map = PRETRAINED_VOCAB_FILES_MAP
    pretrained_init_configuration = PRETRAINED_INIT_CONFIGURATION
    max_model_input_sizes = PRETRAINED_POSITIONAL_EMBEDDINGS_SIZES

    def __init__(
        self,
        vocab_file,
        do_lower_case=True,
        never_split=None,
        unk_token="[UNK]",
        sep_token="[SEP]",
        pad_token="[PAD]",
        cls_token="[CLS]",
        strip_accents=None,
        model_max_length=512,
        **kwargs
    ):
        kwargs["model_max_length"] = model_max_length
        super().__init__(
            do_lower_case=do_lower_case,
            never_split=never_split,
```

(continues on next page)

(continued from previous page)

```
            unk_token=unk_token,
            sep_token=sep_token,
            pad_token=pad_token,
            cls_token=cls_token,
            strip_accents=strip_accents,
            **kwargs,
        )

        if not os.path.isfile(vocab_file):
            raise ValueError(
                f"Can't find a vocabulary file at path '{vocab_file}'. To load the␣
↪vocabulary from a Google pretrained"
                " model use `tokenizer = BertTokenizer.from_pretrained(PRETRAINED_MODEL_
↪NAME)`"
            )
        self.vocab = load_vocab(vocab_file)
        # insert special token
        for token in [unk_token, sep_token, pad_token, cls_token]:
            if token not in self.vocab:
                self.vocab[token] = len(self.vocab)
        self.ids_to_tokens = collections.OrderedDict([(ids, tok) for tok, ids in self.
↪vocab.items()])
        self.whitespace_tokenizer = WhitespaceTokenizer(vocab=self.vocab, do_lower_
↪case=do_lower_case, unk_token=self.unk_token)

    @property
    def do_lower_case(self):
        return self.whitespace_tokenizer.do_lower_case

    @property
    def vocab_size(self):
        return len(self.vocab)

    def get_vocab(self):
        return dict(self.vocab, **self.added_tokens_encoder)

    def _tokenize(self, text):
        if self.do_lower_case:
            text = text.lower()
        split_tokens = self.whitespace_tokenizer.tokenize(text)
        return split_tokens

    def _convert_token_to_id(self, token):
        """Converts a token (str) in an id using the vocab."""
        return self.vocab.get(token, self.vocab.get(self.unk_token))

    def _convert_id_to_token(self, index):
        """Converts an index (integer) in a token (str) using the vocab."""
        return self.ids_to_tokens.get(index, self.unk_token)

    def convert_tokens_to_string(self, tokens):
        """Converts a sequence of tokens (string) in a single string."""
```

(continues on next page)

(continued from previous page)

```
        out_string = " ".join(tokens).replace(" ##", "").strip()
        return out_string

    def build_inputs_with_special_tokens(
        self, token_ids_0: List[int], token_ids_1: Optional[List[int]] = None
    ) -> List[int]:
        """
        Build model inputs from a sequence or a pair of sequence for sequence␣
↪classification tasks by concatenating and
        adding special tokens. A BERT sequence has the following format:
        - single sequence: `[CLS] X [SEP]`
        - pair of sequences: `[CLS] A [SEP] B [SEP]`
        Args:
            token_ids_0 (`List[int]`):
                List of IDs to which the special tokens will be added.
            token_ids_1 (`List[int]`, *optional*):
                Optional second list of IDs for sequence pairs.
        Returns:
            `List[int]`: List of [input IDs](../glossary#input-ids) with the appropriate␣
↪special tokens.
        """
        if token_ids_1 is None:
            return [self.cls_token_id] + token_ids_0 + [self.sep_token_id]
        cls = [self.cls_token_id]
        sep = [self.sep_token_id]
        return cls + token_ids_0 + sep + token_ids_1 + sep

    def get_special_tokens_mask(
        self, token_ids_0: List[int], token_ids_1: Optional[List[int]] = None, already_
↪has_special_tokens: bool = False
    ) -> List[int]:
        """
        Retrieve sequence ids from a token list that has no special tokens added. This␣
↪method is called when adding
        special tokens using the tokenizer `prepare_for_model` method.
        Args:
            token_ids_0 (`List[int]`):
                List of IDs.
            token_ids_1 (`List[int]`, *optional*):
                Optional second list of IDs for sequence pairs.
            already_has_special_tokens (`bool`, *optional*, defaults to `False`):
                Whether or not the token list is already formatted with special tokens␣
↪for the model.
        Returns:
            `List[int]`: A list of integers in the range [0, 1]: 1 for a special token, 0␣
↪for a sequence token.
        """

        if already_has_special_tokens:
            return super().get_special_tokens_mask(
                token_ids_0=token_ids_0, token_ids_1=token_ids_1, already_has_special_
↪tokens=True
```

(continues on next page)

(continued from previous page)

```
            )

        if token_ids_1 is not None:
            return [1] + ([0] * len(token_ids_0)) + [1] + ([0] * len(token_ids_1)) + [1]
        return [1] + ([0] * len(token_ids_0)) + [1]

    def create_token_type_ids_from_sequences(
        self, token_ids_0: List[int], token_ids_1: Optional[List[int]] = None
    ) -> List[int]:
        """
        Create a mask from the two sequences passed to be used in a sequence-pair␣
↪classification task. A BERT sequence
        pair mask has the following format:
        ```
        0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
        | first sequence | second sequence |
        ```
        If `token_ids_1` is `None`, this method only returns the first portion of the␣
↪mask (0s).
        Args:
            token_ids_0 (`List[int]`):
                List of IDs.
            token_ids_1 (`List[int]`, *optional*):
                Optional second list of IDs for sequence pairs.
        Returns:
            `List[int]`: List of [token type IDs](../glossary#token-type-ids) according␣
↪to the given sequence(s).
        """
        sep = [self.sep_token_id]
        cls = [self.cls_token_id]
        if token_ids_1 is None:
            return len(cls + token_ids_0 + sep) * [0]
        return len(cls + token_ids_0 + sep) * [0] + len(token_ids_1 + sep) * [1]

    def save_vocabulary(self, save_directory: str, filename_prefix: Optional[str] =␣
↪None) -> Tuple[str]:
        index = 0
        if os.path.isdir(save_directory):
            vocab_file = os.path.join(
                save_directory, (filename_prefix + "-" if filename_prefix else "") +␣
↪VOCAB_FILES_NAMES["vocab_file"]
            )
        else:
            vocab_file = (filename_prefix + "-" if filename_prefix else "") + save_
↪directory
        with open(vocab_file, "w", encoding="utf-8") as writer:
            for token, token_index in sorted(self.vocab.items(), key=lambda kv: kv[1]):
                if index != token_index:
                    logger.warning(
                        f"Saving vocabulary to {vocab_file}: vocabulary indices are not␣
↪consecutive."
                        " Please check that the vocabulary is not corrupted!"
```

(continues on next page)

(continued from previous page)

```
                    )
                    index = token_index
                writer.write(token + "\n")
                index += 1
        return (vocab_file,)
```

6.5.4 WhitespaceTokenizer

Runs WordPiece tokenization.

```
class WhitespaceTokenizer(object):
    """Runs WordPiece tokenization."""

    def __init__(self, vocab, do_lower_case, unk_token):
        self.vocab = vocab
        self.do_lower_case = do_lower_case
        self.unk_token = unk_token

    def tokenize(self, text):
        """
        Tokenizes a piece of text into its word pieces. This uses a greedy longest-match-
↪first algorithm to perform
        tokenization using the given vocabulary.
        For example, `input = "unaffable"` wil return as output `["un", "##aff", "##able
↪"]`.
        Args:
            text: A single token or whitespace separated tokens. This should have
                already been passed through *BasicTokenizer*.
        Returns:
            A list of wordpiece tokens.
        """

        output_tokens = []
        for token in whitespace_tokenize(text):
            if token in self.vocab:
                output_tokens.append(token)
            else:
                output_tokens.append(self.unk_token)
        return output_tokens
```

6.6 Base Processor

```
import os
import json
import torch
import logging

from torch.utils.data import Dataset
```

(continues on next page)

(continued from previous page)

```
from typing import Dict, List, Optional, Union

logger = logging.getLogger(__name__)
```

6.6.1 EDInputExample

A single training/test example for event detection, representing the basic information of an event trigger, including its example id, the source text it is within, its start and end position, and the event type of the trigger.

Attributes:

- `example_id`: A string or an integer for the unique id of the example.
- `text`: A string representing the source text the event trigger is within.
- `trigger_left`: An integer indicating the left position of the event trigger.
- `trigger_right`: An integer indicating the right position of the event trigger.
- `labels`: A string indicating the event type of the trigger.

```
class EDInputExample(object):
    """A single training/test example for event detection.

    A single training/test example for event detection, representing the basic␣
↪information of an event trigger,
    including its example id, the source text it is within, its start and end position,␣
↪and the label of the event.

    Attributes:
        example_id (`Union[int, str]`):
            A string or an integer for the unique id of the example.
        text (`str`):
            A string representing the source text the event trigger is within.
        trigger_left (`int`, `optional`, defaults to `None`):
            An integer indicating the left position of the event trigger.
        trigger_right (`int`, `optional`, defaults to `None`):
            An integer indicating the right position of the event trigger.
        labels (`int`, `optional`, defaults to `None`):
            A string indicating the event type of the trigger.
    """

    def __init__(self,
                 example_id: Union[int, str],
                 text: str,
                 trigger_left: Optional[int] = None,
                 trigger_right: Optional[int] = None,
                 labels: Optional[str] = None) -> None:
        """Constructs an `EDInputExample`."""
        self.example_id = example_id
        self.text = text
        self.trigger_left = trigger_left
        self.trigger_right = trigger_right
        self.labels = labels
```

6.6.2 EDInputFeatures

Input features of an instance for event detection, representing the basic features of an event trigger, including its example id, the indices of tokens in the vocabulary, attention masks, segment token indices, start and end position, and the event type of the trigger.

Attributes:

- `example_id`: A string or an integer for the unique id of the example.
- `input_ids`: A list of integers representing the indices of input sequence tokens in the vocabulary.
- `attention_mask`: A list of integers (in 0/1) for masks to avoid attention on padding tokens.
- `token_type_ids`: A list of integers indicating the first and second portions of the inputs.
- `trigger_left`: An integer indicating the left position of the event trigger.
- `trigger_right`: An integer indicating the right position of the event trigger.
- `labels`: A string indicating the event type of the trigger.

```
class EDInputFeatures(object):
    """Input features of an instance for event detection.

    Input features of an instance for event detection, representing the basic features␣
↪of an event trigger, including
    its example id, the indices of tokens in the vocabulary, attention masks, segment␣
↪token indices, start and end
    position, and the label of the event.

    Attributes:
        example_id (`Union[int, str]`):
            A string or an integer for the unique id of the example.
        input_ids (`List[int]`):
            A list of integers representing the indices of input sequence tokens in the␣
↪vocabulary.
        attention_mask (`List[int]`):
            A list of integers (in 0/1) for masks to avoid attention on padding tokens.
        token_type_ids (`List[int]`, `optional`, defaults to `None`):
            A list of integers indicating the first and second portions of the inputs.
        trigger_left (`int`, `optional`, defaults to `None`):
            An integer indicating the left position of the event trigger.
        trigger_right (`int`, `optional`, defaults to `None`):
            An integer indicating the right position of the event trigger.
        labels (`str`, `optional`, defaults to `None`):
            A string indicating the event type of the trigger.
    """

    def __init__(self,
                 example_id: Union[int, str],
                 input_ids: List[int],
                 attention_mask: List[int],
                 token_type_ids: Optional[List[int]] = None,
                 trigger_left: Optional[int] = None,
                 trigger_right: Optional[int] = None,
                 labels: Optional[str] = None) -> None:
```

(continues on next page)

(continued from previous page)

```
        """Constructs an `EDInputFeatures`."""
        self.example_id = example_id
        self.input_ids = input_ids
        self.attention_mask = attention_mask
        self.token_type_ids = token_type_ids
        self.trigger_left = trigger_left
        self.trigger_right = trigger_right
        self.labels = labels
```

6.6.3 EAEInputExample

A single training/test example for event argument extraction, representing the basic information of an event trigger, including its example id, the source text it is within, the predicted and actual event type, the input template for the Machine Reading Comprehension (MRC) paradigm, the start and end position of the event trigger and argument, and the label of the event.

Attributes:

- `example_id`: A string or an integer for the unique id of the example.
- `text`: A string representing the source text the event trigger and argument is within.
- `pred_type`: A string indicating the event type predicted by the model.
- `true_type`: A string indicating the actual event type from the annotation.
- `input_template`: The input template for the MRC paradigm.
- `trigger_left`: An integer indicating the left position of the event trigger.
- `trigger_right`: An integer indicating the right position of the event trigger.
- `argument_left`: An integer indicating the left position of the argument mention.
- `argument_right`: An integer indicating the right position of the argument mention.
- `argument_role`: A string indicating the argument role of the argument mention.
- `labels`: A string indicating the label of the event.

```
class EAEInputExample(object):
    """A single training/test example for event argument extraction.

    A single training/test example for event argument extraction, representing the basic␣
↪information of an event
    trigger, including its example id, the source text it is within, the predicted and␣
↪actual event type, the input
    template for the Machine Reading Comprehension (MRC) paradigm, the start and end␣
↪position of the event trigger and
    argument, and the label of the event.

    Attributes:
        example_id (`Union[int, str]`):
            A string or an integer for the unique id of the example.
        text (`str`):
            A string representing the source text the event trigger and argument is␣
↪within.
```

(continues on next page)

(continued from previous page)

```
        pred_type (`str`):
            A string indicating the event type predicted by the model.
        true_type (`str`):
            A string indicating the actual event type from the annotation.
        input_template:
            The input template for the MRC paradigm.
        trigger_left (`int`, `optional`, defaults to `None`):
            An integer indicating the left position of the event trigger.
        trigger_right (`int`, `optional`, defaults to `None`):
            An integer indicating the right position of the event trigger.
        argument_left (`int`, `optional`, defaults to `None`):
            An integer indicating the left position of the argument mention.
        argument_right (`int`, `optional`, defaults to `None`):
            An integer indicating the right position of the argument mention.
        argument_role (`str`, `optional`, defaults to `None`):
            A string indicating the argument role of the argument mention.
        labels (`str`, `optional`, defaults to `None`):
            A string indicating the label of the event.
    """

    def __init__(self,
                 example_id: Union[int, str],
                 text: str,
                 pred_type: str,
                 true_type: str,
                 input_template: Optional = None,
                 trigger_left: Optional[int] = None,
                 trigger_right: Optional[int] = None,
                 argument_left: Optional[int] = None,
                 argument_right: Optional[int] = None,
                 argument_role: Optional[str] = None,
                 labels: Optional[str] = None):
        """Constructs a `EAEInputExample`."""
        self.example_id = example_id
        self.text = text
        self.pred_type = pred_type
        self.true_type = true_type
        self.input_template = input_template
        self.trigger_left = trigger_left
        self.trigger_right = trigger_right
        self.argument_left = argument_left
        self.argument_right = argument_right
        self.argument_role = argument_role
        self.labels = labels
```

6.6.4 EAEInputFeatures

Input features of an instance for event argument extraction, representing the basic features of an argument mention, including its example id, the indices of tokens in the vocabulary, the attention mask, segment token indices, the start and end position of the event trigger and argument mention, and the event type of the trigger.

Attributes:

- `example_id`: A string or an integer for the unique id of the example.
- `input_ids`: A list of integers representing the indices of input sequence tokens in the vocabulary.
- `attention_mask`: A list of integers (in 0/1) for masks to avoid attention on padding tokens.
- `token_type_ids`: A list of integers indicating the first and second portions of the inputs.
- `trigger_left`: An integer for the left position of the event trigger.
- `trigger_right`: An integer for the right position of the event trigger.
- `argument_left`: An integer for the left position of the argument mention.
- `argument_right`: An integer for the right position of the argument mention.
- `labels`: A string indicating the event type of the trigger.

```
class EAEInputFeatures(object):
    """Input features of an instance for event argument extraction.

    Input features of an instance for event argument extraction, representing the basic␣
↪features of an argument mention,
    including its example id, the indices of tokens in the vocabulary, the attention␣
↪mask, segment token indices, the
    start and end position of the event trigger and argument mention, and the label of␣
↪the event.

    Attributes:
        example_id (`Union[int, str]`):
            A string or an integer for the unique id of the example.
        input_ids (`List[int]`):
            A list of integers representing the indices of input sequence tokens in the␣
↪vocabulary.
        attention_mask (`List[int]`):
            A list of integers (in 0/1) for masks to avoid attention on padding tokens.
        token_type_ids (`List[int]`, `optional`, defaults to `None`):
            A list of integers indicating the first and second portions of the inputs.
        trigger_left (`int`, `optional`, defaults to `None`):
            An integer for the left position of the event trigger.
        trigger_right (`int`, `optional`, defaults to `None`):
            An integer for the right position of the event trigger.
        argument_left (`int`, `optional`, defaults to `None`):
            An integer for the left position of the argument mention.
        argument_right (`int`, `optional`, defaults to `None`):
            An integer for the right position of the argument mention.
        labels (`str`, `optional`, defaults to `None`):
            A string indicating the event type of the trigger.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 example_id: Union[int, str],
                 input_ids: List[int],
                 attention_mask: List[int],
                 token_type_ids: Optional[List[int]] = None,
                 trigger_left: Optional[int] = None,
                 trigger_right: Optional[int] = None,
                 argument_left: Optional[int] = None,
                 argument_right: Optional[int] = None,
                 labels: Optional[str] = None) -> None:
        """Constructs an `EAEInputFeatures`."""
        self.example_id = example_id
        self.input_ids = input_ids
        self.attention_mask = attention_mask
        self.token_type_ids = token_type_ids
        self.trigger_left = trigger_left
        self.trigger_right = trigger_right
        self.argument_left = argument_left
        self.argument_right = argument_right
        self.labels = labels
```

6.6.5 EDDataProcessor

The base class of data processor for event detection, which would be inherited to construct task-specific data processors.

Attributes:

- `config`: The pre-defined configurations of the execution.
- `tokenizer`: The tokenizer method proposed for the tokenization process.
- `examples`: A list of ``EDInputExample``s constructed based on the input dataset.
- `input_features`: A list of ``EDInputFeatures``s corresponding to the ``EDInputExample``s.

```
class EDDataProcessor(Dataset):
    """Base class of data processor for event detection.

    The base class of data processor for event detection, which would be inherited to␣
↪construct task-specific data
    processors.

    Attributes:
        config:
            The pre-defined configurations of the execution.
        tokenizer (`str`):
            The tokenizer method proposed for the tokenization process.
        examples (`List[EDInputExample]`):
            A list of `EDInputExample`s constructed based on the input dataset.
        input_features (`List[EDInputFeatures]`):
            A list of `EDInputFeatures`s corresponding to the `EDInputExample`s.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 config,
                 tokenizer) -> None:
        """Constructs an `EDDataProcessor`."""
        self.config = config
        self.tokenizer = tokenizer
        self.examples = []
        self.input_features = []

    def read_examples(self,
                      input_file: str):
        """Obtains a collection of `EDInputExample`s for the dataset."""
        raise NotImplementedError

    def convert_examples_to_features(self):
        """Converts the `EDInputExample`s into `EDInputFeatures`s."""
        raise NotImplementedError

    def _truncate(self,
                  outputs: dict,
                  max_seq_length: int):
        """Truncates the sequence that exceeds the maximum length."""
        is_truncation = False
        if len(outputs["input_ids"]) > max_seq_length:
            print("An instance exceeds the maximum length.")
            is_truncation = True
            for key in ["input_ids", "attention_mask", "token_type_ids", "offset_mapping
↪"]:
                if key not in outputs:
                    continue
                outputs[key] = outputs[key][:max_seq_length]
        return outputs, is_truncation

    def get_ids(self) -> List[Union[int, str]]:
        """Returns the id of the examples."""
        ids = []
        for example in self.examples:
            ids.append(example.example_id)
        return ids

    def __len__(self) -> int:
        """Returns the length of the examples."""
        return len(self.input_features)

    def __getitem__(self,
                    index: int) -> Dict[str, torch.Tensor]:
        """Obtains the features of a given example index and converts them into a␣
↪dictionary."""
        features = self.input_features[index]
        data_dict = dict(
            input_ids=torch.tensor(features.input_ids, dtype=torch.long),
            attention_mask=torch.tensor(features.attention_mask, dtype=torch.float32)
```

(continues on next page)

(continued from previous page)

```
        )
        if features.token_type_ids is not None and self.config.return_token_type_ids:
            data_dict["token_type_ids"] = torch.tensor(features.token_type_ids,␣
↪dtype=torch.long)
        if features.trigger_left is not None:
            data_dict["trigger_left"] = torch.tensor(features.trigger_left, dtype=torch.
↪float32)
        if features.trigger_right is not None:
            data_dict["trigger_right"] = torch.tensor(features.trigger_right,␣
↪dtype=torch.float32)
        if features.labels is not None:
            data_dict["labels"] = torch.tensor(features.labels, dtype=torch.long)
        return data_dict

    def collate_fn(self, batch) -> Dict[str, torch.Tensor]:
        """Collates the samples in batches."""
        output_batch = dict()
        for key in batch[0].keys():
            output_batch[key] = torch.stack([x[key] for x in batch], dim=0)
        if self.config.truncate_in_batch:
            input_length = int(output_batch["attention_mask"].sum(-1).max())
            for key in ["input_ids", "attention_mask", "token_type_ids"]:
                if key not in output_batch:
                    continue
                output_batch[key] = output_batch[key][:, :input_length]
            if "labels" in output_batch and len(output_batch["labels"].shape) == 2:
                if self.config.truncate_seq2seq_output:
                    output_length = int((output_batch["labels"] != -100).sum(-1).max())
                    output_batch["labels"] = output_batch["labels"][:, :output_length]
                else:
                    output_batch["labels"] = output_batch["labels"][:, :input_length]
        return output_batch
```

6.6.6 EAEDataProcessor

The base class of data processor for event argument extraction, which would be inherited to construct task-specific data processors.

Attributes:

- `config`: The pre-defined configurations of the execution.
- `tokenizer`: The tokenizer method proposed for the tokenization process.
- `is_training`: A boolean variable indicating the state is training or not.
- `examples`: A list of ``EDInputExample``s constructed based on the input dataset.
- `input_features`: A list of ``EAEInputFeatures``s corresponding to the ``EAEInputExample``s.
- `data_for_evaluation`: A dictionary representing the evaluation data.
- `event_preds`: A list of event prediction data if the file exists.

```python
class EAEDataProcessor(Dataset):
    """Base class of data processor for event argument extraction.

    The base class of data processor for event argument extraction, which would be␣
↪inherited to construct task-specific
    data processors.

    Attributes:
        config:
            The pre-defined configurations of the execution.
        tokenizer:
            The tokenizer method proposed for the tokenization process.
        is_training (`bool`):
            A boolean variable indicating the state is training or not.
        examples (`List[EDInputExample]`):
            A list of `EDInputExample`s constructed based on the input dataset.
        input_features (`List[EAEInputFeatures]`):
            A list of `EAEInputFeatures`s corresponding to the `EAEInputExample`s.
        data_for_evaluation (`dict`):
            A dictionary representing the evaluation data.
        event_preds (`list`):
            A list of event prediction data if the file exists.
    """

    def __init__(self,
                 config,
                 tokenizer,
                 pred_file: str,
                 is_training: bool) -> None:
        """Constructs a EAEDataProcessor."""
        self.config = config
        self.tokenizer = tokenizer
        self.is_training = is_training
        if hasattr(config, "role2id"):
            self.config.role2id["X"] = -100
        self.examples = []
        self.input_features = []
        # data for trainer evaluation
        self.data_for_evaluation = {}
        # event prediction file path
        if pred_file is not None:
            if not os.path.exists(pred_file):
                logger.warning("%s doesn't exist.We use golden triggers" % pred_file)
                self.event_preds = None
            else:
                self.event_preds = json.load(open(pred_file))
        else:
            logger.warning("Event predictions is none! We use golden triggers.")
            self.event_preds = None

    def read_examples(self,
                      input_file: str):
        """Obtains a collection of `EAEInputExample`s for the dataset."""
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError

    def convert_examples_to_features(self):
        """Converts the `EAEInputExample`s into `EAEInputFeatures`s."""
        raise NotImplementedError

    def get_data_for_evaluation(self) -> Dict[str, Union[int, str]]:
        """Obtains the data for evaluation."""
        self.data_for_evaluation["pred_types"] = self.get_pred_types()
        self.data_for_evaluation["true_types"] = self.get_true_types()
        self.data_for_evaluation["ids"] = self.get_ids()
        if self.examples[0].argument_role is not None:
            self.data_for_evaluation["roles"] = self.get_roles()
        return self.data_for_evaluation

    def get_pred_types(self) -> List[str]:
        """Obtains the event type predicted by the model."""
        pred_types = []
        for example in self.examples:
            pred_types.append(example.pred_type)
        return pred_types

    def get_true_types(self) -> List[str]:
        """Obtains the actual event type from the annotation."""
        true_types = []
        for example in self.examples:
            true_types.append(example.true_type)
        return true_types

    def get_roles(self) -> List[str]:
        """Obtains the role of each argument mention."""
        roles = []
        for example in self.examples:
            roles.append(example.argument_role)
        return roles

    def _truncate(self,
                  outputs: Dict[str, List[int]],
                  max_seq_length: int):
        """Truncates the sequence that exceeds the maximum length."""
        is_truncation = False
        if len(outputs["input_ids"]) > max_seq_length:
            print("An instance exceeds the maximum length.")
            is_truncation = True
            for key in ["input_ids", "attention_mask", "token_type_ids", "offset_mapping
↪"]:
                if key not in outputs:
                    continue
                outputs[key] = outputs[key][:max_seq_length]
        return outputs, is_truncation

    def get_ids(self) -> List[Union[int, str]]:
```

(continues on next page)

(continued from previous page)

```
        """Returns the id of the examples."""
        ids = []
        for example in self.examples:
            ids.append(example.example_id)
        return ids

    def __len__(self) -> int:
        """Returns the length of the examples."""
        return len(self.input_features)

    def __getitem__(self,
                    index: int) -> Dict[str, torch.Tensor]:
        """Returns the features of a given example index in a dictionary."""
        features = self.input_features[index]
        data_dict = dict(
            input_ids=torch.tensor(features.input_ids, dtype=torch.long),
            attention_mask=torch.tensor(features.attention_mask, dtype=torch.float32)
        )
        if features.token_type_ids is not None and self.config.return_token_type_ids:
            data_dict["token_type_ids"] = torch.tensor(features.token_type_ids,␣
↪dtype=torch.long)
        if features.trigger_left is not None:
            data_dict["trigger_left"] = torch.tensor(features.trigger_left, dtype=torch.
↪long)
        if features.trigger_right is not None:
            data_dict["trigger_right"] = torch.tensor(features.trigger_right,␣
↪dtype=torch.long)
        if features.argument_left is not None:
            data_dict["argument_left"] = torch.tensor(features.argument_left,␣
↪dtype=torch.long)
        if features.argument_right is not None:
            data_dict["argument_right"] = torch.tensor(features.argument_right,␣
↪dtype=torch.long)
        if features.labels is not None:
            data_dict["labels"] = torch.tensor(features.labels, dtype=torch.long)
        return data_dict

    def collate_fn(self, batch) -> Dict[str, torch.Tensor]:
        """Collates the samples in batches."""
        output_batch = dict()
        for key in batch[0].keys():
            output_batch[key] = torch.stack([x[key] for x in batch], dim=0)
        if self.config.truncate_in_batch:
            input_length = int(output_batch["attention_mask"].sum(-1).max())
            for key in ["input_ids", "attention_mask", "token_type_ids"]:
                if key not in output_batch:
                    continue
                output_batch[key] = output_batch[key][:, :input_length]
            if "labels" in output_batch and len(output_batch["labels"].shape) == 2:
                if self.config.truncate_seq2seq_output:
                    output_length = int((output_batch["labels"] != -100).sum(-1).max())
                    output_batch["labels"] = output_batch["labels"][:, :output_length]
```

(continues on next page)

(continued from previous page)

```
                else:
                    output_batch["labels"] = output_batch["labels"][:, :input_length]
        return output_batch
```

6.7 Token Classification Processor

```
import json
import logging

from tqdm import tqdm
from typing import List, Optional, Dict

from .base_processor import (
    EDDataProcessor,
    EDInputExample,
    EDInputFeatures,
    EAEDataProcessor,
    EAEInputExample,
    EAEInputFeatures
)

logger = logging.getLogger(__name__)
```

6.7.1 EDTCProcessor

Data processor for token classification for event detection. The class is inherited from the`EDDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; the rest of the attributes and functions are multiplexed from the `EDDataProcessor` class.

```
class EDTCProcessor(EDDataProcessor):
    """Data processor for token classification for event detection.

    Data processor for token classification for event detection. The class is inherited␣
↪from the`EDDataProcessor` class,
    in which the undefined functions, including `read_examples()` and `convert_examples_
↪to_features()` are  implemented;
    the rest of the attributes and functions are multiplexed from the `EDDataProcessor`␣
↪class.
    """

    def __init__(self,
                 config,
                 tokenizer: str,
                 input_file: str) -> None:
        """Constructs an EDTCProcessor."""
        super().__init__(config, tokenizer)
        self.read_examples(input_file)
        self.convert_examples_to_features()
```

(continues on next page)

(continued from previous page)

```
    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EDInputExample`s for the dataset."""
        self.examples = []
        with open(input_file, "r") as f:
            for line in tqdm(f.readlines(), desc="Reading from %s" % input_file):
                item = json.loads(line.strip())
                # training and valid set
                if "events" in item:
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            example = EDInputExample(
                                example_id=trigger["id"],
                                text=item["text"],
                                trigger_left=trigger["position"][0],
                                trigger_right=trigger["position"][1],
                                labels=event["type"]
                            )
                            self.examples.append(example)
                if "negative_triggers" in item:
                    for neg in item["negative_triggers"]:
                        example = EDInputExample(
                            example_id=neg["id"],
                            text=item["text"],
                            trigger_left=neg["position"][0],
                            trigger_right=neg["position"][1],
                            labels="NA"
                        )
                        self.examples.append(example)
                # test set
                if "candidates" in item:
                    for candidate in item["candidates"]:
                        example = EDInputExample(
                            example_id=candidate["id"],
                            text=item["text"],
                            trigger_left=candidate["position"][0],
                            trigger_right=candidate["position"][1],
                            labels="NA",
                        )
                        # # if test set has labels
                        # assert not (self.config.test_exists_labels ^ ("type" in
↪candidate))
                        # if "type" in candidate:
                        #     example.labels = candidate["type"]
                        self.examples.append(example)

    def convert_examples_to_features(self) -> None:
        """Converts the `EDInputExample`s into `EDInputFeatures`s."""
        # merge and then tokenize
        self.input_features = []
        for example in tqdm(self.examples, desc="Processing features for TC"):
```

(continues on next page)

(continued from previous page)

```
            text_left = example.text[:example.trigger_left]
            text_mid = example.text[example.trigger_left:example.trigger_right]
            text_right = example.text[example.trigger_right:]

            if self.config.language == "Chinese":
                text = text_left + self.config.markers[0] + text_mid + self.config.
↪markers[1] + text_right
            else:
                text = text_left + self.config.markers[0] + " " + text_mid + " " + self.
↪config.markers[1] + text_right

            outputs = self.tokenizer(text, padding="max_length", truncation=True, max_
↪length=self.config.max_seq_length)
            is_overflow = False
            try:
                left = outputs["input_ids"].index(self.tokenizer.convert_tokens_to_
↪ids(self.config.markers[0]))
                right = outputs["input_ids"].index(self.tokenizer.convert_tokens_to_
↪ids(self.config.markers[1]))
            except:
                logger.warning("Markers are not in the input tokens.")
                left, right = 0, 0
                is_overflow = True

            # Roberta tokenizer doesn't return token_type_ids
            if "token_type_ids" not in outputs:
                outputs["token_type_ids"] = [0] * len(outputs["input_ids"])

            features = EDInputFeatures(
                example_id=example.example_id,
                input_ids=outputs["input_ids"],
                attention_mask=outputs["attention_mask"],
                token_type_ids=outputs["token_type_ids"],
                trigger_left=left,
                trigger_right=right
            )
            if example.labels is not None:
                features.labels = self.config.type2id[example.labels]
            self.input_features.append(features)
```

6.7.2 EAETCProcessor

Data processor for token classification for event argument extraction. The class is inherited from the `EAEDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; a new function entitled `insert_marker()` is defined, and the rest of the attributes and functions are multiplexed from the `EAEDataProcessor` class.

```
class EAETCProcessor(EAEDataProcessor):
    """Data processor for token classification for event argument extraction.

    Data processor for token classification for event argument extraction. The class is␣
```

(continues on next page)

(continued from previous page)

```
↪inherited from the
    `EAEDataProcessor` class, in which the undefined functions, including `read_
↪examples()` and
    `convert_examples_to_features()` are  implemented; a new function entitled `insert_
↪marker()` is defined, and
    the rest of the attributes and functions are multiplexed from the `EAEDataProcessor`␣
↪class.
    """

    def __init__(self,
                 config,
                 tokenizer: str,
                 input_file: str,
                 pred_file: str,
                 is_training: Optional[bool] = False):
        """Constructs a `EAETCProcessor`."""
        super().__init__(config, tokenizer, pred_file, is_training)
        self.read_examples(input_file)
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EAEInputExample`s for the dataset."""
        self.examples = []
        trigger_idx = 0
        with open(input_file, "r") as f:
            all_lines = f.readlines()
            for line in tqdm(all_lines, desc="Reading from %s" % input_file):
                item = json.loads(line.strip())
                if "events" in item:
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            true_type = event["type"]
                            if self.is_training or self.config.golden_trigger or self.
↪event_preds is None:
                                pred_type = true_type
                            else:
                                pred_type = self.event_preds[trigger_idx]

                            trigger_idx += 1

                            if self.config.eae_eval_mode in ['default', 'loose']:
                                if pred_type == "NA":
                                    continue

                            args_for_trigger = set()
                            positive_offsets = []
                            for argument in trigger["arguments"]:
                                for mention in argument["mentions"]:
                                    example = EAEInputExample(
                                        example_id=trigger["id"],
                                        text=item["text"],
```

(continues on next page)

(continued from previous page)

```
                                    pred_type=pred_type,
                                    true_type=event["type"],
                                    trigger_left=trigger["position"][0],
                                    trigger_right=trigger["position"][1],
                                    argument_left=mention["position"][0],
                                    argument_right=mention["position"][1],
                                    labels=argument["role"]
                                )
                                args_for_trigger.add(mention['mention_id'])
                                positive_offsets.append(mention["position"])
                                self.examples.append(example)
                        if "entities" in item:
                            for entity in item["entities"]:
                                # check whether the entity is an argument
                                is_argument = False
                                for mention in entity["mentions"]:
                                    if mention["mention_id"] in args_for_trigger:
                                        is_argument = True
                                        break
                                if is_argument:
                                    continue
                                # negative arguments
                                for mention in entity["mentions"]:
                                    example = EAEInputExample(
                                        example_id=trigger["id"],
                                        text=item["text"],
                                        pred_type=pred_type,
                                        true_type=event["type"],
                                        trigger_left=trigger["position"][0],
                                        trigger_right=trigger["position"][1],
                                        argument_left=mention["position"][0],
                                        argument_right=mention["position"][1],
                                        labels="NA"
                                    )
                                    if "train" in input_file or self.config.golden_
↪trigger:
                                        example.pred_type = event["type"]
                                    self.examples.append(example)
                        else:
                            for neg in item["negative_triggers"]:
                                is_argument = False
                                neg_set = set(range(neg["position"][0], neg["position
↪"][1]))
                                for pos_offset in positive_offsets:
                                    pos_set = set(range(pos_offset[0], pos_
↪offset[1]))
                                    if not pos_set.isdisjoint(neg_set):
                                        is_argument = True
                                        break
                                if is_argument:
                                    continue
                                example = EAEInputExample(
```

(continues on next page)

(continued from previous page)

```
                                    example_id=trigger["id"],
                                    text=item["text"],
                                    pred_type=pred_type,
                                    true_type=event["type"],
                                    trigger_left=trigger["position"][0],
                                    trigger_right=trigger["position"][1],
                                    argument_left=neg["position"][0],
                                    argument_right=neg["position"][1],
                                    labels="NA"
                                )
                                if "train" in input_file or self.config.golden_
↪trigger:
                                    example.pred_type = event["type"]
                                self.examples.append(example)

                    # negative triggers
                    for trigger in item["negative_triggers"]:
                        if self.config.eae_eval_mode in ['default', 'strict']:
                            if self.is_training or self.config.golden_trigger or self.
↪event_preds is None:
                                pred_type = "NA"
                            else:
                                pred_type = self.event_preds[trigger_idx]

                            if pred_type != "NA":
                                if "entities" in item:
                                    for entity in item["entities"]:
                                        for mention in entity["mentions"]:
                                            example = EAEInputExample(
                                                example_id=trigger_idx,
                                                text=item["text"],
                                                pred_type=pred_type,
                                                true_type="NA",
                                                trigger_left=trigger["position"][0],
                                                trigger_right=trigger["position"][1],
                                                argument_left=mention["position"][0],
                                                argument_right=mention["position"][1],
                                                labels="NA"
                                            )
                                            self.examples.append(example)
                                else:
                                    for neg in item["negative_triggers"]:
                                        example = EAEInputExample(
                                            example_id=trigger_idx,
                                            text=item["text"],
                                            pred_type=pred_type,
                                            true_type=event["type"],
                                            trigger_left=trigger["position"][0],
                                            trigger_right=trigger["position"][1],
                                            argument_left=neg["position"][0],
                                            argument_right=neg["position"][1],
                                            labels="NA"
```

(continues on next page)

(continued from previous page)

```
                                        )
                                        if "train" in input_file or self.config.golden_
↪trigger:
                                            example.pred_type = event["type"]
                                        self.examples.append(example)
                        trigger_idx += 1
                else:
                    for candi in item["candidates"]:
                        pred_type = self.event_preds[trigger_idx]   # we can only use␣
↪pred type here, gold not available
                        if pred_type != "NA":
                            if "entities" in item:
                                for entity in item["entities"]:
                                    for mention in entity["mentions"]:
                                        example = EAEInputExample(
                                            example_id=trigger_idx,
                                            text=item["text"],
                                            pred_type=pred_type,
                                            true_type="NA",
                                            trigger_left=candi["position"][0],
                                            trigger_right=candi["position"][1],
                                            argument_left=mention["position"][0],
                                            argument_right=mention["position"][1],
                                            labels="NA"
                                        )
                                        self.examples.append(example)
                            else:
                                for neg in item["negative_triggers"]:
                                    example = EAEInputExample(
                                        example_id=trigger_idx,
                                        text=item["text"],
                                        pred_type=pred_type,
                                        true_type=event["type"],
                                        trigger_left=trigger["position"][0],
                                        trigger_right=trigger["position"][1],
                                        argument_left=neg["position"][0],
                                        argument_right=neg["position"][1],
                                        labels="NA"
                                    )
                                    if "train" in input_file or self.config.golden_
↪trigger:
                                        example.pred_type = event["type"]
                                    self.examples.append(example)

                        trigger_idx += 1
            if self.event_preds is not None:
                assert trigger_idx == len(self.event_preds)

    def insert_marker(self,
                      text: str,
                      type: str,
                      trigger_position: List[int],
```

(continues on next page)

(continued from previous page)

```
                    argument_position: List[int],
                    markers: Dict[str, str],
                    whitespace: Optional[bool] = True) -> str:
        """Adds a marker at the start and end position of event triggers and argument␣
↪mentions."""
        markered_text = ""
        for i, char in enumerate(text):
            if i == trigger_position[0]:
                markered_text += markers[type][0]
                markered_text += " " if whitespace else ""
            if i == argument_position[0]:
                markered_text += markers["argument"][0]
                markered_text += " " if whitespace else ""
            markered_text += char
            if i == trigger_position[1] - 1:
                markered_text += " " if whitespace else ""
                markered_text += markers[type][1]
            if i == argument_position[1] - 1:
                markered_text += " " if whitespace else ""
                markered_text += markers["argument"][1]
        return markered_text

    def convert_examples_to_features(self) -> None:
        """Converts the `EAEInputExample`s into `EAEInputFeatures`s."""
        # merge and then tokenize
        self.input_features = []
        whitespace = True if self.config.language == "English" else False
        for example in tqdm(self.examples, desc="Processing features for TC"):
            text = self.insert_marker(example.text,
                                      example.pred_type,
                                      [example.trigger_left, example.trigger_right],
                                      [example.argument_left, example.argument_right],
                                      self.config.markers,
                                      whitespace)
            outputs = self.tokenizer(text,
                                      padding="max_length",
                                      truncation=True,
                                      max_length=self.config.max_seq_length)
            is_overflow = False
            # argument position
            try:
                argument_left = outputs["input_ids"].index(
                    self.tokenizer.convert_tokens_to_ids(self.config.markers["argument
↪"][0]))
                argument_right = outputs["input_ids"].index(
                    self.tokenizer.convert_tokens_to_ids(self.config.markers["argument
↪"][1]))
            except:
                argument_left, argument_right = 0, 0
                logger.warning("Argument markers are not in the input tokens.")
                is_overflow = True
            # trigger position
```

(continues on next page)

(continued from previous page)

```
            try:
                trigger_left = outputs["input_ids"].index(
                    self.tokenizer.convert_tokens_to_ids(self.config.markers[example.
↪pred_type][0]))
                trigger_right = outputs["input_ids"].index(
                    self.tokenizer.convert_tokens_to_ids(self.config.markers[example.
↪pred_type][1]))
            except:
                trigger_left, trigger_right = 0, 0
                logger.warning("Trigger markers are not in the input tokens.")
            # Roberta tokenizer doesn't return token_type_ids
            if "token_type_ids" not in outputs:
                outputs["token_type_ids"] = [0] * len(outputs["input_ids"])

            features = EAEInputFeatures(
                example_id=example.example_id,
                input_ids=outputs["input_ids"],
                attention_mask=outputs["attention_mask"],
                token_type_ids=outputs["token_type_ids"],
                trigger_left=trigger_left,
                trigger_right=trigger_right,
                argument_left=argument_left,
                argument_right=argument_right
            )
            if example.labels is not None:
                features.labels = self.config.role2id[example.labels]
                if is_overflow:
                    features.labels = -100
            self.input_features.append(features)
```

6.8 Sequence Labeling Processor

```
import json
import logging
from typing import List, Union, Any, Optional

from tqdm import tqdm
from .base_processor import (
    EDDataProcessor,
    EDInputExample,
    EDInputFeatures,
    EAEDataProcessor,
    EAEInputExample,
    EAEInputFeatures
)

logger = logging.getLogger(__name__)
```

6.8.1 EDSLProcessor

Data processor for sequence labeling for event detection. The class is inherited from the `EDDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; a new function entitled `get_final_labels()` is defined to obtain final results, and the rest of the attributes and functions are multiplexed from the `EDDataProcessor` class.

```
class EDSLProcessor(EDDataProcessor):
    """Data processor for sequence labeling for event detection.
    Data processor for sequence labeling for event detection. The class is inherited␣
↪from the `EDDataProcessor` class,
    in which the undefined functions, including `read_examples()` and `convert_examples_
↪to_features()` are  implemented;
    a new function entitled `get_final_labels()` is defined to obtain final results, and␣
↪the rest of the attributes and
    functions are multiplexed from the `EDDataProcessor` class.
    Attributes:
        is_overflow:
    """

    def __init__(self,
                 config,
                 tokenizer: str,
                 input_file: str) -> None:
        """Constructs a EDSLProcessor."""
        super().__init__(config, tokenizer)
        self.read_examples(input_file)
        self.is_overflow = []
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EDInputExample`s for the dataset."""
        self.examples = []
        language = self.config.language

        with open(input_file, "r", encoding="utf-8") as f:
            for line in tqdm(f.readlines(), desc="Reading from %s" % input_file):
                item = json.loads(line.strip())
                text = item["text"]
                words = get_words(text=text, language=language)
                labels = ["O"] * len(words)

                if "events" in item:
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            left_pos, right_pos = get_left_and_right_pos(text, trigger,␣
↪language, True)
                            labels[left_pos] = f"B-{event['type']}"
                            for i in range(left_pos + 1, right_pos):
                                labels[i] = f"I-{event['type']}"

                example = EDInputExample(example_id=item["id"], text=words,␣
```

(continues on next page)

(continued from previous page)

```
↪labels=labels)
                self.examples.append(example)

    def get_final_labels(self,
                         example: EDInputExample,
                         word_ids_of_each_token: List[int],
                         label_all_tokens: Optional[bool] = False) -> List[Union[str,␣
↪int]]:
        """Obtains the final label of each token."""
        final_labels = []
        pre_word_id = None
        for word_id in word_ids_of_each_token:
            if word_id is None:
                final_labels.append(-100)
            elif word_id != pre_word_id:  # first split token of a word
                final_labels.append(self.config.type2id[example.labels[word_id]])
            else:
                final_labels.append(self.config.type2id[example.labels[word_id]] if␣
↪label_all_tokens else -100)
            pre_word_id = word_id

        return final_labels

    def convert_examples_to_features(self) -> None:
        """Converts the `EDInputExample`s into `EDInputFeatures`s."""
        self.input_features = []

        for example in tqdm(self.examples, desc="Processing features for SL"):
            outputs = self.tokenizer(example.text,
                                     padding="max_length",
                                     truncation=False,
                                     max_length=self.config.max_seq_length,
                                     is_split_into_words=True)
            # Roberta tokenizer doesn't return token_type_ids
            if "token_type_ids" not in outputs:
                outputs["token_type_ids"] = [0] * len(outputs["input_ids"])

            outputs, is_overflow = self._truncate(outputs, self.config.max_seq_length)
            self.is_overflow.append(is_overflow)

            word_ids_of_each_token = get_word_ids(self.tokenizer, outputs, example.
↪text)[: self.config.max_seq_length]
            final_labels = self.get_final_labels(example, word_ids_of_each_token, label_
↪all_tokens=False)

            features = EDInputFeatures(
                example_id=example.example_id,
                input_ids=outputs["input_ids"],
                attention_mask=outputs["attention_mask"],
                token_type_ids=outputs["token_type_ids"],
                labels=final_labels,
            )
```

(continues on next page)

(continued from previous page)

```
            self.input_features.append(features)
```

6.8.2 EAESLProcessor

Data processor for sequence labeling for event argument extraction. The class is inherited from the `EAEDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; twp new functions, entitled `get_final_labels()` and `insert_markers()` are defined, and the rest of the attributes and functions are multiplexed from the `EAEDataProcessor` class.

Attributes:

- `positive_candidate_indices`: A list of integers indicating the indices of positive trigger candidates.

```
class EAESLProcessor(EAEDataProcessor):
    """Data processor for sequence labeling for event argument extraction.
    Data processor for sequence labeling for event argument extraction. The class is␣
↪inherited from the
    `EAEDataProcessor` class, in which the undefined functions, including `read_
↪examples()` and
    `convert_examples_to_features()` are  implemented; twp new functions, entitled `get_
↪final_labels()` and
    `insert_markers()` are defined, and the rest of the attributes and functions are␣
↪multiplexed from the
    `EAEDataProcessor` class.
    Attributes:
        positive_candidate_indices (`List[int]`):
            A list of integers indicating the indices of positive trigger candidates.
        is_overflow:
    """

    def __init__(self,
                 config: str,
                 tokenizer: str,
                 input_file: str,
                 pred_file: str,
                 is_training: Optional[bool] = False) -> None:
        """Constructs an EAESLProcessor/"""
        super().__init__(config, tokenizer, pred_file, is_training)
        self.positive_candidate_indices = []
        self.is_overflow = []
        self.config.role2id["X"] = -100
        self.read_examples(input_file)
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EAEInputExample`s for the dataset."""
        self.examples = []
        language = self.config.language
        trigger_idx = 0
        with open(input_file, "r", encoding="utf-8") as f:
            for line in tqdm(f.readlines(), desc="Reading from %s" % input_file):
```

(continues on next page)

(continued from previous page)

```
                item = json.loads(line.strip())
                text = item["text"]
                words = get_words(text=text, language=language)

                if "events" in item:
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            pred_type = self.get_single_pred(trigger_idx, input_file,␣
↪true_type=event["type"])
                            trigger_idx += 1

                            # Evaluation mode for EAE
                            # If the predicted event type is NA, We don't consider the␣
↪trigger
                            if self.config.eae_eval_mode in ["default", "loose"] and␣
↪pred_type == "NA":
                                continue
                            trigger_left, trigger_right = get_left_and_right_pos(text,␣
↪trigger, language, True)
                            labels = ["O"] * len(words)

                            for argument in trigger["arguments"]:
                                for mention in argument["mentions"]:
                                    left_pos, right_pos = get_left_and_right_pos(text,␣
↪mention, language, True)
                                    labels[left_pos] = f"B-{argument['role']}"
                                    for i in range(left_pos + 1, right_pos):
                                        labels[i] = f"I-{argument['role']}"

                            example = EAEInputExample(
                                example_id=item["id"],
                                text=words,
                                pred_type=pred_type,
                                true_type=event["type"],
                                trigger_left=trigger_left,
                                trigger_right=trigger_right,
                                labels=labels,
                            )
                            self.examples.append(example)

                    # negative triggers
                    for neg in item["negative_triggers"]:
                        pred_type = self.get_single_pred(trigger_idx, input_file, true_
↪type="NA")
                        trigger_idx += 1
                        if self.config.eae_eval_mode == "loose":
                            continue
                        elif self.config.eae_eval_mode in ["default", "strict"]:
                            if pred_type != "NA":
                                neg_left, neg_right = get_left_and_right_pos(text, neg,␣
↪language, True)
                                example = EAEInputExample(
```

(continues on next page)

(continued from previous page)

```
                                    example_id=item["id"],
                                    text=words,
                                    pred_type=pred_type,
                                    true_type="NA",
                                    trigger_left=neg_left,
                                    trigger_right=neg_right,
                                    labels=["O"] * len(words),
                                )
                                self.examples.append(example)
                        else:
                            raise ValueError("Invalid eac_eval_mode: %s" % self.config.
↪eae_eval_mode)
                else:
                    for can in item["candidates"]:
                        can_left, can_right = get_left_and_right_pos(text, can, language,
↪ True)
                        labels = ["O"] * len(words)
                        pred_type = self.event_preds[trigger_idx]
                        trigger_idx += 1
                        if pred_type != "NA":
                            example = EAEInputExample(
                                example_id=item["id"],
                                text=words,
                                pred_type=pred_type,
                                true_type="NA",  # true type not given, set to NA.
                                trigger_left=can_left,
                                trigger_right=can_right,
                                labels=labels,
                            )
                            self.examples.append(example)
                            self.positive_candidate_indices.append(trigger_idx-1)
            if self.event_preds is not None:
                assert trigger_idx == len(self.event_preds)

    def get_final_labels(self,
                         labels: dict,
                         word_ids_of_each_token: List[Any],
                         label_all_tokens: bool = False) -> List[Union[str, int]]:
        """Obtains the final label of each token."""
        final_labels = []
        pre_word_id = None
        for word_id in word_ids_of_each_token:
            if word_id is None:
                final_labels.append(-100)
            elif word_id != pre_word_id:  # first split token of a word
                final_labels.append(self.config.role2id[labels[word_id]])
            else:
                final_labels.append(self.config.role2id[labels[word_id]] if label_all_
↪tokens else -100)
            pre_word_id = word_id

        return final_labels
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def insert_marker(text: list,
                      event_type: str,
                      labels,
                      trigger_pos: List[int],
                      markers):
        """Adds a marker at the start and end position of event triggers and argument␣
↪mentions."""
        left, right = trigger_pos

        marked_text = text[:left] + [markers[event_type][0]] + text[left:right] +␣
↪[markers[event_type][1]] + text[right:]
        marked_labels = labels[:left] + ["X"] + labels[left:right] + ["X"] +␣
↪labels[right:]

        assert len(marked_text) == len(marked_labels)
        return marked_text, marked_labels

    def convert_examples_to_features(self) -> None:
        """Converts the `EAEInputExample`s into `EAEInputFeatures`s."""
        self.input_features = []
        self.is_overflow = []

        for example in tqdm(self.examples, desc="Processing features for SL"):
            text, labels = self.insert_marker(example.text,
                                              example.pred_type,
                                              example.labels,
                                              [example.trigger_left, example.trigger_
↪right],
                                              self.config.markers)
            outputs = self.tokenizer(text,
                                     padding="max_length",
                                     truncation=False,
                                     max_length=self.config.max_seq_length,
                                     is_split_into_words=True)
            # Roberta tokenizer doesn't return token_type_ids
            if "token_type_ids" not in outputs:
                outputs["token_type_ids"] = [0] * len(outputs["input_ids"])
            outputs, is_overflow = self._truncate(outputs, self.config.max_seq_length)
            self.is_overflow.append(is_overflow)

            word_ids_of_each_token = get_word_ids(self.tokenizer, outputs, example.
↪text)[: self.config.max_seq_length]
            final_labels = self.get_final_labels(labels, word_ids_of_each_token, label_
↪all_tokens=False)

            features = EAEInputFeatures(
                example_id=example.example_id,
                input_ids=outputs["input_ids"],
                attention_mask=outputs["attention_mask"],
                token_type_ids=outputs["token_type_ids"],
```

(continues on next page)

(continued from previous page)

```
                labels=final_labels,
            )
            self.input_features.append(features)
```

6.9 Sequence-to-Sequence Processor

```
import re
import json
import logging
from typing import List, Union, Tuple, Optional

from tqdm import tqdm
from collections import defaultdict
from .base_processor import (
    EDInputExample,
    EDDataProcessor,
    EDInputFeatures,
    EAEDataProcessor,
    EAEInputExample,
    EAEInputFeatures
)

type_start = "<"
type_end = ">"
split_word = ":"

logger = logging.getLogger(__name__)
```

6.9.1 extract_argument

Extracts the arguments from the raw text.

Args:

- `raw_text`: A string indicating the input raw text.
- `instance_id`: The id of the input example.
- `event_type`: A string indicating the type of the event.
- `template`: The template of the event argument extraction.

```
def extract_argument(raw_text: str,
                     instance_id: Union[int, str],
                     event_type: str,
                     template=re.compile(f"{type_start}|{type_end}")) -> List[Tuple]:
    """Extracts the arguments from the raw text.
    Args:
        raw_text (`str`):
            A string indicating the input raw text.
        instance_id (`Union[int, str]`):
```

(continues on next page)

(continued from previous page)

```
            The id of the input example.
        event_type (`str`):
            A string indicating the type of the event.
        template (`optional`, defaults to `re.compile(f"[{type_start}{type_end}]")`):
            The template of the event argument extraction.
    """
    arguments = []
    for span in template.split(raw_text):
        if span.strip() == "":
            continue
        words = span.strip().split(split_word)
        if len(words) != 2:
            continue
        role = words[0].strip().replace(" ", "")
        value = words[1].strip().replace(" ", "")
        if role != "" and value != "":
            arguments.append((instance_id, event_type, role, value))
    arguments = list(set(arguments))
    return arguments
```

6.9.2 EDSeq2SeqProcessor

Data processor for Sequence-to-Sequence (Seq2Seq) for event detection. The class is inherited from the `EDDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; the rest of the attributes and functions are multiplexed from the `EDDataProcessor` class.

```
class EDSeq2SeqProcessor(EDDataProcessor):
    """Data processor for Sequence-to-Sequence (Seq2Seq) for event detection.
    Data processor for Sequence-to-Sequence (Seq2Seq) for event detection. The class is␣
↪inherited from the
    `EDDataProcessor` class, in which the undefined functions, including `read_
↪examples()` and
    `convert_examples_to_features()` are  implemented; the rest of the attributes and␣
↪functions are multiplexed from the
    `EDDataProcessor` class.
    """

    def __init__(self,
                 config,
                 tokenizer,
                 input_file: str) -> None:
        """Constructs a `EDSeq2SeqProcessor`."""
        super().__init__(config, tokenizer)
        self.read_examples(input_file)
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EDInputExample`s for the dataset."""
        self.examples = []
```

(continues on next page)

(continued from previous page)

```
        with open(input_file, "r", encoding="utf-8") as f:
            for idx, line in enumerate(tqdm(f.readlines(), desc="Reading from %s" %␣
↪input_file)):
                item = json.loads(line.strip())
                if "source" in item:
                    kwargs = {"source": [item["source"]]}
                    if item["source"] in ["<duee>", "<fewfc>", "<leven>"]:
                        self.config.language = "Chinese"
                    else:
                        self.config.language = "English"
                else:
                    kwargs = {"source": []}

                words = get_words(text=item["text"], language=self.config.language)
                # training and valid set
                if "events" in item:
                    labels = []
                    for event in item["events"]:
                        type = get_plain_label(event["type"])
                        for trigger in event["triggers"]:
                            labels.append(f"{type_start} {type}{split_word} {trigger[
↪'trigger_word']} {type_end}")
                    labels = "".join(labels)

                    example = EDInputExample(
                        example_id=idx,
                        text=words,
                        labels=labels,
                        **kwargs,
                    )
                    self.examples.append(example)
                else:
                    example = EDInputExample(example_id=idx, text=words, labels="",␣
↪**kwargs)
                    self.examples.append(example)

    def convert_examples_to_features(self) -> None:
        """Converts the `EDInputExample`s into `EDInputFeatures`s."""
        self.input_features = []
        for example in tqdm(self.examples, desc="Processing features for Seq2Seq"):
            # context
            input_context = self.tokenizer(example.kwargs["source"]+example.text,
                                           truncation=True,
                                           padding="max_length",
                                           max_length=self.config.max_seq_length,
                                           is_split_into_words=True)
            # output labels
            label_outputs = self.tokenizer(example.labels.split(),
                                           truncation=True,
                                           padding="max_length",
                                           max_length=self.config.max_out_length,
                                           is_split_into_words=True)
```

(continues on next page)

(continued from previous page)

```
            # set -100 to unused token
            for i, flag in enumerate(label_outputs["attention_mask"]):
                if flag == 0:
                    label_outputs["input_ids"][i] = -100
            features = EDInputFeatures(
                example_id=example.example_id,
                input_ids=input_context["input_ids"],
                attention_mask=input_context["attention_mask"],
                labels=label_outputs["input_ids"],
            )
            self.input_features.append(features)
```

6.9.3 EAESeq2SeqProcessor

Data processor for token classification for event argument extraction. The class is inherited from the `EAEDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; a new function entitled `insert_marker()` is defined, and the rest of the attributes and functions are multiplexed from the `EAEDataProcessor` class.

```
class EAESeq2SeqProcessor(EAEDataProcessor):
    """Data processor for sequence to sequence for event argument extraction.
    Data processor for token classification for event argument extraction. The class is␣
↪inherited from the
    `EAEDataProcessor` class, in which the undefined functions, including `read_
↪examples()` and
    `convert_examples_to_features()` are  implemented; a new function entitled `insert_
↪marker()` is defined, and
    the rest of the attributes and functions are multiplexed from the `EAEDataProcessor`␣
↪class.
    """

    def __init__(self,
                 config,
                 tokenizer: str,
                 input_file: str,
                 pred_file: str,
                 is_training: Optional[bool] = False) -> None:
        """Constructs a `EAESeq2SeqProcessor`."""
        super().__init__(config, tokenizer, pred_file, is_training)
        self.read_examples(input_file)
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EAEInputExample`s for the dataset."""
        self.examples = []
        self.data_for_evaluation["golden_arguments"] = []
        self.data_for_evaluation["roles"] = []
        language = self.config.language
        trigger_idx = 0
        with open(input_file, "r", encoding="utf-8") as f:
```

(continues on next page)

(continued from previous page)

```
            for line in tqdm(f.readlines(), desc="Reading from %s" % input_file):
                item = json.loads(line.strip())
                if "source" in item:
                    kwargs = {"source": [item["source"]]}
                    if item["source"] in ["<duee>", "<fewfc>", "<leven>"]:
                        self.config.language = "Chinese"
                    else:
                        self.config.language = "English"
                else:
                    kwargs = {"source": []}

                text = item["text"]
                words = get_words(text=text, language=language)

                if "events" in item:
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            pred_type = self.get_single_pred(trigger_idx, input_file,␣
↪true_type=event["type"])
                            pred_type = get_plain_label(pred_type)
                            trigger_idx += 1

                            # Evaluation mode for EAE
                            # If the predicted event type is NA, We don't consider the␣
↪trigger
                            if self.config.eae_eval_mode in ["default", "loose"] and␣
↪pred_type == "NA":
                                continue

                            labels = []
                            arguments_per_trigger = defaultdict(list)
                            for argument in trigger["arguments"]:
                                role = get_plain_label(argument["role"])
                                for mention in argument["mentions"]:
                                    arguments_per_trigger[role].append(mention["mention
↪"])
                                    labels.append(f"{type_start} {role}{split_word}
↪{mention['mention']} {type_end}")
                            labels = "".join(labels)

                            self.data_for_evaluation["golden_arguments"].
↪append(dict(arguments_per_trigger))
                            example = EAEInputExample(
                                example_id=trigger_idx - 1,
                                text=words,
                                pred_type=pred_type,
                                true_type=get_plain_label(event["type"]),
                                trigger_left=trigger["position"][0],
                                trigger_right=trigger["position"][1],
                                labels=labels,
                                **kwargs,
                            )
```

(continues on next page)

(continued from previous page)

```
                        self.examples.append(example)
                # negative triggers
                for neg_trigger in item["negative_triggers"]:
                    pred_type = self.get_single_pred(trigger_idx, input_file, true_
↪type="NA")
                    pred_type = get_plain_label(pred_type)
                    trigger_idx += 1

                    if self.config.eae_eval_mode == "loose":
                        continue
                    elif self.config.eae_eval_mode in ["default", "strict"]:
                        if pred_type != "NA":
                            arguments_per_trigger = {}
                            self.data_for_evaluation["golden_arguments"].
↪append(dict(arguments_per_trigger))
                            example = EAEInputExample(
                                example_id=trigger_idx - 1,
                                text=words,
                                pred_type=pred_type,
                                true_type="NA",
                                trigger_left=neg_trigger["position"][0],
                                trigger_right=neg_trigger["position"][1],
                                labels="",
                                **kwargs,
                            )
                            self.examples.append(example)
                    else:
                        raise ValueError("Invaild eac_eval_mode: %s" % self.config.
↪eae_eval_mode)
            else:
                for candi in item["candidates"]:
                    pred_type = self.event_preds[trigger_idx]
                    pred_type = get_plain_label(pred_type)
                    trigger_idx += 1
                    if pred_type != "NA":
                        arguments_per_trigger = {}
                        self.data_for_evaluation["golden_arguments"].
↪append(dict(arguments_per_trigger))
                        example = EAEInputExample(
                            example_id=trigger_idx - 1,
                            text=words,
                            pred_type=pred_type,
                            true_type="NA",  # true type not given, set to NA.
                            trigger_left=candi["position"][0],
                            trigger_right=candi["position"][1],
                            labels="",
                            **kwargs,
                        )
                        self.examples.append(example)
        if self.event_preds is not None and not self.config.golden_trigger:
            assert trigger_idx == len(self.event_preds)
        print('there are {} examples'.format(len(self.examples)))
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def insert_marker(tokens: List[str],
                      trigger_pos: List[int],
                      markers: List,
                      whitespace: Optional[bool] = True) -> List[str]:
        """Adds a marker at the start and end position of event triggers and argument␣
↪mentions."""
        space = " " if whitespace else ""
        marked_words = []
        char_pos = 0
        for i, token in enumerate(tokens):
            if char_pos == trigger_pos[0]:
                marked_words.append(markers[0])
            char_pos += len(token) + len(space)
            marked_words.append(token)
            if char_pos == trigger_pos[1] + len(space):
                marked_words.append(markers[1])
        return marked_words

    def convert_examples_to_features(self) -> None:
        """Converts the `EAEInputExample`s into `EAEInputFeatures`s."""
        self.input_features = []
        whitespace = True if self.config.language == "English" else False
        for example in tqdm(self.examples, desc="Processing features for Seq2Seq"):
            # context
            words = self.insert_marker(example.text,
                                       [example.trigger_left, example.trigger_right],
                                       self.config.markers,
                                       whitespace)
            input_context = self.tokenizer(example.kwargs["source"] + words,
                                           truncation=True,
                                           padding="max_length",
                                           max_length=self.config.max_seq_length,
                                           is_split_into_words=True)
            # output labels
            label_outputs = self.tokenizer(example.labels.split(),
                                           padding="max_length",
                                           truncation=True,
                                           max_length=self.config.max_out_length,
                                           is_split_into_words=True)
            # set -100 to unused token
            for i, flag in enumerate(label_outputs["attention_mask"]):
                if flag == 0:
                    label_outputs["input_ids"][i] = -100
            features = EAEInputFeatures(
                example_id=example.example_id,
                input_ids=input_context["input_ids"],
                attention_mask=input_context["attention_mask"],
                labels=label_outputs["input_ids"],
            )
            self.input_features.append(features)
```

6.10 Machine Reading Comprehension (MRC) Converter

```
import collections
import logging
from typing import Dict, List, Optional

logger = logging.getLogger(__name__)
```

6.10.1 read_query_templates

Loads query templates from a prompt file. If a translation is required, the query templates would be translated from English to Chinese based on four types of regulations.

Args:

- `prompt_file`: A string indicating the path of the prompt file.
- `translate`: A boolean variable indicating whether or not to translate the query templates into Chinese.

Returns:

- `query_templates`: A dictionary containing the query templates applicable for every event type and argument role.

```
def read_query_templates(prompt_file: str,
                         translate: Optional[bool] = False) -> Dict[str, Dict[str,␣
↪List[str]]]:
    """Loads query templates from a prompt file.
    Loads query templates from a prompt file. If a translation is required, the query␣
↪templates would be translated from
    English to Chinese based on four types of regulations.
    Args:
        prompt_file (`str`):
            A string indicating the path of the prompt file.
        translate (`bool`, `optional`, defaults to `False`):
            A boolean variable indicating whether or not to translate the query␣
↪templates into Chinese.
    Returns:
        query_templates (`Dict[str, Dict[str, List[str]]]`)
            A dictionary containing the query templates applicable for every event type␣
↪and argument role.
    """
    et_translation = dict()
    ar_translation = dict()
    if translate:
        # the event types and argument roles in ACE2005-zh are expressed in English, we␣
↪translate them to Chinese
        et_file = "/".join(prompt_file.split('/')[:-1]) + "/chinese_event_types.txt"
        title = None

        for line in open(et_file, encoding='utf-8').readlines():
            num = line.split()[0]
            chinese = line.split()[1][:line.split()[1].index("")]
            english = line[line.index("") + 1:line.index('')]
```

(continues on next page)

(continued from previous page)

```
            if '.' not in num:
                title = chinese, english
            if title:
                et_translation['{}.{}'.format(title[1], english)] = "{}.{}".
↪format(title[0], chinese)

        ar_file = "/".join(prompt_file.split('/')[:-1]) + "/chinese_arg_roles.txt"
        for line in open(ar_file, encoding='utf-8').readlines():
            english, chinese = line.strip().split()
            ar_translation[english] = chinese

    query_templates = dict()
    with open(prompt_file, "r", encoding='utf-8') as f:
        for line in f:
            event_arg, query = line.strip().split(",")
            event_type, arg_name = event_arg.split("_")

            if event_type not in query_templates:
                query_templates[event_type] = dict()
            if arg_name not in query_templates[event_type]:
                query_templates[event_type][arg_name] = list()

            if translate:
                # 0 template arg_name
                query_templates[event_type][arg_name].append(ar_translation[arg_name])
                # 1 template arg_name + in trigger (replace [trigger] when forming the␣
↪instance)
                query_templates[event_type][arg_name].append(ar_translation[arg_name] +
↪"[trigger]")
                # 2 template arg_query
                query_templates[event_type][arg_name].append(query)
                # 3 arg_query + trigger (replace [trigger] when forming the instance)
                query_templates[event_type][arg_name].append(query[:-1] + "[trigger]?")
            else:
                # 0 template arg_name
                query_templates[event_type][arg_name].append(arg_name)
                # 1 template arg_name + in trigger (replace [trigger] when forming the␣
↪instance)
                query_templates[event_type][arg_name].append(arg_name + " in [trigger]")
                # 2 template arg_query
                query_templates[event_type][arg_name].append(query)
                # 3 arg_query + trigger (replace [trigger] when forming the instance)
                query_templates[event_type][arg_name].append(query[:-1] + " in [trigger]?
↪")

    for event_type in query_templates:
        for arg_name in query_templates[event_type]:
            assert len(query_templates[event_type][arg_name]) == 4

    return query_templates
```

6.10.2 _get_best_indexes

Gets the n-best logits from a list. The methods returns a list containing the indexes of the n-best logits that satisfies both the logits are n-best and greater than the logit of the "cls" token.

Args:

```
def _get_best_indexes(logits: List[int],
                      n_best_size: Optional[int] = 1,
                      larger_than_cls: Optional[bool] = False,
                      cls_logit: Optional[int] = None) -> List[int]:
    """Gets the n-best logits from a list.
    Gets the n-best logits from a list. The methods returns a list containing the␣
↪indexes of the n-best logits that
    satisfies both the logits are n-best and greater than the logit of the "cls" token.
    """
    index_and_score = sorted(enumerate(logits), key=lambda x: x[1], reverse=True)

    best_indexes = []
    for i in range(len(index_and_score)):
        if i >= n_best_size:
            break
        if larger_than_cls:
            if index_and_score[i][1] < cls_logit:
                break
        best_indexes.append(index_and_score[i][0])
    return best_indexes
```

6.10.3 char_pos_to_word_pos

Returns the word-level position of a mention by counting the number of words before the start position of the mention.

Args:

- `text`: A string representing the source text that the mention is within.
- `position`: An integer indicating the character-level position of the mention.

Returns:

- An integer indicating the word-level position of the mention.

```
def char_pos_to_word_pos(text: str,
                         position: int) -> int:
    """Returns the word-level position of a mention.
    Returns the word-level position of a mention by counting the number of words before␣
↪the start position of the
    mention.
    Args:
        text (`str`):
            A string representing the source text that the mention is within.
        position (`int`)
            An integer indicating the character-level position of the mention.
    Returns:
        An integer indicating the word-level position of the mention.
```

(continues on next page)

(continued from previous page)

```
    """
    return len(text[:position].split())
```

6.10.4 make_predictions

Obtains the prediction from the Machine Reading Comprehension (MRC) model.

```
def make_predictions(all_start_logits, all_end_logits, training_args):
    """Obtains the prediction from the Machine Reading Comprehension (MRC) model."""
    data_for_evaluation = training_args.data_for_evaluation
    assert len(all_start_logits) == len(data_for_evaluation["ids"])
    # all golden labels
    final_all_labels = []
    for arguments in data_for_evaluation["golden_arguments"]:
        arguments_per_trigger = []
        for argument in arguments["arguments"]:
            event_argument_type = arguments["true_type"] + "_" + argument["role"]
            for mention in argument["mentions"]:
                arguments_per_trigger.append(
                    (event_argument_type, (mention["position"][0], mention["position
↪"][1]), arguments["id"]))
        final_all_labels.extend(arguments_per_trigger)
    # predictions
    _PrelimPrediction = collections.namedtuple("PrelimPrediction",
                                               ["start_index", "end_index", "start_logit
↪", "end_logit"])
    final_all_predictions = []
    for example_id, (start_logits, end_logits) in enumerate(zip(all_start_logits, all_
↪end_logits)):
        event_argument_type = data_for_evaluation["pred_types"][example_id] + "_" + \
                              data_for_evaluation["roles"][example_id]
        start_indexes = _get_best_indexes(start_logits, 20, True, start_logits[0])
        end_indexes = _get_best_indexes(end_logits, 20, True, end_logits[0])
        # add span preds
        prelim_predictions = []
        for start_index in start_indexes:
            for end_index in end_indexes:
                if start_index < data_for_evaluation["text_range"][example_id]["start"]␣
↪or \
                        end_index < data_for_evaluation["text_range"][example_id]["start
↪"]:
                    continue
                if start_index >= data_for_evaluation["text_range"][example_id]["end"]␣
↪or \
                        end_index >= data_for_evaluation["text_range"][example_id]["end
↪"]:
                    continue
                if end_index < start_index:
                    continue
                word_start_index = start_index - 1
                word_end_index = end_index - 1
```

(continues on next page)

(continued from previous page)

```
                length = word_end_index - word_start_index + 1
                if length > 5:
                    continue
                prelim_predictions.append(
                    _PrelimPrediction(start_index=word_start_index, end_index=word_end_
↪index,
                                      start_logit=start_logits[start_index], end_
↪logit=end_logits[end_index]))
        # sort
        prelim_predictions = sorted(prelim_predictions, key=lambda x: (x.start_logit + x.
↪end_logit), reverse=True)
        # get final pred in format: [event_type_offset_argument_type, [start_offset, end_
↪offset]]
        max_num_pred_per_arg = 1
        predictions_per_query = []
        for _, pred in enumerate(prelim_predictions[:max_num_pred_per_arg]):
            na_prob = (start_logits[0] + end_logits[0]) - (pred.start_logit + pred.end_
↪logit)
            predictions_per_query.append((event_argument_type, (pred.start_index, pred.
↪end_index), na_prob,
                                          data_for_evaluation["ids"][example_id]))
        final_all_predictions.extend(predictions_per_query)

    logger.info("\nAll predictions and labels generated. %d %d\n" % (len(final_all_
↪predictions), len(final_all_labels)))
    return final_all_predictions, final_all_labels
```

6.10.5 find_best_thresh

```
def find_best_thresh(new_preds, new_all_gold):
    best_score = 0
    best_na_thresh = 0
    gold_arg_n, pred_arg_n = len(new_all_gold), 0

    candidate_preds = []
    for argument in new_preds:
        candidate_preds.append(argument[:-2] + argument[-1:])
        pred_arg_n += 1

        pred_in_gold_n, gold_in_pred_n = 0, 0
        # pred_in_gold_n
        for argu in candidate_preds:
            if argu in new_all_gold:
                pred_in_gold_n += 1
        # gold_in_pred_n
        for argu in new_all_gold:
            if argu in candidate_preds:
                gold_in_pred_n += 1

        prec_c, recall_c, f1_c = 0, 0, 0
```

(continues on next page)

(continued from previous page)

```
        if pred_arg_n != 0:
            prec_c = 100.0 * pred_in_gold_n / pred_arg_n
        else:
            prec_c = 0
        if gold_arg_n != 0:
            recall_c = 100.0 * gold_in_pred_n / gold_arg_n
        else:
            recall_c = 0
        if prec_c or recall_c:
            f1_c = 2 * prec_c * recall_c / (prec_c + recall_c)
        else:
            f1_c = 0

        if f1_c > best_score:
            best_score = f1_c
            best_na_thresh = argument[-2]

    return best_na_thresh + 1e-10
```

6.10.6 compute_mrc_F1_cls

```
def compute_mrc_F1_cls(all_predictions, all_labels):
    all_predictions = sorted(all_predictions, key=lambda x: x[-2])
    # best_na_thresh = 0
    best_na_thresh = find_best_thresh(all_predictions, all_labels)
    print("Best thresh founded. %.6f" % best_na_thresh)

    final_new_preds = []
    for argument in all_predictions:
        if argument[-2] < best_na_thresh:
            final_new_preds.append(argument[:-2] + argument[-1:])  # no na_prob

    # get results (classification)
    gold_arg_n, pred_arg_n, pred_in_gold_n, gold_in_pred_n = 0, 0, 0, 0
    # pred_arg_n
    for argument in final_new_preds:
        pred_arg_n += 1
    # gold_arg_n
    for argument in all_labels:
        gold_arg_n += 1
    # pred_in_gold_n
    for argument in final_new_preds:
        if argument in all_labels:
            pred_in_gold_n += 1
    # gold_in_pred_n
    for argument in all_labels:
        if argument in final_new_preds:
            gold_in_pred_n += 1

    prec_c, recall_c, f1_c = 0, 0, 0
```

(continues on next page)

(continued from previous page)

```
    if pred_arg_n != 0:
        prec_c = 100.0 * pred_in_gold_n / pred_arg_n
    else:
        prec_c = 0
    if gold_arg_n != 0:
        recall_c = 100.0 * gold_in_pred_n / gold_arg_n
    else:
        recall_c = 0
    if prec_c or recall_c:
        f1_c = 2 * prec_c * recall_c / (prec_c + recall_c)
    else:
        f1_c = 0

    logger.info("Precision: %.2f, recall: %.2f" % (prec_c, recall_c))
    return f1_c
```

6.11 Machine Reading Comprehension (MRC) Processor

```
import pdb
import json
import logging

from tqdm import tqdm
from .base_processor import (
    EAEDataProcessor,
    EAEInputExample,
    EAEInputFeatures
)
from .mrc_converter import read_query_templates
from .input_utils import get_words, get_left_and_right_pos
from collections import defaultdict

logger = logging.getLogger(__name__)
```

6.11.1 EAEMRCProcessor

Data processor for Machine Reading Comprehension (MRC) for event argument extraction. The class is inherited from the `EAEDataProcessor` class, in which the undefined functions, including `read_examples()` and `convert_examples_to_features()` are implemented; a new function entitled `remove_sub_word()` is defined to remove the annotations whose word is a sub-word, the rest of the attributes and functions are multiplexed from the `EAEDataProcessor` class.

```
class EAEMRCProcessor(EAEDataProcessor):
    """Data processor for Machine Reading Comprehension (MRC) for event argument␣
↪extraction.

    Data processor for Machine Reading Comprehension (MRC) for event argument extraction.
↪ The class is inherited from
```

(continues on next page)

(continued from previous page)

```
    the `EAEDataProcessor` class, in which the undefined functions, including `read_
↪examples()` and
    `convert_examples_to_features()` are implemented; a new function entitled `remove_sub_
↪word()` is defined to remove
    the annotations whose word is a sub-word, the rest of the attributes and functions␣
↪are multiplexed from the
    `EAEDataProcessor` class.
    """

    def __init__(self,
                 config,
                 tokenizer,
                 input_file: str,
                 pred_file: str,
                 is_training: bool = False) -> None:
        """Constructs a `EAEMRCProcessor`."""
        super().__init__(config, tokenizer, pred_file, is_training)
        self.read_examples(input_file)
        self.convert_examples_to_features()

    def read_examples(self,
                      input_file: str) -> None:
        """Obtains a collection of `EAEInputExample`s for the dataset."""
        self.examples = []
        self.data_for_evaluation["golden_arguments"] = []
        trigger_idx = 0
        query_templates = read_query_templates(self.config.prompt_file,
                                               translate=self.config.dataset_name ==
↪"ACE2005-ZH")
        template_id = self.config.mrc_template_id
        with open(input_file, "r", encoding="utf-8") as f:
            for idx, line in enumerate(tqdm(f.readlines(), desc="Reading from %s" %␣
↪input_file)):
                item = json.loads(line.strip())
                if "events" in item:
                    words = get_words(text=item["text"], language=self.config.language)
                    for event in item["events"]:
                        for trigger in event["triggers"]:
                            if self.is_training or self.config.golden_trigger or self.
↪event_preds is None:
                                pred_event_type = event["type"]
                            else:
                                pred_event_type = self.event_preds[trigger_idx]
                            trigger_idx += 1

                            # Evaluation mode for EAE
                            # If predicted event type is NA:
                            #   in [default] and [loose] modes, we don't consider the␣
↪trigger
                            #   in [strict] mode, we consider the trigger
                            if self.config.eae_eval_mode in ["default", "loose"] and␣
↪pred_event_type == "NA":
```

(continues on next page)

(continued from previous page)

```
                                continue

                            # golden label for the trigger
                            arguments_per_trigger = dict(id=trigger_idx-1,
                                                         arguments=[],
                                                         pred_type=pred_event_type,
                                                         true_type=event["type"])
                            for argument in trigger["arguments"]:
                                arguments_per_role = dict(role=argument["role"],␣
↪mentions=[])
                                for mention in argument["mentions"]:
                                    left_pos, right_pos = get_left_and_right_
↪pos(text=item["text"],
                                                                                ␣
↪trigger=mention,
                                                                                ␣
↪language=self.config.language)
                                    arguments_per_role["mentions"].append({
                                        "position": [left_pos, right_pos - 1]
                                    })
                                arguments_per_trigger["arguments"].append(arguments_per_
↪role)
                            self.data_for_evaluation["golden_arguments"].
↪append(arguments_per_trigger)

                            if pred_event_type == "NA":
                                assert self.config.eae_eval_mode == "strict"
                                # in strict mode, we add the gold args for the trigger␣
↪but do not make predictions
                                continue

                            trigger_left, trigger_right = get_left_and_right_
↪pos(text=item["text"],
                                                                                ␣
↪trigger=trigger,
                                                                                ␣
↪language=self.config.language)

                            for role in query_templates[pred_event_type].keys():
                                query = query_templates[pred_event_type][role][template_
↪id]
                                query = query.replace("[trigger]", self.tokenizer.
↪tokenize(trigger["trigger_word"])[0])
                                query = get_words(text=query, language=self.config.
↪language)
                                if self.is_training:
                                    no_answer = True
                                    for argument in trigger["arguments"]:
                                        if argument["role"] not in query_templates[pred_
↪event_type]:
                                            # raise ValueError(
                                            #     "No template for %s in %s" % (argument[
```

(continues on next page)

(continued from previous page)

```
↪"role"], pred_event_type)
                                        # )
                                        logger.warning(
                                            "No template for %s in %s" % (argument[
↪"role"], pred_event_type))
                                        pass
                                    if argument["role"] != role:
                                        continue
                                    no_answer = False
                                    for mention in argument["mentions"]:
                                        left_pos, right_pos = get_left_and_right_
↪pos(text=item["text"],
                                                                                    ␣
↪trigger=mention,
                                                                                    ␣
↪language=self.config.language)
                                        example = EAEInputExample(
                                            example_id=trigger_idx-1,
                                            text=words,
                                            pred_type=pred_event_type,
                                            true_type=event["type"],
                                            input_template=query,
                                            trigger_left=trigger_left,
                                            trigger_right=trigger_right,
                                            argument_left=left_pos,
                                            argument_right=right_pos - 1,
                                            argument_role=role
                                        )
                                        self.examples.append(example)
                                if no_answer:
                                    example = EAEInputExample(
                                        example_id=trigger_idx-1,
                                        text=words,
                                        pred_type=pred_event_type,
                                        true_type=event["type"],
                                        input_template=query,
                                        trigger_left=trigger_left,
                                        trigger_right=trigger_right,
                                        argument_left=-1,
                                        argument_right=-1,
                                        argument_role=role
                                    )
                                    self.examples.append(example)
                            else:
                                # one instance per query
                                example = EAEInputExample(
                                    example_id=trigger_idx-1,
                                    text=words,
                                    pred_type=pred_event_type,
                                    true_type=event["type"],
                                    input_template=query,
                                    trigger_left=trigger_left,
```

(continues on next page)

(continued from previous page)

```
                                    trigger_right=trigger_right,
                                    argument_left=-1,
                                    argument_right=-1,
                                    argument_role=role
                                )
                                self.examples.append(example)
                    # negative triggers
                    for neg_trigger in item["negative_triggers"]:
                        if self.is_training or self.config.golden_trigger or self.event_
↪preds is None:
                            pred_event_type = "NA"
                        else:
                            pred_event_type = self.event_preds[trigger_idx]

                        trigger_idx += 1
                        if self.config.eae_eval_mode == "loose":
                            continue
                        elif self.config.eae_eval_mode in ["default", "strict"]:
                            if pred_event_type == "NA":
                                continue
                            trigger_left, trigger_right = get_left_and_right_
↪pos(text=item["text"],
                                                                                  ␣
↪trigger=neg_trigger,
                                                                                  ␣
↪language=self.config.language)
                            for role in query_templates[pred_event_type].keys():
                                query = query_templates[pred_event_type][role][template_
↪id]
                                query = query.replace("[trigger]",
                                                      self.tokenizer.tokenize(neg_trigger[
↪"trigger_word"])[0])
                                query = get_words(text=query, language=self.config.
↪language)
                                # one instance per query
                                example = EAEInputExample(
                                    example_id=trigger_idx-1,
                                    text=words,
                                    pred_type=pred_event_type,
                                    true_type="NA",
                                    input_template=query,
                                    trigger_left=trigger_left,
                                    trigger_right=trigger_right,
                                    argument_left=-1,
                                    argument_right=-1,
                                    argument_role=role
                                )
                                self.examples.append(example)

                        else:
                            raise ValueError("Invalid eae_eval_mode: %s" % self.config.
↪eae_eval_mode)
```

(continues on next page)

(continued from previous page)

```
                else:
                    for candi in item["candidates"]:
                        trigger_left, trigger_right = get_left_and_right_pos(text=item[
↪"text"],
                                                                             ␣
↪trigger=candi,
                                                                             ␣
↪language=self.config.language)
                        pred_event_type = self.event_preds[trigger_idx]
                        trigger_idx += 1
                        if pred_event_type != "NA":
                            for role in query_templates[pred_event_type].keys():
                                query = query_templates[pred_event_type][role][template_
↪id]
                                query = query.replace("[trigger]", self.tokenizer.
↪tokenize(candi["trigger_word"])[0])
                                query = get_words(text=query, language=self.config.
↪language)
                                # one instance per query
                                example = EAEInputExample(
                                    example_id=trigger_idx-1,
                                    text=words,
                                    pred_type=pred_event_type,
                                    true_type="NA",
                                    input_template=query,
                                    trigger_left=trigger_left,
                                    trigger_right=trigger_right,
                                    argument_left=-1,
                                    argument_right=-1,
                                    argument_role=role
                                )
                                self.examples.append(example)
            if self.event_preds is not None:
                assert trigger_idx == len(self.event_preds)

    def convert_examples_to_features(self) -> None:
        """Converts the `EAEInputExample`s into `EAEInputFeatures`s."""
        self.input_features = []
        self.data_for_evaluation["text_range"] = []
        self.data_for_evaluation["text"] = []

        for example in tqdm(self.examples, desc="Processing features for MRC"):
            # context
            input_context = self.tokenizer(example.text,
                                           truncation=True,
                                           max_length=self.config.max_seq_length,
                                           is_split_into_words=True)
            # template
            input_template = self.tokenizer(example.input_template,
                                            truncation=True,
                                            padding="max_length",
                                            max_length=self.config.max_seq_length,
```

(continues on next page)

(continued from previous page)

```
                                            is_split_into_words=True)

            input_context = self.remove_sub_word(input_context)
            # concatenate
            input_ids = input_context["input_ids"] + input_template["input_ids"]
            attention_mask = input_context["attention_mask"] + input_template["attention_
↪mask"]
            token_type_ids = [0] * len(input_context["input_ids"]) + [1] * len(input_
↪template["input_ids"])
            # truncation
            input_ids = input_ids[:self.config.max_seq_length]
            attention_mask = attention_mask[:self.config.max_seq_length]
            token_type_ids = token_type_ids[:self.config.max_seq_length]
            # output labels
            start_position = 0 if example.argument_left == -1 else example.argument_left␣
↪+ 1
            end_position = 0 if example.argument_right == -1 else example.argument_right␣
↪+ 1
            # data for evaluation
            text_range = dict()
            text_range["start"] = 1
            text_range["end"] = text_range["start"] + sum(input_context["attention_mask
↪"][1:])
            self.data_for_evaluation["text_range"].append(text_range)
            self.data_for_evaluation["text"].append(example.text)
            # features
            features = EAEInputFeatures(
                example_id=example.example_id,
                input_ids=input_ids,
                attention_mask=attention_mask,
                token_type_ids=token_type_ids,
                argument_left=start_position,
                argument_right=end_position
            )
            self.input_features.append(features)

    @staticmethod
    def remove_sub_word(inputs):
        """Removes the annotations whose word is a sub-word."""
        outputs = defaultdict(list)
        pre_word_id = -1
        for token_id, word_id in enumerate(inputs.word_ids()):
            if token_id == 0 or (word_id != pre_word_id and word_id is not None):
                for key in inputs:
                    outputs[key].append(inputs[key][token_id])
            pre_word_id = word_id
        return outputs
```

6.12 Input Utils

6.12.1 get_bio_labels

Generates the id of the BIO labels corresponding to the original label. The correspondences between the BIO labels and their ids are saved in a dictionary.

Args:

- `original_labels`: A list of strings representing the original labels within the dataset.
- `labels_to_exclude`: A list of strings indicating the labels excluded to use, the id of which would not be generated.

Returns:

- `bio_labels`: A dictionary containing the correspondence the BIO labels and their ids.

```
def get_bio_labels(original_labels: List[str],
                   labels_to_exclude: Optional[List[str]] = ["NA"]) -> Dict[str, int]:
    """Generates the id of the BIO labels corresponding to the original label.

    Generates the id of the BIO labels corresponding to the original label. The␣
↪correspondences between the BIO labels
    and their ids are saved in a dictionary.

    Args:
        original_labels (`List[str]`):
            A list of strings representing the original labels within the dataset.
        labels_to_exclude (`List[str]`, `optional`, defaults to ["NA"]):
            A list of strings indicating the labels excluded to use, the id of which␣
↪would not be generated.

    Returns:
        bio_labels (`Dict[str, int]`):
            A dictionary containing the correspondence the BIO labels and their ids.
    """
    bio_labels = {"O": 0}
    for label in original_labels:
        if label in labels_to_exclude:
            continue
        bio_labels[f"B-{label}"] = len(bio_labels)
        bio_labels[f"I-{label}"] = len(bio_labels)
    return bio_labels
```

6.12.2 get_start_poses

Obtains the start position of each word within the sentence. The character-level start positions of each word are stored in a list.

Args:

- `sentence`: A string representing the input sentence.

Returns:

- `start_poses`: A list of integers representing the character-level start position of each word within the sentence.

```
def get_start_poses(sentence: str) -> List[int]:
    """Obtains the start position of each word within the sentence.

    Obtains the start position of each word within the sentence. The character-level␣
↪start positions of each word are
    stored in a list.

    Args:
        sentence (`str`):
            A string representing the input sentence.

    Returns:
        start_poses (`List[int]`):
            A list of integers representing the character-level start position of each␣
↪word within the sentence.
    """
    words = sentence.split()
    start_pos = 0
    start_poses = []
    for word in words:
        start_poses.append(start_pos)
        start_pos += len(word) + 1
    return start_poses
```

6.12.3 check_if_start

Check whether the start position of the mention is the beginning of a word, that is, check whether a trigger or an argument is a sub-word.

Args: - `start_poses`: A list of integers representing the character-level start position of each word within the sentence. - `char_pos`: A list of integers indicating the start and end position of a mention.

Returns:

Returns *True* if the start position of the mention is the start of a word; returns *False* otherwise.

```
def check_if_start(start_poses: List[int],
                   char_pos: List[int]) -> bool:
    """Check whether the start position of the mention is the beginning of a word.

    Check whether the start position of the mention is the beginning of a word, that is,␣
↪check whether a trigger or an
    argument is a sub-word.
```

(continues on next page)

(continued from previous page)

```
    Args:
        start_poses (`List[int]`):
            A list of integers representing the character-level start position of each␣
↪word within the sentence.
        char_pos (`List[int]`):
            A list of integers indicating the start and end position of a mention.

    Returns:
        Returns `True` if the start position of the mention is the start of a word;␣
↪returns `False` otherwise.
    """
    if char_pos[0] in start_poses:
        return True
    return False
```

6.12.4 get_word_position

Returns the word-level position of a given mention by matching the index of its character-level start position in the list containing the start position of each word within the sentence.

Args:

- `start_poses`: A list of integers representing the character-level start position of each word within the sentence.
- `char_pos`: A list of integers indicating the start and end position of a given mention.

Returns:

An integer indicating the word-level position of the given mention.

```
def get_word_position(start_poses: List[int],
                      char_pos: List[int]) -> int:
    """Returns the word-level position of a given mention.

    Returns the word-level position of a given mention by matching the index of its␣
↪character-level start position in
    the list containing the start position of each word within the sentence.

    Args:
        start_poses (`List[int]`):
            A list of integers representing the character-level start position of each␣
↪word within the sentence.
        char_pos (`List[int]`)
            A list of integers indicating the start and end position of a given mention.

    Returns:
        `int`:
            An integer indicating the word-level position of the given mention.
    """
    return start_poses.index(char_pos[0])
```

6.12.5 get_words

Obtains the words within the source text. The recognition of words differs according to language. The words are obtained through splitting white spaces in English, while each Chinese character is regarded as a word in Chinese.

Args:

- `text`: A string representing the input source text.
- `language`: A string indicating the language of the source text, English or Chinese.

Returns:

- `words` : A list of strings containing the words within the source text.

```
def get_words(text: str,
              language: str) -> List[str]:
    """Obtains the words within the given text.

    Obtains the words within the source text. The recognition of words differs according␣
↪to language. The words are
    obtained through splitting white spaces in English, while each Chinese character is␣
↪regarded as a word in Chinese.

    Args:
        text (`str`):
            A string representing the input source text.
        language (`str`):
            A string indicating the language of the source text, English or Chinese.

    Returns:
        words (`List[str]`):
            A list of strings containing the words within the source text.
    """
    if language == "English":
        words = text.split()
    elif language == "Chinese":
        words = list(text)
    else:
        raise NotImplementedError
    return words
```

6.12.6 get_left_and_right_pos

Obtains the word-level position of the trigger word's start and end position. The method of obtaining the position differs according to language. The method returns the number of words before the given position for English texts, while for Chinese, each character is regarded as a word.

Args:

- `text`: A string representing the source text that the trigger word is within.
- `trigger`: A dictionary containing the trigger word, position, and arguments of an event trigger.
- `language`: A string indicating the language of the source text and trigger word, English or Chinese.
- `keep_space`: A flag that indicates whether to keep the space in Chinese text during offset calculating. During data preprocessing, the space has to be kept due to the offsets consider space. During evaluation, the space is

automatically removed by the tokenizer and the output hidden states do not involve space logits, therefore, offset counting should not keep the space.

```
def get_left_and_right_pos(text: str,
                           trigger: Dict[str, Union[int, str, List[int], List[Dict]]],
                           language: str,
                           keep_space: bool = False) -> Tuple[int, int]:
    """Obtains the word-level position of the trigger word's start and end position.
    Obtains the word-level position of the trigger word's start and end position. The␣
↪method of obtaining the position
    differs according to language. The method returns the number of words before the␣
↪given position for English texts,
    while for Chinese, each character is regarded as a word.
    Args:
        text (`str`):
            A string representing the source text that the trigger word is within.
        trigger (`Dict[str, Union[int, str, List[int], List[Dict]]]`):
            A dictionary containing the trigger word, position, and arguments of an␣
↪event trigger.
        language (`str`):
            A string indicating the language of the source text and trigger word,␣
↪English or Chinese.
        keep_space (`bool`):
            A flag that indicates whether to keep the space in Chinese text during␣
↪offset calculating.
                During data preprocessing, the space has to be kept due to the offsets␣
↪consider space.
                During evaluation, the space is automatically removed by the tokenizer␣
↪and the output hidden states do
                not involve space logits, therefore, offset counting should not keep the␣
↪space.
    Returns:
        left_pos (`int`), right_pos (`int`):
            Two integers indicating the number of words before the start and end␣
↪position of the trigger word.
    """
    if language == "English":
        left_pos = len(text[:trigger["position"][0]].split())
        right_pos = len(text[:trigger["position"][1]].split())
    elif language == "Chinese":
        left_pos = trigger["position"][0] if keep_space else len("".join(text[:trigger[
↪"position"][0]].split()))
        right_pos = trigger["position"][1] if keep_space else len("".join(text[:trigger[
↪"position"][1]].split()))
    else:
        raise NotImplementedError
    return left_pos, right_pos
```

6.12.7 get_word_ids

Returns a list indicating the word corresponding to each token. Special tokens added by the tokenizer are mapped to None and other tokens are mapped to the index of their corresponding word (several tokens will be mapped to the same word index if they are parts of that word).

Args:

- tokenizer: The tokenizer that has been used for word tokenization.
- outputs: The outputs of the tokenizer.
- word_list: A list of word strings.

Returns:

- word_ids: A list mapping the tokens to their actual word in the initial sentence.

```
def get_word_ids(tokenizer: PreTrainedTokenizer,
                 outputs: BatchEncoding,
                 word_list: List[str]) -> List[int]:
    """Return a list mapping the tokens to their actual word in the initial sentence for␣
↪a tokenizer.
    Return a list indicating the word corresponding to each token. Special tokens added␣
↪by the tokenizer are mapped to
    None and other tokens are mapped to the index of their corresponding word (several␣
↪tokens will be mapped to the same
    word index if they are parts of that word).
    Args:
        tokenizer (`PreTrainedTokenizer`):
            The tokenizer that has been used for word tokenization.
        outputs (`BatchEncoding`):
            The outputs of the tokenizer.
        word_list (`List[str]`):
            A list of word strings.
    Returns:
        word_ids (`List[int]`):
            A list mapping the tokens to their actual word in the initial sentence
    """
    word_list = [w.lower() for w in word_list]
    try:
        word_ids = outputs.word_ids()
        return word_ids
    except:
        assert isinstance(tokenizer, WordLevelTokenizer)
        pass
    tokens = tokenizer.convert_ids_to_tokens(outputs["input_ids"])
    word_ids = []
    word_idx = 0

    for token in tokens:
        if token not in word_list and token != "[UNK]":
            word_ids.append(None)
        else:
            if token != "[UNK]":
                assert token == word_list[word_idx]
```

(continues on next page)

(continued from previous page)

```
            word_ids.append(word_idx)
            word_idx += 1
    return word_ids
```

6.12.8 check_pred_len

Check whether the length of the prediction sequence equals that of the original word sequence. The prediction sequence consists of prediction for each word in the original sentence. Sometimes, there might be special tokens or extra space in the original sentence, and the tokenizer will automatically ignore them, which may cause the output length differs from the input length.

Args:

- `pred`: A list of predicted event types or argument roles.
- `item`: A single item of the training/valid/test data.
- `language`: The language of the input text.

```
def check_pred_len(pred: List[str],
                   item: Dict[str, Union[str, List[dict]]],
                   language: str) -> None:
    """Check whether the length of the prediction sequence equals that of the original␣
↪word sequence.
    The prediction sequence consists of prediction for each word in the original␣
↪sentence. Sometimes, there might be
    special tokens or extra space in the original sentence, and the tokenizer will␣
↪automatically ignore them, which may
    cause the output length differs from the input length.
    Args:
        pred (`List[str]`):
            A list of predicted event types or argument roles.
        item (`Dict[str, Union[str, List[dict]]]`):
            A single item of the training/valid/test data.
        language ('str'):
            The language of the input text.
    Returns:
        None.
    """
    if language == "English":
        if len(pred) != len(item["text"].split()):
            logger.warning("There might be special tokens in the input text: {}".
↪format(item["text"]))

    elif language == "Chinese":
        if len(pred) != len("".join(item["text"].split())):  # remove space token
            logger.warning("There might be special tokens in the input text: {}".
↪format(item["text"]))
    else:
        raise NotImplementedError
```

6.12.9 get_ed_candidates

Obtain the candidate tokens for the event detection (ED) task. The unified evaluation considers prediction of each token that is possibly a trigger (ED candidate).

Args:

- `item`: A single item of the training/valid/test data.

Returns: - `candidates`: A list of dictionary that contains the possible trigger. - `label_names`: A list of string contains the ground truth label for each possible trigger.

```
def get_ed_candidates(item: Dict[str, Union[str, List[dict]]]) -> Tuple[List[dict],␣
↪List[str]]:
    """Obtain the candidate tokens for the event detection (ED) task.
    The unified evaluation considers prediction of each token that is possibly a trigger␣
↪(ED candidate).
    Args:
        item (`Dict[str, Union[str, List[dict]]]`):
            A single item of the training/valid/test data.
    Returns:
        candidates(`List[dict]`), label_names (`List[str]`):
            candidates: A list of dictionary that contains the possible trigger.
            label_names: A list of string contains the ground truth label for each␣
↪possible trigger.
    """
    candidates = []
    label_names = []
    if "events" in item:
        for event in item["events"]:
            for trigger in event["triggers"]:
                label_names.append(event["type"])
                candidates.append(trigger)
        for neg_trigger in item["negative_triggers"]:
            label_names.append("NA")
            candidates.append(neg_trigger)
    else:
        candidates = item["candidates"]

    return candidates, label_names
```

6.12.10 check_is_argument

Checks whether a given mention is argument or not. If it is an argument, we have to exclude it from the negative arguments list.

Args:

- `mention`: The mention that contains the word, position and other meta information like id, etc.
- `positive_offsets`: A list that contains the offsets of all the ground truth arguments.

Returns:

- `is_argument`: A flag that indicates whether the mention is an argument or not.

```
def check_is_argument(mention: Dict[str, Union[str, dict]] = None,
                      positive_offsets: List[Tuple[int, int]] = None) -> bool:
    """Check whether a given mention is argument or not.
    Check whether a given mention is argument or not. If it is an argument, we have to␣
↪exclude it from the negative
    arguments list.
    Args:
        mention (`Dict[str, Union[str, dict]]`):
            The mention that contains the word, position and other meta information like␣
↪id, etc.
        positive_offsets (`List[Tuple[int, int]]`):
            A list that contains the offsets of all the ground truth arguments.
    Returns:
        is_argument(`bool`):
            A flag that indicates whether the mention is an argument or not.
    """
    is_argument = False
    if positive_offsets:
        mention_set = set(range(mention["position"][0], mention["position"][1]))
        for pos_offset in positive_offsets:
            pos_set = set(range(pos_offset[0], pos_offset[1]))
            if not pos_set.isdisjoint(mention_set):
                is_argument = True
                break
    return is_argument
```

6.12.11 get_negative_argument_candidates

Obtains the negative candidate arguments, which are not included in the actual arguments list, for the specified trigger. The unified evaluation considers prediction of each token that is possibly an argument (EAE candidate).

Args:

- `item`:`` A single item of the training/valid/test data.
- `positive_offsets`: A list that contains the offsets of all the ground truth arguments.

Returns:

- `candidates`: A list of dictionary that contains the possible arguments.
- `label_names`: A list of string contains the ground truth label for each possible argument.

```
def get_negative_argument_candidates(item: Dict[str, Union[str, List[dict]]],
                                     positive_offsets: List[Tuple[int, int]] = None,
                                     ) -> List[Dict[str, Union[str, dict]]]:
    """Obtain the negative candidate arguments for each trigger in the event argument␣
↪extraction (EAE) task.
    Obtain the negative candidate arguments, which are not included in the actual␣
↪arguments list, for the specified
    trigger. The unified evaluation considers prediction of each token that is possibly␣
↪an argument (EAE candidate).
    Args:
        item (`Dict[str, Union[str, List[dict]]]`):
```

(continues on next page)

(continued from previous page)

```
            A single item of the training/valid/test data.
        positive_offsets (`List[Tuple[int, int]]`):
            A list that contains the offsets of all the ground truth arguments.
    Returns:
        candidates(`List[dict]`), label_names (`List[str]`):
            candidates: A list of dictionary that contains the possible arguments.
            label_names: A list of string contains the ground truth label for each␣
↪possible argument.
    """
    if "entities" in item:
        neg_arg_candidates = []
        for entity in item["entities"]:
            ent_is_arg = any([check_is_argument(men, positive_offsets) for men in entity[
↪"mentions"]])
            neg_arg_candidates.extend([] if ent_is_arg else entity["mentions"])
    else:
        neg_arg_candidates = item["negative_triggers"]
    return neg_arg_candidates
```

6.12.12 get_eae_candidates

Obtains the candidate arguments for each trigger in the event argument extraction (EAE) task. The unified evaluation considers prediction of each token that is possibly an argument (EAE candidate). And the EAE

Args:

- `item`: A single item of the training/valid/test data.
- `trigger`: A single item of trigger in the item.

Returns:

- `candidates`: A list of dictionary that contains the possible arguments.
- `label_names`: A list of string contains the ground truth label for each possible argument.task requires the model to predict the argument role of each candidate given a specific trigger.

```
def get_eae_candidates(item: Dict[str, Union[str, List[dict]]],
                       trigger: Dict[str, Union[str, dict]]) -> Tuple[List[dict],␣
↪List[str]]:
    """Obtain the candidate arguments for each trigger in the event argument extraction␣
↪(EAE) task.
    The unified evaluation considers prediction of each token that is possibly an␣
↪argument (EAE candidate). And the EAE
    task requires the model to predict the argument role of each candidate given a␣
↪specific trigger.
    Args:
        item (`Dict[str, Union[str, List[dict]]]`):
            A single item of the training/valid/test data.
        trigger (`Dict[str, Union[str, List[dict]]]`):
            A single item of trigger in the item.
    Returns:
        candidates(`List[dict]`), label_names (`List[str]`):
            candidates: A list of dictionary that contains the possible arguments.
```

(continues on next page)

(continued from previous page)

```
            label_names: A list of string contains the ground truth label for each␣
↪possible argument.
    """
    candidates = []
    positive_offsets = []
    label_names = []
    if "arguments" in trigger:
        arguments = sorted(trigger["arguments"], key=lambda a: a["role"])
        for argument in arguments:
            for mention in argument["mentions"]:
                label_names.append(argument["role"])
                candidates.append(mention)
                positive_offsets.append(mention["position"])

    neg_arg_candidates = get_negative_argument_candidates(item, positive_
↪offsets=positive_offsets)

    for neg in neg_arg_candidates:
        is_argument = check_is_argument(neg, positive_offsets)
        if not is_argument:
            label_names.append("NA")
            candidates.append(neg)

    return candidates, label_names
```

6.12.13 get_event_preds

Loads the event detection predictions of each token for event argument extraction. The Event Argument Extraction task requires the event detection predictions. If the event prediction file exists, we use the predictions by the event detection model. Otherwise, we use the golden event type for each token.

Args:

- `pred_file`: The file that contains the event detection predictions for each token.

Returns:

- `event_preds`: A list of the predicted event types for each token.

```
def get_event_preds(pred_file: Union[str, Path]) -> List[str]:
    """Load the event detection predictions of each token for event argument extraction.
    The Event Argument Extraction task requires the event detection predictions. If the␣
↪event prediction file exists,
    we use the predictions by the event detection model. Otherwise, we use the golden␣
↪event type for each token.
    Args:
        pred_file (`Union[str, Path]`):
            The file that contains the event detection predictions for each token.
    Returns:
        event_preds (`List[str]`):
            A list of the predicted event types for each token.
    """
    if pred_file is not None and os.path.exists(pred_file):
```

(continues on next page)

(continued from previous page)

```
        event_preds = json.load(open(pred_file))
    else:
        event_preds = None
        logger.info("Load {} failed, using golden triggers for EAE evaluation".
↪format(pred_file))

    return event_preds
```

6.12.14 get_plain_label

This function is used in the Seq2seq paradigm that the model has to generate the event types and argument roles. Some event types and argument roles are formatted, such as `Attack.Time-Start`, we convert them in to a plain one, like `timestart`, by removing the event type in the front and shifting upper case to lower case.

```
def get_plain_label(input_label: str) -> str:
    """Convert the formatted original event type or argument role to a plain one.
    This function is used in the Seq2seq paradigm that the model has to generate the␣
↪event types and argument roles.
    Some  event types and argument roles are formatted, such as `Attack.Time-Start`, we␣
↪convert them in to a plain
    one, like `timestart`, by removing the event type in the front and shifting upper␣
↪case to lower case.
    Args:
        input_label (`str`):
            The original label with format.
    Returns:
        return_label (`str`):
            The plain label without format.
    """
    if input_label == "NA":
        return input_label

    return_label = "".join("".join(input_label.split(".")[-1].split("-")).split("_")).
↪lower()

    return return_label
```

6.12.15 str_full_to_half

Converts a full-width string to a half-width one. The corpus of some datasets contain full-width strings, which may bring about unexpected error for mapping the tokens to the original input sentence.

Args:

- `ustring`: Original string.

Returns:

- `rstring`: Output string with the full-width tokens converted

```
def str_full_to_half(ustring: str) -> str:
    """Convert a full-width string to a half-width one.
```

(continues on next page)

(continued from previous page)

```
    The corpus of some datasets contain full-width strings, which may bring about␣
↪unexpected error for mapping the
    tokens to the original input sentence.
    Args:
        ustring(`str`):
            Original string.
    Returns:
        rstring (`str`):
            Output string with the full-width tokens converted
    """
    rstring = ""
    for uchar in ustring:
        inside_code = ord(uchar)
        if inside_code == 12288:   # full width space
            inside_code = 32
        elif 65281 <= inside_code <= 65374:    # full width char (exclude space)
            inside_code -= 65248
        rstring += chr(inside_code)
    return rstring
```

6.13 Backbone

```
import numpy as np
import os
import pdb
import torch
import torch.nn as nn
import torch.nn.functional as F

from audioop import bias
from typing import List, Optional, Tuple, Union
from unicodedata import bidirectional

from transformers import BertModel, BertTokenizerFast
from transformers import RobertaModel, RobertaTokenizerFast
from transformers import T5ForConditionalGeneration, T5TokenizerFast
from transformers import MT5ForConditionalGeneration
from transformers import BartForConditionalGeneration, BartTokenizerFast
from transformers.utils import ModelOutput

from ..input_engineering.whitespace_tokenizer import WordLevelTokenizer, load_vocab,␣
↪VOCAB_FILES_NAMES
```

6.13.1 get_backbone

Obtains the backbone model and tokenizer. The backbone model is selected from BERT, RoBERTa, T5, MT5, CNN, and LSTM, corresponding to a distinct tokenizer.

Args:

- `model_type`: A string indicating the model being used as the backbone network.
- `model_name_or_path`: A string indicating the path of the pre-trained model.
- `tokenizer_name`: A string indicating the repository name for the model in the hub or a path to a local folder.
- `markers`: A list of strings to mark the start and end position of event triggers and argument mentions.
- `model_args`: The pre-defined arguments for the model.
- `new_tokens`: A list of strings indicating new tokens to be added to the tokenizer's vocabulary.

Returns:

- `model`: The backbone model, which is selected from BERT, RoBERTa, T5, MT5, CNN, and LSTM.
- `tokenizer`: The tokenizer proposed for the tokenization process, corresponds to the backbone model.
- `config`: The configurations of the model.

```
def get_backbone(model_type: str,
                 model_name_or_path: str,
                 tokenizer_name: str,
                 markers: List[str],
                 model_args: Optional = None,
                 new_tokens: Optional[List[str]] = []):
    """Obtains the backbone model and tokenizer.
    Obtains the backbone model and tokenizer. The backbone model is selected from BERT,␣
↪RoBERTa, T5, MT5, CNN, and LSTM,
    corresponding to a distinct tokenizer.
    Args:
        model_type (`str`):
            A string indicating the model being used as the backbone network.
        model_name_or_path (`str`):
            A string indicating the path of the pre-trained model.
        tokenizer_name (`str`):
            A string indicating the repository name for the model in the hub or a path␣
↪to a local folder.
        markers (`List[str]`):
            A list of strings to mark the start and end position of event triggers and␣
↪argument mentions.
        model_args (`optional`, defaults to `None`):
            The pre-defined arguments for the model.  TODO: The data type of `model_args`␣
↪should be configured.
        new_tokens (`List[str]`, `optional`, defaults to []):
            A list of strings indicating new tokens to be added to the tokenizer's␣
↪vocabulary.
    Returns:
        model (`Union[BertModel, RobertaModel, T5ForConditionalGeneration, CNN, LSTM]`):
            The backbone model, which is selected from BERT, RoBERTa, T5, MT5, CNN, and␣
↪LSTM.
        tokenizer (`str`):
```

(continues on next page)

(continued from previous page)

```
            The tokenizer proposed for the tokenization process, corresponds to the␣
↪backbone model.
        config:
            The configurations of the model.         TODO: The data type of `config`␣
↪should be configured.
    """
    if model_type == "bert":
        model = BertModel.from_pretrained(model_name_or_path)
        tokenizer = BertTokenizerFast.from_pretrained(tokenizer_name, never_
↪split=markers)
    elif model_type == "roberta":
        model = RobertaModel.from_pretrained(model_name_or_path)
        tokenizer = RobertaTokenizerFast.from_pretrained(tokenizer_name, never_
↪split=markers, add_prefix_space=True)
    elif model_type == "bart":
        model = BartForConditionalGeneration.from_pretrained(model_name_or_path)
        tokenizer = BartTokenizerFast.from_pretrained(tokenizer_name, never_
↪split=markers, add_prefix_space=True)
    elif model_type == "t5":
        model = T5ForConditionalGeneration.from_pretrained(model_name_or_path)
        tokenizer = T5TokenizerFast.from_pretrained(tokenizer_name, never_split=markers)
    elif model_type == "mt5":
        model = MT5ForConditionalGeneration.from_pretrained(model_name_or_path)
        tokenizer = T5TokenizerFast.from_pretrained(tokenizer_name, never_split=markers)
    elif model_type == "cnn":
        tokenizer = WordLevelTokenizer.from_pretrained(model_args.vocab_file)
        model = CNN(model_args, len(tokenizer))
    elif model_type == 'lstm':
        tokenizer = WordLevelTokenizer.from_pretrained(model_args.vocab_file)
        model = LSTM(model_args, len(tokenizer))
    else:
        raise ValueError("No such model. %s" % model_type)

    for token in new_tokens:
        tokenizer.add_tokens(token, special_tokens=True)
    if len(new_tokens) > 0:
        model.resize_token_embeddings(len(tokenizer))

    config = model.config
    return model, tokenizer, config
```

6.13.2 WordEmbedding

Base class for word embedding, in which the word embeddings are loaded from a pre-trained word embedding file and could be resized into a distinct size.

Attributes:

- `word_embeddings`: A tensor representing the word embedding matrix, whose dimension is (number of tokens) * (embedding dimension).
- `position_embeddings`: A tensor representing the position embedding matrix, whose dimension is (number of positions) * (embedding dimension).

- dropout: An nn.Dropout layer for the dropout operation with the pre-defined dropout rate.

```
class WordEmbedding(nn.Module):
    """Base class for word embedding.
    Base class for word embedding, in which the word embeddings are loaded from a pre-
↪trained word embedding file and
    could be resized into a distinct size.
    Attributes:
        word_embeddings (`torch.Tensor`):
            A tensor representing the word embedding matrix, whose dimension is (number␣
↪of tokens) * (embedding
            dimension).
        position_embeddings (`torch.Tensor`):
            A tensor representing the position embedding matrix, whose dimension is␣
↪(number of positions) * (embedding
            dimension).
        dropout (`nn.Dropout`):
            An `nn.Dropout` layer for the dropout operation with the pre-defined dropout␣
↪rate.
    """
    def __init__(self,
                 config,
                 vocab_size: int) -> None:
        """Constructs a `WordEmbedding`."""
        super(WordEmbedding, self).__init__()
        if not os.path.exists(os.path.join(config.vocab_file, VOCAB_FILES_NAMES["vocab_
↪file"].replace("txt", "npy"))):
            embeddings = load_vocab(os.path.join(config.vocab_file, VOCAB_FILES_NAMES[
↪"vocab_file"]),
                                    return_embeddings=True)
            np.save(os.path.join(config.vocab_file, VOCAB_FILES_NAMES["vocab_file"].
↪replace("txt", "npy")), embeddings)
        else:
            embeddings = np.load(os.path.join(config.vocab_file, VOCAB_FILES_NAMES[
↪"vocab_file"].replace("txt", "npy")))
        self.word_embeddings = nn.Embedding.from_pretrained(torch.tensor(embeddings),␣
↪freeze=False, padding_idx=0)
        self.position_embeddings = nn.Embedding(config.num_position_embeddings, config.
↪position_embedding_dim)
        self.register_buffer("position_ids", torch.arange(config.num_position_
↪embeddings).expand((1, -1)))
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.resize_token_embeddings(vocab_size)

    def resize_token_embeddings(self,
                                vocab_size: int) -> None:
        """Resizes the embeddings from the pre-trained embedding dimension to pre-
↪defined embedding size."""
        if len(self.word_embeddings.weight) > vocab_size:
            raise ValueError("Invalid vocab_size %d < original vocab size." % vocab_size)
        elif len(self.word_embeddings.weight) == vocab_size:
            pass
        else:
```

(continues on next page)

(continued from previous page)

```
            num_added_token = vocab_size - len(self.word_embeddings.weight)
            embedding_dim = self.word_embeddings.weight.shape[1]
            average_embedding = torch.mean(self.word_embeddings.weight, dim=0).expand(1,␣
↪-1)
            self.word_embeddings.weight = nn.Parameter(torch.cat(
                (
                    self.word_embeddings.weight.data,
                    average_embedding.expand(num_added_token, embedding_dim)
                )
            ))

    def forward(self,
                input_ids: torch.Tensor,
                position_ids: Optional[torch.Tensor] = None) -> torch.Tensor:
        """Generates word embeddings and position embeddings and concatenates them␣
↪together."""
        input_shape = input_ids.size()
        batch_size, seq_length = input_shape[0], input_shape[1]
        if position_ids is None:
            position_ids = self.position_ids[:, :seq_length].expand(batch_size, seq_
↪length)
        # input embeddings & position embeddings
        inputs_embeds = self.word_embeddings(input_ids)
        position_embeds = self.position_embeddings(position_ids)
        embeds = torch.cat((inputs_embeds, position_embeds), dim=-1)
        embeds = self.dropout(embeds)
        return embeds
```

6.13.3 Output

A class for the model's output, containing the hidden states of the sequence.

```
class Output(ModelOutput):
    """A class for the model's output, containing the hidden states of the sequence."""
    last_hidden_state: torch.Tensor = None
```

6.13.4 CNN

A Convolutional Neural Network (CNN) as the backbone model, which comprises a 1-d convolutional layer, a relu activation layer, and a dropout layer. The last hidden state of the model would be returned.

Attributes:

- `config`: The configurations of the model.
- `embedding`: A `WordEmbedding` instance representing the embedding matrices of tokens and positions.
- `conv`: A `nn.Conv1d` layer representing 1-dimensional convolution layer.
- `dropout`: An `nn.Dropout` layer for the dropout operation with the pre-defined dropout rate.

```
class CNN(nn.Module):
    """A Convolutional Neural Network (CNN) as backbone model.
    A Convolutional Neural Network (CNN) as the backbone model, which comprises a 1-d␣
↪convolutional layer, a relu
    activation layer, and a dropout layer. The last hidden state of the model would be␣
↪returned.
    Attributes:
        config:
            The configurations of the model.
        embedding (`WordEmbedding`):
            A `WordEmbedding` instance representing the embedding matrices of tokens and␣
↪positions.
        conv (`nn.Conv1d`):
            A `nn.Conv1d` layer representing 1-dimensional convolution layer.
        dropout (`nn.Dropout`):
            An `nn.Dropout` layer for the dropout operation with the pre-defined dropout␣
↪rate.
    """
    def __init__(self,
                 config,
                 vocab_size: int,
                 kernel_size: Optional[int] = 3,
                 padding_size: Optional[int] = 1) -> None:
        """Constructs a `CNN`."""
        super(CNN, self).__init__()
        self.config = config
        self.embedding = WordEmbedding(config, vocab_size)
        self.conv = nn.Conv1d(config.word_embedding_dim + config.position_embedding_dim,
                              config.hidden_size,
                              kernel_size,
                              padding=padding_size)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def resize_token_embeddings(self,
                                vocab_size: int) -> None:
        """Resizes the embeddings from the pre-trained embedding dimension to pre-
↪defined embedding size."""
        self.embedding.resize_token_embeddings(vocab_size)

    def forward(self,
                input_ids: torch.Tensor,
                attention_mask: torch.Tensor,
                token_type_ids: torch.Tensor,
                return_dict: Optional[bool] = True) -> Union[Output, Tuple[torch.
↪Tensor]]:
        """Conducts the convolution operations on the input tokens."""
        x = self.embedding(input_ids)  # (B, L, H)
        x = x.transpose(1, 2)  # (B, H, L)
        x = F.relu(self.conv(x).transpose(1, 2))  # (B, H, L)
        x = self.dropout(x)
        if return_dict:
            return Output(last_hidden_state=x)
        else:
```

(continues on next page)

(continued from previous page)

```
            return x
```

6.13.5 LSTM

A bidirectional two-layered Long Short-Term Memory (LSTM) network as the backbone model, which utilizes recurrent computations for hidden states and addresses long-term information preservation and short-term input skipping using gated memory cells.

Attributes:

- `config`: The configurations of the model.
- `embedding`: A `WordEmbedding` instance representing the embedding matrices of tokens and positions.
- `rnn`: A `nn.LSTM` layer representing a bi-directional two-layered LSTM network, which manipulates the word embedding and position embedding for recurrent computations.
- `dropout`: An `nn.Dropout` layer for the dropout operation with the pre-defined dropout rate.

```
class LSTM(nn.Module):
    """A Long Short-Term Memory (LSTM) network as backbone model.
    A bidirectional two-layered Long Short-Term Memory (LSTM) network as the backbone␣
↪model, which utilizes recurrent
    computations for hidden states and addresses long-term information preservation and␣
↪short-term input skipping
    using gated memory cells.
    Attributes:
        config:
            The configurations of the model.
        embedding (`WordEmbedding`):
            A `WordEmbedding` instance representing the embedding matrices of tokens and␣
↪positions.
        rnn (`nn.LSTM`):
            A `nn.LSTM` layer representing a bi-directional two-layered LSTM network,␣
↪which manipulates the word
            embedding and position embedding for recurrent computations.
        dropout (`nn.Dropout`):
            An `nn.Dropout` layer for the dropout operation with the pre-defined dropout␣
↪rate.
       """
    def __init__(self,
                 config,
                 vocab_size: int) -> None:
        """Constructs a `LSTM`."""
        super(LSTM, self).__init__()
        self.config = config
        self.embedding = WordEmbedding(config, vocab_size)
        self.rnn = nn.LSTM(config.word_embedding_dim + config.position_embedding_dim,
                           config.hidden_size,
                           num_layers=2,
                           bidirectional=True,
                           batch_first=True,
                           dropout=config.hidden_dropout_prob)
```

(continues on next page)

(continued from previous page)

```
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def resize_token_embeddings(self,
                                vocab_size: int) -> None:
        """Resizes the embeddings from the pre-trained embedding dimension to pre-
↪defined embedding size."""
        self.embedding.resize_token_embeddings(vocab_size)

    def prepare_pack_padded_sequence(self,
                                     input_ids: torch.Tensor,
                                     input_lengths: torch.Tensor,
                                     descending: Optional[bool] = True):
        """Sorts the input sequences based on their length."""
        sorted_input_lengths, indices = torch.sort(input_lengths, descending=descending)
        _, desorted_indices = torch.sort(indices, descending=False)
        sorted_input_ids = input_ids[indices]
        return sorted_input_ids, sorted_input_lengths, desorted_indices

    def forward(self,
                input_ids: torch.Tensor,
                attention_mask: torch.Tensor,
                token_type_ids: torch.Tensor,
                return_dict: Optional[bool] = True):
        """Forward propagation of a LSTM network."""
        # add a pseudo input of max_length
        add_pseudo = max(torch.sum(attention_mask, dim=-1).tolist()) != input_ids.
↪shape[1]
        if add_pseudo:
            input_ids = torch.cat((torch.zeros_like(input_ids[0]).unsqueeze(0), input_
↪ids), dim=0)
            attention_mask = torch.cat((torch.ones_like(attention_mask[0]).unsqueeze(0),␣
↪attention_mask), dim=0)
        input_length = torch.sum(attention_mask, dim=-1).to(torch.long)
        sorted_input_ids, sorted_seq_length, desorted_indices = self.prepare_pack_padded_
↪sequence(input_ids,
                                                                                    ␣
↪          input_length)
        x = self.embedding(sorted_input_ids)  # (B, L, H)
        packed_embedded = nn.utils.rnn.pack_padded_sequence(x, sorted_seq_length.cpu(),␣
↪batch_first=True)
        self.rnn.flatten_parameters()
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        output, _ = nn.utils.rnn.pad_packed_sequence(packed_output, batch_first=True)
        x = output[desorted_indices]
        if add_pseudo:
            x = self.dropout(x)[1:, :, :]  # remove the pseudo input
        else:
            x = self.dropout(x)

        if return_dict:
            return Output(
                last_hidden_state=x
```

(continues on next page)

(continued from previous page)

```
            )
        else:
            return (x)
```

6.14 Model

```
import os
import torch
import torch.nn as nn

from typing import Dict, Optional, Union
from transformers import BartForConditionalGeneration, MT5ForConditionalGeneration,␣
↪T5ForConditionalGeneration

from OmniEvent.aggregation.aggregation import get_aggregation, aggregate
from OmniEvent.head.head import get_head
from OmniEvent.head.classification import LinearHead
from OmniEvent.arguments import (
    ModelArguments,
    DataArguments,
    TrainingArguments,
    ArgumentParser
)
from OmniEvent.utils import check_web_and_convert_path
```

6.14.1 get_model

Returns the model proposed to be utilized for training and prediction based on the pre-defined paradigm. The paradigms of training and prediction include token classification, sequence labeling, Sequence-to-Sequence (Seq2Seq), and Machine Reading Comprehension (MRC).

Args:

- `model_args`: The arguments of the model for training and prediction.
- `backbone`: The backbone model obtained from the `get_backbone()` method.

Returns:

- The model method/class proposed to be utilized for training and prediction.

```
def get_model(model_args,
              backbone):
    """Returns the model proposed to be utilized for training and prediction.
    Returns the model proposed to be utilized for training and prediction based on the␣
↪pre-defined paradigm. The
    paradigms of training and prediction include token classification, sequence labeling,
↪ Sequence-to-Sequence
    (Seq2Seq), and Machine Reading Comprehension (MRC).
    Args:
        model_args:
```

(continues on next page)

(continued from previous page)

```
            The arguments of the model for training and prediction.
        backbone:
            The backbone model obtained from the `get_backbone()` method.
    Returns:
        The model method/class proposed to be utilized for training and prediction.
    """
    if model_args.paradigm == "token_classification":
        return ModelForTokenClassification(model_args, backbone)
    elif model_args.paradigm == "sequence_labeling":
        return ModelForSequenceLabeling(model_args, backbone)
    elif model_args.paradigm == "seq2seq":
        return backbone
    elif model_args.paradigm == "mrc":
        return ModelForMRC(model_args, backbone)
    else:
        raise ValueError("No such paradigm")
```

6.14.2 get_model_cls

```
def get_model_cls(model_args):
    if model_args.paradigm == "token_classification":
        return ModelForTokenClassification
    elif model_args.paradigm == "sequence_labeling":
        return ModelForSequenceLabeling
    elif model_args.paradigm == "seq2seq":
        if model_args.model_type == "bart":
            return BartForConditionalGeneration
        elif model_args.model_type == "t5":
            return T5ForConditionalGeneration
        elif model_args.model_type == "mt5":
            return MT5ForConditionalGeneration
        else:
            raise ValueError("Invalid model_type %s" % model_args.model_type)
    elif model_args.paradigm == "mrc":
        return ModelForMRC
    else:
        raise ValueError("No such paradigm")
```

6.14.3 BaseModel

```
class BaseModel(nn.Module):

    @classmethod
    def from_pretrained(cls, model_name_or_path: Union[str, os.PathLike], config=None,␣
↪**kwargs):
        if config is None:
            parser = ArgumentParser((ModelArguments, DataArguments, TrainingArguments))
            model_args, _, _ = parser.from_pretrained(model_name_or_path, **kwargs)
        path = check_web_and_convert_path(model_name_or_path, 'model')
```

(continues on next page)

(continued from previous page)

```
        model = get_model(model_args)
        model.load_state_dict(torch.load(path), strict=False)
        return model
```

6.14.4 ModelForTokenClassification

BERT model for token classification, which firstly obtains hidden states through the backbone model, then aggregates the hidden states through the aggregation method/class, and finally classifies each token to their corresponding label through token-wise linear transformation.

Attributes:

- `config`: The configurations of the model.
- `backbone`: The backbone network obtained from the `get_backbone()` method to output initial hidden states.
- `aggregation`: The aggregation method/class for aggregating the hidden states output by the backbone network.
- `cls_head`: A `ClassificationHead` instance classifying each token into its corresponding label through a token-wise linear transformation.

```
class ModelForTokenClassification(BaseModel):
    """BERT model for token classification.
    BERT model for token classification, which firstly obtains hidden states through the␣
↪backbone model, then aggregates
    the hidden states through the aggregation method/class, and finally classifies each␣
↪token to their corresponding
    label through token-wise linear transformation.
    Attributes:
        config:
            The configurations of the model.
        backbone:
            The backbone network obtained from the `get_backbone()` method to output␣
↪initial hidden states.
        aggregation:
            The aggregation method/class for aggregating the hidden states output by the␣
↪backbone network.
        cls_head (`ClassificationHead`):
            A `ClassificationHead` instance classifying each token into its corresponding␣
↪label through a token-wise
            linear transformation.
    """

    def __init__(self,
                 config,
                 backbone) -> None:
        """Constructs a `ModelForTokenClassification`."""
        super(ModelForTokenClassification, self).__init__()
        self.config = config
        self.backbone = backbone
        self.aggregation = get_aggregation(config)
        self.cls_head = get_head(config)
```

(continues on next page)

(continued from previous page)

```
    def forward(self,
                input_ids: torch.Tensor,
                attention_mask: torch.Tensor,
                token_type_ids: Optional[torch.Tensor] = None,
                trigger_left: Optional[torch.Tensor] = None,
                trigger_right: Optional[torch.Tensor] = None,
                argument_left: Optional[torch.Tensor] = None,
                argument_right: Optional[torch.Tensor] = None,
                labels: Optional[torch.Tensor] = None) -> Dict[str, torch.Tensor]:
        """Manipulates the inputs through a backbone, aggregation, and classification␣
↪module,
           returns the predicted logits and loss."""
        # backbone encode
        outputs = self.backbone(input_ids=input_ids,
                                attention_mask=attention_mask,
                                token_type_ids=token_type_ids,
                                return_dict=True)
        hidden_states = outputs.last_hidden_state
        # aggregation
        hidden_state = aggregate(self.config,
                                 self.aggregation,
                                 hidden_states,
                                 trigger_left,
                                 trigger_right,
                                 argument_left,
                                 argument_right)
        # classification
        logits = self.cls_head(hidden_state)
        # compute loss
        loss = None
        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)
        return dict(loss=loss, logits=logits)
```

6.14.5 ModelForSequenceLabeling

BERT model for sequence labeling, which firstly obtains hidden states through the backbone model, then labels each token to their corresponding label, and finally decodes the label through a Conditional Random Field (CRF) module.

Attributes:

- `config`: The configurations of the model.
- `backbone`: The backbone network obtained from the `get_backbone()` method to output initial hidden states.
- `cls_head`: A `ClassificationHead` instance classifying each token into its corresponding label through a token-wise linear transformation.

```
class ModelForSequenceLabeling(BaseModel):
    """BERT model for sequence labeling.
    BERT model for sequence labeling, which firstly obtains hidden states through the␣
↪backbone model, then labels each
```

(continues on next page)

(continued from previous page)

```
    token to their corresponding label, and finally decodes the label through a␣
↪Conditional Random Field (CRF) module.
    Attributes:
        config:
            The configurations of the model.
        backbone:
            The backbone network obtained from the `get_backbone()` method to output␣
↪initial hidden states.
        cls_head (`ClassificationHead`):
            A `ClassificationHead` instance classifying each token into its corresponding␣
↪label through a token-wise
            linear transformation.
    """

    def __init__(self,
                 config,
                 backbone) -> None:
        """Constructs a `ModelForSequenceLabeling`."""
        super(ModelForSequenceLabeling, self).__init__()
        self.config = config
        self.backbone = backbone
        self.cls_head = LinearHead(config)
        self.head = get_head(config)

    def forward(self,
                input_ids: torch.Tensor,
                attention_mask: torch.Tensor,
                token_type_ids: Optional[torch.Tensor] = None,
                labels: Optional[torch.Tensor] = None) -> Dict[str, torch.Tensor]:
        """Manipulates the inputs through a backbone, classification, and CRF module,
           returns the predicted logits and loss."""
        # backbone encode
        outputs = self.backbone(input_ids=input_ids,
                                attention_mask=attention_mask,
                                token_type_ids=token_type_ids,
                                return_dict=True)
        hidden_states = outputs.last_hidden_state
        # classification
        logits = self.cls_head(hidden_states)  # [batch_size, seq_length, num_labels]
        # compute loss
        loss = None
        if labels is not None:
            if self.config.head_type != "crf":
                loss_fn = nn.CrossEntropyLoss()
                loss = loss_fn(logits.reshape(-1, logits.shape[-1]), labels.reshape(-1))
            else:
                # CRF
                labels[:, 0] = 0
                mask = labels != -100
                tags = labels * mask.to(torch.long)
                loss = -self.head(emissions=logits,
                                  tags=tags,
```

(continues on next page)

(continued from previous page)

```
                                mask=mask,
                                reduction="token_mean")
                labels[:, 0] = -100
        else:
            if self.config.head_type == "crf":
                mask = torch.ones_like(logits[:, :, 0])
                preds = self.head.decode(emissions=logits, mask=mask)
                logits = torch.LongTensor(preds)

        return dict(loss=loss, logits=logits)
```

6.14.6 ModelForMRC

BERT model for Machine Reading Comprehension (MRC), which firstly obtains hidden states through the backbone model, then predicts the start and end logits of each mention type through an MRC head.

Attributes:

- `config`: The configurations of the model.
- `backbone`: The backbone network obtained from the `get_backbone()` method to output initial hidden states.
- `mrc_head`: A `ClassificationHead` instance classifying the hidden states into start and end logits of each mention type through token-wise linear transformations.

```
class ModelForMRC(BaseModel):
    """BERT Model for Machine Reading Comprehension (MRC).
    BERT model for Machine Reading Comprehension (MRC), which firstly obtains hidden␣
↪states through the backbone model,
    then predicts the start and end logits of each mention type through an MRC head.
    Attributes:
        config:
            The configurations of the model.
        backbone:
            The backbone network obtained from the `get_backbone()` method to output␣
↪initial hidden states.
        mrc_head (`MRCHead`):
            A `ClassificationHead` instance classifying the hidden states into start and␣
↪end logits of each mention type
            through token-wise linear transformations.
    """

    def __init__(self,
                 config,
                 backbone) -> None:
        """Constructs a `ModelForMRC`."""
        super(ModelForMRC, self).__init__()
        self.backbone = backbone
        self.mrc_head = get_head(config)

    def forward(self,
                input_ids: torch.Tensor,
                attention_mask: torch.Tensor,
```

(continues on next page)

(continued from previous page)

```
                token_type_ids: Optional[torch.Tensor] = None,
                argument_left: Optional[torch.Tensor] = None,
                argument_right: Optional[torch.Tensor] = None) -> Dict[str, torch.
↪Tensor]:
        """Manipulates the inputs through a backbone and a MRC head module,
           returns the predicted start and logits and loss."""
        # backbone encode
        outputs = self.backbone(input_ids=input_ids,
                                attention_mask=attention_mask,
                                token_type_ids=token_type_ids,
                                return_dict=True)
        hidden_states = outputs.last_hidden_state
        start_logits, end_logits = self.mrc_head(hidden_states)
        total_loss = None
        # pdb.set_trace()
        if argument_left is not None and argument_right is not None:
            # If we are on multi-GPU, split add a dimension
            if len(argument_left.size()) > 1:
                argument_left = argument_left.squeeze(-1)
            if len(argument_right.size()) > 1:
                argument_right = argument_right.squeeze(-1)
            # sometimes the start/end positions are outside our model inputs, we ignore␣
↪these terms
            ignored_index = start_logits.size(1)
            argument_left = argument_left.clamp(0, ignored_index)
            argument_right = argument_right.clamp(0, ignored_index)

            loss_fct = nn.CrossEntropyLoss(ignore_index=ignored_index)
            start_loss = loss_fct(start_logits, argument_left)
            end_loss = loss_fct(end_logits, argument_right)
            total_loss = (start_loss + end_loss) / 2

        logits = torch.cat((start_logits, end_logits), dim=-1)  # [batch_size, seq_
↪length*2]
        return dict(loss=total_loss, logits=logits)
```

6.15 Label Smoother Sum

Note: Copyright Text2Event from https://github.com/luyaojie/Text2Event.

Licensed under the MIT License.

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
# Copyright Text2Event from https://github.com/luyaojie/Text2Event.
# Licensed under the MIT License.

import torch
```

(continues on next page)

(continued from previous page)

```
from dataclasses import dataclass
from typing import Dict
```

6.15.1 SumLabelSmoother

A label-smoothing sum module operated on the pre-computed output from the model, which is a regularization technique that addresses the overfitting and overconfidence problems by adding some noises to decrease the weights of the actual samples when calculating losses.

Attributes:

- `epsilon`: A float variable indicating the label smoothing factor.
- `ignore_index`: An integer representing the index in the labels to ignore when computing the loss.

```
@dataclass
class SumLabelSmoother:
    """A label-smoothing sum module operated on the pre-computed output from the model.
    A label-smoothing sum module operated on the pre-computed output from the model, which is a regularization technique
    that addresses the overfitting and overconfidence problems by adding some noises to decrease the weights of the
    actual samples when calculating losses.
    Attributes:
        epsilon (`float`, `optional`, defaults to 0.1):
            A float variable indicating the label smoothing factor.
        ignore_index (`int`, `optional`, defaults to -100):
            An integer representing the index in the labels to ignore when computing the loss.
    """

    epsilon: float = 0.1
    ignore_index: int = -100

    def __call__(self,
                 model_output: Dict[str, torch.Tensor],
                 labels: torch.Tensor) -> float:
        """Conducts the label smoothing process."""
        logits = model_output["logits"] if isinstance(model_output, dict) else model_output[0]
        log_probs = -torch.nn.functional.log_softmax(logits, dim=-1)
        if labels.dim() == log_probs.dim() - 1:
            labels = labels.unsqueeze(-1)

        padding_mask = labels.eq(self.ignore_index)
        # In case the ignore_index is -100, the gather will fail, so we replace labels by 0. The padding_mask
        # will ignore them in any case.
        labels.clamp_min_(0)
        nll_loss = log_probs.gather(dim=-1, index=labels)
        smoothed_loss = log_probs.sum(dim=-1, keepdim=True)
```

(continues on next page)

(continued from previous page)

```
        nll_loss.masked_fill_(padding_mask, 0.0)
        smoothed_loss.masked_fill_(padding_mask, 0.0)

        # Take the mean over the label dimensions, then divide by the number of active␣
↪elements (i.e. not-padded):
        # num_active_elements = padding_mask.numel() - padding_mask.long().sum()
        nll_loss = nll_loss.sum()  # / num_active_elements
        smoothed_loss = smoothed_loss.sum()  # / (num_active_elements * log_probs.shape[-
↪1])
        eps_i = self.epsilon / log_probs.size(-1)
        return (1 - self.epsilon) * nll_loss + eps_i * smoothed_loss
```

6.16 Constraint Decoding

Note: Copyright Text2Event from https://github.com/luyaojie/Text2Event.

Licensed under the MIT License.

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
# Copyright Text2Event from https://github.com/luyaojie/Text2Event.
# Licensed under the MIT License.

from typing import List, Dict
import os
import re
import sys
sys.path.append("../")
import pdb

# debug = True if 'DEBUG' in os.environ else False
# debug_step = True if 'DEBUG_STEP' in os.environ else False
debug = False
debug_step = False

from ..input_engineering.seq2seq_processor import type_start, type_end


def get_label_name_tree(label_name_list, tokenizer, end_symbol='<end>'):
    sub_token_tree = dict()

    label_tree = dict()
    for typename in label_name_list:
        after_tokenized = tokenizer.encode(typename, add_special_tokens=False)
        label_tree[typename] = after_tokenized

    for _, sub_label_seq in label_tree.items():
        parent = sub_token_tree
        for value in sub_label_seq:
```

(continues on next page)

(continued from previous page)

```
            if value not in parent:
                parent[value] = dict()
            parent = parent[value]

        parent[end_symbol] = None

    return sub_token_tree


def match_sublist(the_list, to_match):
    """
    :param the_list: [1, 2, 3, 4, 5, 6, 1, 2, 4, 5]
    :param to_match: [1, 2]
    :return:
        [(0, 1), (6, 7)]
    """
    len_to_match = len(to_match)
    matched_list = list()
    for index in range(len(the_list) - len_to_match + 1):
        if to_match == the_list[index:index + len_to_match]:
            matched_list += [(index, index + len_to_match - 1)]
    return matched_list


def find_bracket_position(generated_text, _type_start, _type_end):
    bracket_position = {_type_start: list(), _type_end: list()}
    for index, char in enumerate(generated_text):
        if char in bracket_position:
            bracket_position[char] += [index]
    return bracket_position


def generated_search_src_sequence(generated, src_sequence, end_sequence_search_
↪tokens=None):
    # print(generated, src_sequence) if debug else None

    if len(generated) == 0:
        # It has not been generated yet. All SRC are valid.
        return src_sequence

    matched_tuples = match_sublist(the_list=src_sequence, to_match=generated)

    valid_token = list()
    for _, end in matched_tuples:
        next_index = end + 1
        if next_index < len(src_sequence):
            valid_token += [src_sequence[next_index]]

    if end_sequence_search_tokens:
        valid_token += end_sequence_search_tokens

    return valid_token
```

(continues on next page)

(continued from previous page)

```
def get_constraint_decoder(tokenizer, type_schema, source_prefix=None):
    return StruConstraintDecoder(tokenizer=tokenizer, type_schema=type_schema, source_
↪prefix=source_prefix)


class ConstraintDecoder:
    def __init__(self, tokenizer, source_prefix):
        self.tokenizer = tokenizer
        self.source_prefix = source_prefix
        self.source_prefix_tokenized = tokenizer.encode(source_prefix,
                                                        add_special_tokens=False) if␣
↪source_prefix else []

    def get_state_valid_tokens(self, src_sentence: List[str], tgt_generated: List[str]) -
↪> List[str]:
        pass

    def constraint_decoding(self, batch_id, src_sentence, tgt_generated):
        if self.source_prefix_tokenized:
            # Remove Source Prefix for Generation
            src_sentence = src_sentence[len(self.source_prefix_tokenized):]

        if debug:
            # if batch_id == 4:
            print("Src:", self.tokenizer.convert_ids_to_tokens(src_sentence))
            print("Tgt:", self.tokenizer.convert_ids_to_tokens(tgt_generated))
            print(batch_id, len(tgt_generated), tgt_generated)

        valid_token_ids = self.get_state_valid_tokens(
            src_sentence.tolist(),
            tgt_generated.tolist()
        )
        # pdb.set_trace()

        # if debug:
        #     print('=======================================')
        #     print('valid tokens:', self.tokenizer.convert_ids_to_tokens(
        #         valid_token_ids), valid_token_ids)
        #     if debug_step:
        #         input()

        # return self.tokenizer.convert_tokens_to_ids(valid_tokens)
        return valid_token_ids


class StruConstraintDecoder(ConstraintDecoder):
    def __init__(self, tokenizer, type_schema, *args, **kwargs):
        super().__init__(tokenizer, *args, **kwargs)
        self.tree_end = '<tree-end>'
        self.type_tree = get_label_name_tree(type_schema["role_list"],
```

(continues on next page)

(continued from previous page)

```
                                            tokenizer=self.tokenizer,
                                            end_symbol=self.tree_end)
        self.type_start = self.tokenizer.convert_tokens_to_ids([type_start])[0]
        self.type_end = self.tokenizer.convert_tokens_to_ids([type_end])[0]

    def check_state(self, tgt_generated):
        if tgt_generated[-1] == self.tokenizer.pad_token_id:
            return 'start', -1

        special_token_set = {self.type_start, self.type_end}
        special_index_token = list(
            filter(lambda x: x[1] in special_token_set, list(enumerate(tgt_generated))))

        last_special_index, last_special_token = special_index_token[-1]

        if len(special_index_token) == 1:
            if last_special_token != self.type_start:
                return 'error', 0

        bracket_position = find_bracket_position(
            tgt_generated, _type_start=self.type_start, _type_end=self.type_end)
        start_number, end_number = len(bracket_position[self.type_start]), len(
            bracket_position[self.type_end])

        if start_number == end_number:
            return 'end_generate', -1
        if start_number == end_number + 1:
            state = 'start_first_generation'
        elif start_number == end_number + 2:
            state = 'generate_span'
        else:
            state = 'error'
        return state, last_special_index

    def search_prefix_tree_and_sequence(self, generated: List[str], prefix_tree: Dict,␣
↪src_sentence: List[str],
                                        end_sequence_search_tokens: List[str] = None):
        """
        Generate Type Name + Text Span
        :param generated:
        :param prefix_tree:
        :param src_sentence:
        :param end_sequence_search_tokens:
        :return:
        """
        tree = prefix_tree
        for index, token in enumerate(generated):
            tree = tree[token]
            is_tree_end = len(tree) == 1 and self.tree_end in tree

            if is_tree_end:
                valid_token = generated_search_src_sequence(
```

(continues on next page)

(continued from previous page)

```
                    generated=generated[index + 1:],
                    src_sequence=src_sentence,
                    end_sequence_search_tokens=end_sequence_search_tokens,
                )
                return valid_token

            if self.tree_end in tree:
                try:
                    valid_token = generated_search_src_sequence(
                        generated=generated[index + 1:],
                        src_sequence=src_sentence,
                        end_sequence_search_tokens=end_sequence_search_tokens,
                    )
                    return valid_token
                except IndexError:
                    # Still search tree
                    continue

        valid_token = list(tree.keys())
        return valid_token

    def get_state_valid_tokens(self, src_sentence, tgt_generated):
        """
        :param src_sentence:
        :param tgt_generated:
        :return:
            List[str], valid token list
        """
        if self.tokenizer.eos_token_id in src_sentence:
            src_sentence = src_sentence[:src_sentence.index(
                self.tokenizer.eos_token_id)]

        state, index = self.check_state(tgt_generated)

        print("State: %s" % state) if debug else None

        if state == 'error':
            print("Error:")
            print("Src:", src_sentence)
            print("Tgt:", tgt_generated)
            valid_tokens = [self.tokenizer.eos_token_id]

        elif state == 'start':
            valid_tokens = [self.type_start]

        elif state == 'start_first_generation':
            valid_tokens = [self.type_start, self.type_end]

        elif state == 'generate_span':

            if tgt_generated[-1] == self.type_start:
                # Start Event Label
```

(continues on next page)

(continued from previous page)

```
                return list(self.type_tree.keys())

            elif tgt_generated[-1] == self.type_end:
                raise RuntimeError('Invalid %s in %s' %
                                   (self.type_end, tgt_generated))

            else:
                try:
                    valid_tokens = self.search_prefix_tree_and_sequence(
                        generated=tgt_generated[index + 1:],
                        prefix_tree=self.type_tree,
                        src_sentence=src_sentence,
                        end_sequence_search_tokens=[self.type_end]
                    )
                except:
                    print("Warning! An unexpected token is generated due to len(valid_
↪tokens) < num_beams.")
                    valid_tokens = [self.tokenizer.eos_token_id]

        elif state == 'end_generate':
            valid_tokens = [self.tokenizer.eos_token_id]

        else:
            raise NotImplementedError(
                'State `%s` for %s is not implemented.' % (state, self.__class__))

        print("Valid: %s" % valid_tokens) if debug else None
        return valid_tokens


class SpanConstraintDecoder(ConstraintDecoder):
    def __init__(self, tokenizer, type_schema, *args, **kwargs):
        super().__init__(tokenizer, *args, **kwargs)
        self.tree_end = '<tree-end>'
        self.type_tree = get_label_name_tree(type_schema["role_list"],
                                             tokenizer=self.tokenizer,
                                             end_symbol=self.tree_end)

    def check_state(self, tgt_generated, special_tokens_in_tgt):
        if tgt_generated[-1] == self.tokenizer.pad_token_id:
            return 'start', -1
        else:
            index = len(tgt_generated)
            for i, token in enumerate(tgt_generated):
                if token == special_tokens_in_tgt[-1]:
                    index = i+1
                    break
            return "generate", index

    def get_special_tokens(self, sentence):
        special_template = re.compile("<extra_id_\d+>")
        tokens = self.tokenizer.convert_ids_to_tokens(sentence)
```

(continues on next page)

(continued from previous page)

```
        special_tokens = []
        for token in tokens:
            if special_template.match(token) is not None:
                special_tokens.append(token)
        return self.tokenizer.convert_tokens_to_ids(special_tokens)

    def truncate_src(self, src_sentence):
        special_template = re.compile("<extra_id_\d+>")
        index = len(src_sentence)
        tokens = self.tokenizer.convert_ids_to_tokens(src_sentence)
        for i, token in enumerate(tokens):
            if special_template.match(token) is not None:
                index = i
                break
        return src_sentence[:index]

    def get_state_valid_tokens(self, src_sentence, tgt_generated):
        """
        :param src_sentence:
        :param tgt_generated:
        :return:
            List[str], valid token list
        """
        if self.tokenizer.eos_token_id in src_sentence:
            src_sentence = src_sentence[:src_sentence.index(
                self.tokenizer.eos_token_id)]

        special_tokens_in_src = self.get_special_tokens(src_sentence)
        special_tokens_in_gen = self.get_special_tokens(tgt_generated)

        # truncate
        src_sentence = self.truncate_src(src_sentence)

        state, index = self.check_state(tgt_generated, special_tokens_in_gen)


        if state == 'start':
            valid_tokens = [special_tokens_in_src[0]]

        elif state == 'generate':
            # valid_tokens = [self.type_start, self.type_end]
            valid_special_tokens = [self.tokenizer.convert_tokens_to_ids("[SEP]")]
            for token in special_tokens_in_src:
                if token not in special_tokens_in_gen:
                    valid_special_tokens.append(token)
            valid_tokens = generated_search_src_sequence(
                        generated=tgt_generated[index:],
                        src_sequence=src_sentence,
                        end_sequence_search_tokens=[self.tokenizer.eos_token_id],
                    )
            valid_tokens = valid_special_tokens + valid_tokens
        else:
```

(continues on next page)

(continued from previous page)

```
            raise NotImplementedError(
                'State `%s` for %s is not implemented.' % (state, self.__class__))

        print("Valid: %s" % valid_tokens) if debug else None
        return valid_tokens
```

6.17 Aggregation

```
import pdb
import torch
import torch.nn as nn
import torch.nn.functional as F

from typing import
```

6.17.1 get_aggregation

Obtains the aggregation method to be utilized based on the model's configurations. The aggregation methods include selecting the ``<cls>``s' representations, selecting the markers' representations, max-pooling, and dynamic multi-pooling.

Args:

- `config`: The configurations of the model.

Returns:

- The proposed method/class for the aggregation process.

```
def get_aggregation(config):
    """Obtains the aggregation method to be utilized.
    Obtains the aggregation method to be utilized based on the model's configurations.␣
↪The aggregation methods include
    selecting the `<cls>`s' representations, selecting the markers' representations, max-
↪pooling, and dynamic
    multi-pooling.
    Args:
        config:
            The configurations of the model.
    Returns:
        The proposed method/class for the aggregation process.
    """
    if config.aggregation == "cls":
        return select_cls
    elif config.aggregation == "marker":
        return select_marker
    elif config.aggregation == "dynamic_pooling":
        return DynamicPooling(config)
    elif config.aggregation == "max_pooling":
        return max_pooling
```

(continues on next page)

(continued from previous page)

```
    else:
        raise ValueError("Invaild %s aggregation method" % config.aggregation)
```

6.17.2 aggregate

Aggregates information to each position. The aggregation methods include selecting the "cls"s' representations, selecting the markers' representations, max-pooling, and dynamic multi-pooling.

Args:

- `config`: The configurations of the model.
- `method`: The method proposed to be utilized in the aggregation process.
- `hidden_states`: A tensor representing the hidden states output by the backbone model.
- `trigger_left`: A tensor indicating the left position of the triggers.
- `trigger_right`: A tensor indicating the right position of the triggers.
- `argument_left`: A tensor indicating the left position of the arguments.
- `argument_right`: A tensor indicating the right position of the arguments.

```
def aggregate(config,
              method,
              hidden_states: torch.Tensor,
              trigger_left: torch.Tensor,
              trigger_right: torch.Tensor,
              argument_left: torch.Tensor,
              argument_right: torch.Tensor):
    """Aggregates information to each position.
    Aggregates information to each position. The aggregation methods include selecting␣
↪the "cls"s' representations,
    selecting the markers' representations, max-pooling, and dynamic multi-pooling.
    Args:
        config:
            The configurations of the model.
        method:
            The method proposed to be utilized in the aggregation process.
            TODO: The data type of the variable `method` should be configured.
        hidden_states (`torch.Tensor`):
            A tensor representing the hidden states output by the backbone model.
        trigger_left (`torch.Tensor`):
            A tensor indicating the left position of the triggers.
        trigger_right (`torch.Tensor`):
            A tensor indicating the right position of the triggers.
        argument_left (`torch.Tensor`):
            A tensor indicating the left position of the arguments.
        argument_right (`torch.Tensor`):
            A tensor indicating the right position of the arguments.
    """
    if config.aggregation == "cls":
        return method(hidden_states)
    elif config.aggregation == "marker":
```

(continues on next page)

(continued from previous page)

```
        if argument_left is not None:
            return method(hidden_states, argument_left, argument_right)
        else:
            return method(hidden_states, trigger_left, trigger_right)
    elif config.aggregation == "max_pooling":
        return method(hidden_states)
    elif config.aggregation == "dynamic_pooling":
        return method(hidden_states, trigger_left, argument_left)
    else:
        raise ValueError("Invaild %s aggregation method" % config.aggregation)
```

6.17.3 max_pooling

Applies the max-pooling operation over the representation of the entire input sequence to capture the most useful information. The operation processes on the hidden states, which are output by the backbone model.

Args:

- ``hidden_states`: A tensor representing the hidden states output by the backbone model.

Returns:

- pooled_states: A tensor represents the max-pooled hidden states, containing the most useful information of the sequence.

```
def max_pooling(hidden_states: torch.Tensor) -> torch.Tensor:
    """Applies the max-pooling operation over the sentence representation.
    Applies the max-pooling operation over the representation of the entire input␣
↪sequence to capture the most useful
    information. The operation processes on the hidden states, which are output by the␣
↪backbone model.
    Args:
        hidden_states (`torch.Tensor`):
            A tensor representing the hidden states output by the backbone model.
    Returns:
        pooled_states (`torch.Tensor`):
            A tensor represents the max-pooled hidden states, containing the most useful␣
↪information of the sequence.
    """
    batch_size, seq_length, hidden_size = hidden_states.size()
    pooled_states = F.max_pool1d(input=hidden_states.transpose(1, 2), kernel_size=seq_
↪length).squeeze(-1)
    return pooled_states
```

6.17.4 select_cls

Returns the representations of each sequence's <cls> token by slicing the hidden state tensor output by the backbone model. The representations of the <cls> tokens contain general information of the sequences.

Args:

- `hidden_states`: A tensor represents the hidden states output by the backbone model.

Returns:

- A tensor containing the representations of each sequence's *<cls>* token.

```
def select_cls(hidden_states: torch.Tensor) -> torch.Tensor:
    """Returns the representations of the `<cls>` tokens.
    Returns the representations of each sequence's `<cls>` token by slicing the hidden␣
↪state tensor output by the
    backbone model. The representations of the `<cls>` tokens contain general information␣
↪of the sequences.
    Args:
        hidden_states (`torch.Tensor`):
            A tensor represents the hidden states output by the backbone model.
    Returns:
        `torch.Tensor`:
            A tensor containing the representations of each sequence's `<cls>` token.
    """
    return hidden_states[:, 0, :]
```

6.17.5 select_marker

Returns the representations of each sequence's marker tokens by slicing the hidden state tensor output by the backbone model.

Args:

- `hidden_states`: A tensor representing the hidden states output by the backbone model.
- `left`: A tensor indicates the left position of the markers.
- `right`: A tensor indicates the right position of the markers.

Returns:

- ``marker_output`: A tensor containing the representations of each sequence's marker tokens by concatenating their left and right token's representations.

```
def select_marker(hidden_states: torch.Tensor,
                  left: torch.Tensor,
                  right: torch.Tensor) -> torch.Tensor:
    """Returns the representations of the marker tokens.
    Returns the representations of each sequence's marker tokens by slicing the hidden␣
↪state tensor output by the
    backbone model.
    Args:
        hidden_states (`torch.Tensor`):
            A tensor representing the hidden states output by the backbone model.
        left (`torch.Tensor`):
```

(continues on next page)

(continued from previous page)

```
            A tensor indicates the left position of the markers.
        right (`torch.Tensor`):
            A tensor indicates the right position of the markers.
    Returns:
        marker_output (`torch.Tensor`):
            A tensor containing the representations of each sequence's marker tokens by␣
↪concatenating their left and
            right token's representations.
    """
    batch_size = hidden_states.size(0)
    batch_indice = torch.arange(batch_size)
    left_states = hidden_states[batch_indice, left.to(torch.long), :]
    right_states = hidden_states[batch_indice, right.to(torch.long), :]
    marker_output = torch.cat((left_states, right_states), dim=-1)
    return marker_output
```

6.17.6 DynamicPooling

Dynamic multi-pooling layer for Convolutional Neural Network (CNN), which is able to capture more valuable information within a sentence, particularly for some cases, such as multiple triggers are within a sentence and different argument candidate may play a different role with a different trigger.

Attributes:

- `activation`: An *nn.Tanh* layer representing the tanh activation function.
- `dropout`: An *nn.Dropout* layer for the dropout operation with the default dropout rate (0.5).

```
class DynamicPooling(nn.Module):
    """Dynamic multi-pooling layer for Convolutional Neural Network (CNN).
    Dynamic multi-pooling layer for Convolutional Neural Network (CNN), which is able to␣
↪capture more valuable
    information within a sentence, particularly for some cases, such as multiple␣
↪triggers are within a sentence and
    different argument candidate may play a different role with a different trigger.
    Attributes:
        dense (`nn.Linear`):
            TODO: The purpose of the linear layer should be configured.
        activation (`nn.Tanh`):
            An `nn.Tanh` layer representing the tanh activation function.
        dropout (`nn.Dropout`):
            An `nn.Dropout` layer for the dropout operation with the default dropout rate␣
↪(0.5).
    """
    def __init__(self,
                 config) -> None:
        """Constructs a `DynamicPooling`."""
        super(DynamicPooling, self).__init__()
        self.dense = nn.Linear(config.hidden_size*config.head_scale, config.hidden_
↪size*config.head_scale)
        self.activation = nn.Tanh()
        self.dropout = nn.Dropout()
```

(continues on next page)

(continued from previous page)

```
    def get_mask(self,
                 position: torch.Tensor,
                 batch_size: int,
                 seq_length: int,
                 device: str) -> torch.Tensor:
        """Returns the mask indicating whether the token is padded or not."""
        all_masks = []
        for i in range(batch_size):
            mask = torch.zeros((seq_length), dtype=torch.int16, device=device)
            mask[:int(position[i])] = 1
            all_masks.append(mask.to(torch.bool))
        all_masks = torch.stack(all_masks, dim=0)
        return all_masks

    def max_pooling(self,
                    hidden_states: torch.Tensor,
                    mask: torch.Tensor) -> torch.Tensor:
        """Conducts the max-pooling operation on the hidden states."""
        batch_size, seq_length, hidden_size = hidden_states.size()
        conved = hidden_states.transpose(1, 2)
        conved = conved.transpose(0, 1)
        states = (conved * mask).transpose(0, 1)
        states += torch.ones_like(states)
        pooled_states = F.max_pool1d(input=states, kernel_size=seq_length).contiguous().
↪view(batch_size, hidden_size)
        pooled_states -= torch.ones_like(pooled_states)
        return pooled_states

    def forward(self,
                hidden_states: torch.Tensor,
                trigger_position: torch.Tensor,
                argument_position: Optional[torch.Tensor] = None) -> torch.Tensor:
        """Conducts the dynamic multi-pooling process on the hidden states."""
        batch_size, seq_length = hidden_states.size()[:2]
        trigger_mask = self.get_mask(trigger_position, batch_size, seq_length, hidden_
↪states.device)
        if argument_position is not None:
            argument_mask = self.get_mask(argument_position, batch_size, seq_length,␣
↪hidden_states.device)
            left_mask = torch.logical_and(trigger_mask, argument_mask).to(torch.float32)
            middle_mask = torch.logical_xor(trigger_mask, argument_mask).to(torch.
↪float32)
            right_mask = 1 - torch.logical_or(trigger_mask, argument_mask).to(torch.
↪float32)
            # pooling
            left_states = self.max_pooling(hidden_states, left_mask)
            middle_states = self.max_pooling(hidden_states, middle_mask)
            right_states = self.max_pooling(hidden_states, right_mask)
            pooled_output = torch.cat((left_states, middle_states, right_states), dim=-1)
        else:
            left_mask = trigger_mask.to(torch.float32)
```

(continues on next page)

(continued from previous page)

```
            right_mask = 1 - left_mask
            left_states = self.max_pooling(hidden_states, left_mask)
            right_states = self.max_pooling(hidden_states, right_mask)
            pooled_output = torch.cat((left_states, right_states), dim=-1)

        return pooled_output
```

6.18 Classification Head

```
from .classification import LinearHead, MRCHead
from .crf import CRF
```

6.18.1 get_head

```
def get_head(config):
    if config.head_type == "linear":
        return LinearHead(config)
    elif config.head_type == "mrc":
        return MRCHead(config)
    elif config.head_type == "crf":
        return CRF(config.num_labels, batch_first=True)
    elif config.head_type in ["none", "None"] or config.head_type is None:
        return None
    else:
        raise ValueError("Invalid head_type %s in config" % config.head_type)
```

6.19 Classification Head

```
from turtle import forward
import torch
import torch.nn as nn
```

6.19.1 LinearHead

A token-wise classification head for classifying hidden states to label distributions through a linear transformation, selecting the label with the highest probability corresponding to each logit.

Attributes:

- `classifier`: An `nn.Linear` layer classifying each logit into its corresponding label.

```
class LinearHead(nn.Module):
    """A token-wise classification head for classifying the hidden states to label␣
↪distributions.
    A token-wise classification head for classifying hidden states to label␣
```

(continues on next page)

(continued from previous page)

```
↪distributions through a linear
    transformation, selecting the label with the highest probability corresponding to␣
↪each logit.
    Attributes:
        classifier (`nn.Linear`):
            An `nn.Linear` layer classifying each logit into its corresponding label.
    """
    def __init__(self, config):
        super(LinearHead, self).__init__()
        self.classifier = nn.Linear(config.hidden_size*config.head_scale, config.num_
↪labels)

    def forward(self,
                hidden_state: torch.Tensor) -> torch.Tensor:
        """Classifies hidden states to label distribution."""
        logits = self.classifier(hidden_state)
        return logits
```

6.19.2 MRCHead

A classification head for the Machine Reading Comprehension (MRC) paradigm, predicting the answer of each question corresponding to a mention type. The classifier returns two logits indicating the start and end position of each mention corresponding to the question.

Attributes:

- `qa_outputs`: An `nn.Linear` layer transforming the hidden states to two logits, indicating the start and end position of a given mention type.

```
class MRCHead(nn.Module):
    """A token-wise classification head for the Machine Reading Comprehension (MRC)␣
↪paradigm.
    A classification head for the Machine Reading Comprehension (MRC) paradigm,␣
↪predicting the answer of each question
    corresponding to a mention type. The classifier returns two logits indicating the␣
↪start and end position of each
    mention corresponding to the question.
    Attributes:
        qa_outputs (`nn.Linear`):
            An `nn.Linear` layer transforming the hidden states to two logits, indicating␣
↪the start and end position
            of a given mention type.
    """
    def __init__(self,
                 config) -> None:
        """Constructs a `MRCHead`."""
        super(MRCHead, self).__init__()
        self.qa_outputs = nn.Linear(config.hidden_size, 2)

    def forward(self,
                hidden_state: torch.Tensor):
        """The forward propagation of `MRCHead`."""
```

(continues on next page)

(continued from previous page)

```
        logits = self.qa_outputs(hidden_state)
        start_logits, end_logits = logits.split(1, dim=-1)
        start_logits = start_logits.squeeze(-1).contiguous()
        end_logits = end_logits.squeeze(-1).contiguous()
        return start_logits, end_logits
```

6.20 Conditional Random Field (CRF)

Note: Copyright pytorch-crf from https://github.com/kmkurn/pytorch-crf.

Licensed under the MIT License.

6.20.1 CRF

This module implements a Conditional Random Field (CRF). The forward computation of this class computes the log likelihood of the given sequence of tags and emission score tensor. This class also has `CRF.decode()` method which finds the best tag sequence given an emission score tensor using Viterbi algorithm.

Attributes:

- `num_tags`: An integer indicating the number of tags to be predicted.
- `batch_first`: A boolean variable indicating whether or not splitting the data in batches.
- `start_transitions`: An `nn.Parameter` matrix containing the start transition score tensor of size `(num_tags,)`.
- `end_transitions`: An `nn.Parameter` matrix containing the end transition score tensor of size `(num_tags,)`.
- `transitions`: An `nn.Parameter` matrix indicating the score tensor of size `(num_tags, num_tags)`.

```
class CRF(nn.Module):
    """Conditional Random Field (CRF) module.
    This module implements a Conditional Random Field (CRF). The forward computation of␣
↪this class computes the log
    likelihood of the given sequence of tags and emission score tensor. This class also␣
↪has `CRF.decode()` method which
    finds the best tag sequence given an emission score tensor using Viterbi algorithm.
    Attributes:
        num_tags (`int`):
            An integer indicating the number of tags to be predicted.
        batch_first (`bool`):
            A boolean variable indicating whether or not splitting the data in batches.
        start_transitions (`nn.Parameter`):
            An `nn.Parameter` matrix containing the start transition score tensor of size␣
↪`(num_tags,)`.
        end_transitions (`nn.Parameter`):
            An `nn.Parameter` matrix containing the end transition score tensor of size␣
↪`(num_tags,)`.
        transitions (`nn.Parameter`):
            An `nn.Parameter` matrix indicating the score tensor of size `(num_tags, num_
```

(continues on next page)

(continued from previous page)

```
↪tags)`.
    """

    def __init__(self,
                 num_tags: int,
                 batch_first: bool = False) -> None:
        """Constructs a `CRF`."""
        if num_tags <= 0:
            raise ValueError(f'invalid number of tags: {num_tags}')
        super().__init__()
        self.num_tags = num_tags
        self.batch_first = batch_first
        self.start_transitions = nn.Parameter(torch.empty(num_tags))
        self.end_transitions = nn.Parameter(torch.empty(num_tags))
        self.transitions = nn.Parameter(torch.empty(num_tags, num_tags))

        self.reset_parameters()

    def reset_parameters(self) -> None:
        """Initialize the transition parameters.
        The parameters will be initialized randomly from a uniform distribution between -
↪0.1 and 0.1.
        """
        nn.init.uniform_(self.start_transitions, -0.1, 0.1)
        nn.init.uniform_(self.end_transitions, -0.1, 0.1)
        nn.init.uniform_(self.transitions, -0.1, 0.1)

    def __repr__(self) -> str:
        """Displays the class name and the number of tags."""
        return f'{self.__class__.__name__}(num_tags={self.num_tags})'

    def forward(self,
                emissions: torch.Tensor,
                tags: torch.LongTensor,
                mask: Optional[torch.ByteTensor] = None,
                reduction: str = 'sum') -> torch.Tensor:
        """Compute the conditional log likelihood of a sequence of tags given emission␣
↪scores."""
        self._validate(emissions, tags=tags, mask=mask)
        if reduction not in ('none', 'sum', 'mean', 'token_mean'):
            raise ValueError(f'invalid reduction: {reduction}')
        if mask is None:
            mask = torch.ones_like(tags, dtype=torch.uint8)

        if self.batch_first:
            emissions = emissions.transpose(0, 1)
            tags = tags.transpose(0, 1)
            mask = mask.transpose(0, 1)

        # shape: (batch_size,)
        numerator = self._compute_score(emissions, tags, mask)
        # shape: (batch_size,)
```

(continues on next page)

(continued from previous page)

```
        denominator = self._compute_normalizer(emissions, mask)
        # shape: (batch_size,)
        llh = numerator - denominator

        if reduction == 'none':
            return llh
        if reduction == 'sum':
            return llh.sum()
        if reduction == 'mean':
            return llh.mean()
        assert reduction == 'token_mean'
        return llh.sum() / mask.type_as(emissions).sum()

    def decode(self, emissions: torch.Tensor,
               mask: Optional[torch.ByteTensor] = None) -> List[List[int]]:
        """Find the most likely tag sequence using Viterbi algorithm."""
        self._validate(emissions, mask=mask)
        if mask is None:
            mask = emissions.new_ones(emissions.shape[:2], dtype=torch.uint8)

        if self.batch_first:
            emissions = emissions.transpose(0, 1)
            mask = mask.transpose(0, 1)

        return self._viterbi_decode(emissions, mask)

    def _validate(self,
                  emissions: torch.Tensor,
                  tags: Optional[torch.LongTensor] = None,
                  mask: Optional[torch.ByteTensor] = None) -> None:
        """Validates the emission dimension and whether its slice satisfies tag number,␣
↪tag shape and mask shape."""
        if emissions.dim() != 3:
            raise ValueError(f'emissions must have dimension of 3, got {emissions.dim()}
↪')
        if emissions.size(2) != self.num_tags:
            raise ValueError(
                f'expected last dimension of emissions is {self.num_tags}, '
                f'got {emissions.size(2)}')

        if tags is not None:
            if emissions.shape[:2] != tags.shape:
                raise ValueError(
                    'the first two dimensions of emissions and tags must match, '
                    f'got {tuple(emissions.shape[:2])} and {tuple(tags.shape)}')

        if mask is not None:
            if emissions.shape[:2] != mask.shape:
                raise ValueError(
                    'the first two dimensions of emissions and mask must match, '
                    f'got {tuple(emissions.shape[:2])} and {tuple(mask.shape)}')
            no_empty_seq = not self.batch_first and mask[0].all()
```

(continues on next page)

(continued from previous page)

```
            no_empty_seq_bf = self.batch_first and mask[:, 0].all()
            if not no_empty_seq and not no_empty_seq_bf:
                raise ValueError('mask of the first timestep must all be on')

    def _compute_score(self,
                       emissions: torch.Tensor,
                       tags: torch.LongTensor,
                       mask: torch.ByteTensor) -> torch.Tensor:
        """Computes the score based on the emission and transition matrix."""
        # emissions: (seq_length, batch_size, num_tags)
        # tags: (seq_length, batch_size)
        # mask: (seq_length, batch_size)
        assert emissions.dim() == 3 and tags.dim() == 2
        assert emissions.shape[:2] == tags.shape
        assert emissions.size(2) == self.num_tags
        assert mask.shape == tags.shape
        assert mask[0].all()

        seq_length, batch_size = tags.shape
        mask = mask.type_as(emissions)

        # Start transition score and first emission
        # shape: (batch_size,)
        score = self.start_transitions[tags[0]]
        score += emissions[0, torch.arange(batch_size), tags[0]]

        for i in range(1, seq_length):
            # Transition score to next tag, only added if next timestep is valid (mask␣
↪== 1)
            # shape: (batch_size,)
            score += self.transitions[tags[i - 1], tags[i]] * mask[i]

            # Emission score for next tag, only added if next timestep is valid (mask ==␣
↪1)
            # shape: (batch_size,)
            score += emissions[i, torch.arange(batch_size), tags[i]] * mask[i]

        # End transition score
        # shape: (batch_size,)
        seq_ends = mask.long().sum(dim=0) - 1
        # shape: (batch_size,)
        last_tags = tags[seq_ends, torch.arange(batch_size)]
        # shape: (batch_size,)
        score += self.end_transitions[last_tags]

        return score

    def _compute_normalizer(self,
                            emissions: torch.Tensor,
                            mask: torch.ByteTensor) -> torch.Tensor:
        """Compute the log-sum-exp score."""
        # emissions: (seq_length, batch_size, num_tags)
```

(continues on next page)

(continued from previous page)

```
        # mask: (seq_length, batch_size)
        assert emissions.dim() == 3 and mask.dim() == 2
        assert emissions.shape[:2] == mask.shape
        assert emissions.size(2) == self.num_tags
        assert mask[0].all()

        seq_length = emissions.size(0)

        # Start transition score and first emission; score has size of
        # (batch_size, num_tags) where for each batch, the j-th column stores
        # the score that the first timestep has tag j
        # shape: (batch_size, num_tags)
        score = self.start_transitions + emissions[0]

        for i in range(1, seq_length):
            # Broadcast score for every possible next tag
            # shape: (batch_size, num_tags, 1)
            broadcast_score = score.unsqueeze(2)

            # Broadcast emission score for every possible current tag
            # shape: (batch_size, 1, num_tags)
            broadcast_emissions = emissions[i].unsqueeze(1)

            # Compute the score tensor of size (batch_size, num_tags, num_tags) where
            # for each sample, entry at row i and column j stores the sum of scores of␣
↪all
            # possible tag sequences so far that end with transitioning from tag i to␣
↪tag j
            # and emitting
            # shape: (batch_size, num_tags, num_tags)
            next_score = broadcast_score + self.transitions + broadcast_emissions

            # Sum over all possible current tags, but we're in score space, so a sum
            # becomes a log-sum-exp: for each sample, entry i stores the sum of scores of
            # all possible tag sequences so far, that end in tag i
            # shape: (batch_size, num_tags)
            next_score = torch.logsumexp(next_score, dim=1)

            # Set score to the next score if this timestep is valid (mask == 1)
            # shape: (batch_size, num_tags)
            score = torch.where(mask[i].unsqueeze(1), next_score, score)

        # End transition score
        # shape: (batch_size, num_tags)
        score += self.end_transitions

        # Sum (log-sum-exp) over all possible tags
        # shape: (batch_size,)
        return torch.logsumexp(score, dim=1)

    def _viterbi_decode(self,
                        emissions: torch.FloatTensor,
```

(continues on next page)

(continued from previous page)

```
                        mask: torch.ByteTensor) -> List[List[int]]:
        """Decodes the optimal path using Viterbi algorithm."""
        # emissions: (seq_length, batch_size, num_tags)
        # mask: (seq_length, batch_size)
        assert emissions.dim() == 3 and mask.dim() == 2
        assert emissions.shape[:2] == mask.shape
        assert emissions.size(2) == self.num_tags
        assert mask[0].all()

        seq_length, batch_size = mask.shape

        # Start transition and first emission
        # shape: (batch_size, num_tags)
        score = self.start_transitions + emissions[0]
        history = []

        # score is a tensor of size (batch_size, num_tags) where for every batch,
        # value at column j stores the score of the best tag sequence so far that ends
        # with tag j
        # history saves where the best tags candidate transitioned from; this is used
        # when we trace back the best tag sequence

        # Viterbi algorithm recursive case: we compute the score of the best tag sequence
        # for every possible next tag
        for i in range(1, seq_length):
            # Broadcast viterbi score for every possible next tag
            # shape: (batch_size, num_tags, 1)
            broadcast_score = score.unsqueeze(2)

            # Broadcast emission score for every possible current tag
            # shape: (batch_size, 1, num_tags)
            broadcast_emission = emissions[i].unsqueeze(1)

            # Compute the score tensor of size (batch_size, num_tags, num_tags) where
            # for each sample, entry at row i and column j stores the score of the best
            # tag sequence so far that ends with transitioning from tag i to tag j and␣
↪emitting
            # shape: (batch_size, num_tags, num_tags)
            next_score = broadcast_score + self.transitions + broadcast_emission

            # Find the maximum score over all possible current tag
            # shape: (batch_size, num_tags)
            next_score, indices = next_score.max(dim=1)

            # Set score to the next score if this timestep is valid (mask == 1)
            # and save the index that produces the next score
            # shape: (batch_size, num_tags)
            score = torch.where(mask[i].unsqueeze(1), next_score, score)
            history.append(indices)

        # End transition score
        # shape: (batch_size, num_tags)
```

(continues on next page)

(continued from previous page)

```
        score += self.end_transitions

        # Now, compute the best path for each sample

        # shape: (batch_size,)
        seq_ends = mask.long().sum(dim=0) - 1
        best_tags_list = []

        for idx in range(batch_size):
            # Find the tag which maximizes the score at the last timestep; this is our␣
↪best tag
            # for the last timestep
            _, best_last_tag = score[idx].max(dim=0)
            best_tags = [best_last_tag.item()]

            # We trace back where the best last tag comes from, append that to our best␣
↪tag
            # sequence, and trace it back again, and so on
            for hist in reversed(history[:seq_ends[idx]]):
                best_last_tag = hist[idx][best_tags[-1]]
                best_tags.append(best_last_tag.item())

            # Reverse the order because we start from the last timestep
            best_tags.reverse()
            best_tags_list.append(best_tags)

        return best_tags_list
```

6.21 Evaluation Metrics

```
import copy

import torch
import numpy as np

from sklearn.metrics import f1_score
from seqeval.metrics import f1_score as span_f1_score
from seqeval.scheme import IOB2
from typing import Tuple, Dict, List, Optional, Union

from ..input_engineering.mrc_converter import make_predictions, compute_mrc_F1_cls
from ..input_engineering.seq2seq_processor import extract_argument
```

6.21.1 compute_unified_micro_f1

Compute the F1 score of the unified evaluation on the converted word-level predictions.

Args:

- `label_names`: A list of ground truth labels of each word.
- `results`: A list of predicted event types or argument roles of each word.

Returns:

- `micro_f1`: The computation results of F1 score.

```
def compute_unified_micro_f1(label_names: List[str], results: List[str]) -> float:
    """Computes the F1 score of the converted word-level predictions.
    Compute the F1 score of the unified evaluation on the converted word-level␣
↪predictions.
    Args:
        label_names (`List[str]`):
            A list of ground truth labels of each word.
        results (`List[str]`):
            A list of predicted event types or argument roles of each word.
    Returns:
        micro_f1 (`float`):
            The computation results of  F1 score.
    """
    pos_labels = list(set(label_names))
    pos_labels.remove("NA")
    micro_f1 = f1_score(label_names, results, labels=pos_labels, average="micro") * 100.0
    return micro_f1
```

6.21.2 f1_score_overall

Computes the overall F1 score of the predictions based on the calculation of the overall precision and recall after counting the true predictions, in which both the prediction of mention and type are correct.

Args:

- `preds`: A list of strings indicating the prediction of labels from the model.
- `labels`: A list of strings indicating the actual labels obtained from the annotated dataset.

Returns: - `precision`, `recall`, and `f1`: Three float variables representing the computation results of precision, recall, and F1 score, respectively.

```
def f1_score_overall(preds: Union[List[str], List[tuple]],
                     labels: Union[List[str], List[tuple]]) -> Tuple[float, float,␣
↪float]:
    """Computes the overall F1 score of the predictions.
    Computes the overall F1 score of the predictions based on the calculation of the␣
↪overall precision and recall after
    counting the true predictions, in which both the prediction of mention and type are␣
↪correct.
    Args:
        preds (`Union[List[str], List[tuple]]`):
            A list of strings indicating the prediction of labels from the model.
```

(continues on next page)

(continued from previous page)

```
        labels (`Union[List[str], List[tuple]]`):
            A list of strings indicating the actual labels obtained from the annotated␣
↪dataset.
    Returns:
        precision (`float`), recall (`float`), and f1 (`float`):
            Three integers representing the computation results of precision, recall,␣
↪and F1 score, respectively.
    """
    true_pos = 0
    label_stack = copy.deepcopy(labels)
    for pred in preds:
        if pred in label_stack:
            true_pos += 1
            label_stack.remove(pred)  # one prediction can only be matched to one ground␣
↪truth.
    precision = true_pos / (len(preds)+1e-10)
    recall = true_pos / (len(labels)+1e-10)
    f1 = 2 * precision * recall / (precision + recall + 1e-10)
    return precision, recall, f1
```

6.21.3 compute_seq_F1

Computes the F1 score of the Sequence-to-Sequence (Seq2Seq) paradigm. The predictions of the model are firstly decoded into strings, then the overall F1 score of the prediction could be calculated.

Args:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.

Returns: - A dictionary containing the calculation result of the F1 score.

```
def compute_seq_F1(logits: np.ndarray,
                   labels: np.ndarray,
                   **kwargs) -> Dict[str, float]:
    """Computes the F1 score of the Sequence-to-Sequence (Seq2Seq) paradigm.
    Computes the F1 score of the  Sequence-to-Sequence (Seq2Seq) paradigm. The␣
↪predictions of the model are firstly
    decoded into strings, then the overall F1 score of the prediction could be␣
↪calculated.
    Args:
        logits (`List[int]`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels: (`List[str]`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
    Returns:
        `Dict[str: float]`:
            A dictionary containing the calculation result of the F1 score.
    """
    tokenizer = kwargs["tokenizer"]
```

(continues on next page)

(continued from previous page)

```
    training_args = kwargs["training_args"]
    decoded_preds = tokenizer.batch_decode(logits, skip_special_tokens=False)

    # Replace -100 in the labels as we can't decode them.
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=False)

    def clean_str(x_str):
        for to_remove_token in [tokenizer.eos_token, tokenizer.pad_token]:
            x_str = x_str.replace(to_remove_token, '')

        return x_str.strip()
    if training_args.task_name == "EAE":
        pred_types = training_args.data_for_evaluation["pred_types"]
        true_types = training_args.data_for_evaluation["true_types"]
        assert len(true_types) == len(decoded_labels)
        assert len(decoded_preds) == len(decoded_labels)
        pred_arguments, golden_arguments = [], []
        for i, (pred, label) in enumerate(zip(decoded_preds, decoded_labels)):
            pred = clean_str(pred)
            label = clean_str(label)
            pred_arguments.extend(extract_argument(pred, i, pred_types[i]))
            golden_arguments.extend(extract_argument(label, i, true_types[i]))
        precision, recall, micro_f1 = f1_score_overall(pred_arguments, golden_arguments)
    else:
        assert len(decoded_preds) == len(decoded_labels)
        pred_triggers, golden_triggers = [], []
        for i, (pred, label) in enumerate(zip(decoded_preds, decoded_labels)):
            pred = clean_str(pred)
            label = clean_str(label)
            pred_triggers.extend(extract_argument(pred, i, "NA"))
            golden_triggers.extend(extract_argument(label, i, "NA"))
        precision, recall, micro_f1 = f1_score_overall(pred_triggers, golden_triggers)
    return {"micro_f1": micro_f1*100}
```

6.21.4 select_start_position

Selects the preds and labels of the first sub-word token for each word. The `PreTrainedTokenizer` tends to split word into sub-word tokens, and we select the prediction of the first sub-word token as the prediction of this word.

Args:

- `preds`: The prediction ids of the input.
- `labels`: The label ids of the input.
- `merge`: Whether merge the predictions and labels into a one-dimensional list.

Return:

- `final_preds`, `final_labels`: The tuple of final predictions and labels.

```
def select_start_position(preds: np.ndarray,
                          labels: np.ndarray,
```

(continues on next page)

(continued from previous page)

```
                        merge: Optional[bool] = True) -> Tuple[List[List[str]],␣
↪List[List[str]]]:
    """Select the preds and labels of the first sub-word token for each word.
    The PreTrainedTokenizer tends to split word into sub-word tokens, and we select the␣
↪prediction of the first sub-word
    token as the prediction of this word.
    Args:
        preds (`np.ndarray`):
            The prediction ids of the input.
        labels (`np.ndarray`):
            The label ids of the input.
        merge (`bool`):
            Whether merge the predictions and labels into a one-dimensional list.
    Return:
        final_preds, final_labels (`Tuple[List[List[str]], List[List[str]]]`):
            The tuple of final predictions and labels.
    """
    final_preds = []
    final_labels = []

    if merge:
        final_preds = preds[labels != -100].tolist()
        final_labels = labels[labels != -100].tolist()
    else:
        for i in range(labels.shape[0]):
            final_preds.append(preds[i][labels[i] != -100].tolist())
            final_labels.append(labels[i][labels[i] != -100].tolist())

    return final_preds, final_labels
```

6.21.5 convert_to_names

Converts the given labels from id to their names by obtaining the value based on the given key from `id2label` dictionary, containing the correspondence between the ids and names of each label.

Args:

- `instances`: A list of string lists containing label ids of the instances.
- `id2label`: A dictionary containing the correspondence between the ids and names of each label.

Returns:

- `name_instances`: A list of string lists containing the label names, where each value corresponds to the id in the input list.

```
def convert_to_names(instances: List[List[str]],
                     id2label: Dict[str, str]) -> List[List[str]]:
    """Converts the given labels from id to their names.
    Converts the given labels from id to their names by obtaining the value based on the␣
↪given key from `id2label`
    dictionary, containing the correspondence between the ids and names of each label.
    Args:
```

(continues on next page)

(continued from previous page)

```
        instances (`List[List[str]]`):
            A list of string lists containing label ids of the instances.
        id2label (`Dict[int, str]`):
            A dictionary containing the correspondence between the ids and names of each␣
↪label.
    Returns:
        name_instances (`List[List[str]]`):
            A list of string lists containing the label names, where each value␣
↪corresponds to the id in the input list.
    """
    name_instances = []
    for instance in instances:
        name_instances.append([id2label[item] for item in instance])
    return name_instances
```

6.21.6 compute_span_F1

Computes the F1 score of the Sequence Labeling (SL) paradigm. The prediction of the model is converted into strings, then the overall F1 score of the prediction could be calculated.

Args:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.

Returns: - A dictionary containing the calculation result of F1 score.

```
def compute_span_F1(logits: np.ndarray,
                    labels: np.ndarray,
                    **kwargs) -> Dict[str, int]:
    """Computes the F1 score of the Sequence Labeling (SL) paradigm.
    Computes the F1 score of the Sequence Labeling (SL) paradigm. The prediction of the␣
↪model is converted into strings,
    then the overall F1 score of the prediction could be calculated.
    Args:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
    Returns:
        `Dict[str: float]`:
            A dictionary containing the calculation result of F1 score.
    """

    preds = np.argmax(logits, axis=-1) if len(logits.shape) == 3 else logits
    # convert id to name
    training_args = kwargs["training_args"]
    if training_args.task_name == "EAE":
        id2label = {id: role for role, id in training_args.role2id.items()}
    elif training_args.task_name == "ED":
```

(continues on next page)

(continued from previous page)

```
        id2label = {id: role for role, id in training_args.type2id.items()}
    else:
        raise ValueError("No such task!")
    final_preds, final_labels = select_start_position(preds, labels, False)
    final_preds = convert_to_names(final_preds, id2label)
    final_labels = convert_to_names(final_labels, id2label)

    # if the type is wrongly predicted, set arguments NA
    if training_args.task_name == "EAE":
        pred_types = training_args.data_for_evaluation["pred_types"]
        true_types = training_args.data_for_evaluation["true_types"]
        assert len(pred_types) == len(true_types)
        assert len(pred_types) == len(final_labels)
        for i, (pred, true) in enumerate(zip(pred_types, true_types)):
            if pred != true:
                final_preds[i] = [id2label[0]] * len(final_preds[i])  # set to NA

    micro_f1 = span_f1_score(final_labels, final_preds, mode='strict', scheme=IOB2) *␣
↪100.0
    return {"micro_f1": micro_f1}
```

6.21.7 compute_F1

Computes the F1 score of the Token Classification (TC) paradigm. The prediction of the model is converted into strings, then the overall F1 score of the prediction could be calculated.

Args:

- ``logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.

Returns: - A dictionary containing the calculation result of F1 score.

```
def compute_F1(logits: np.ndarray,
               labels: np.ndarray,
               **kwargs) -> Dict[str, float]:
    """Computes the F1 score of the Token Classification (TC) paradigm.
    Computes the F1 score of the Token Classification (TC) paradigm. The prediction of␣
↪the model is converted into
    strings, then the overall F1 score of the prediction could be calculated.
    Args:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
    Returns:
        `Dict[str: float]`:
            A dictionary containing the calculation result of F1 score.
    """
    predictions = np.argmax(logits, axis=-1)
```

(continues on next page)

(continued from previous page)

```
    training_args = kwargs["training_args"]
    # if the type is wrongly predicted, set arguments NA
    if training_args.task_name == "EAE":
        pred_types = training_args.data_for_evaluation["pred_types"]
        true_types = training_args.data_for_evaluation["true_types"]
        assert len(pred_types) == len(true_types)
        assert len(pred_types) == len(predictions)
        for i, (pred, true) in enumerate(zip(pred_types, true_types)):
            if pred != true:
                predictions[i] = 0 # set to NA
        pos_labels = list(set(training_args.role2id.values()))
    else:
        pos_labels = list(set(training_args.type2id.values()))
    pos_labels.remove(0)
    micro_f1 = f1_score(labels, predictions, labels=pos_labels, average="micro") * 100.0
    return {"micro_f1": micro_f1}
```

6.21.8 softmax

Conducts the softmax operation on the last dimension and returns a numpy array.

Args:

- `logits`: An numpy array of integers containing the type of each logit.
- `dim`: An integer indicating the dimension for the softmax operation.

Returns:

- An numpy array representing the normalized probability of each logit corresponding to each type of label.

```
def softmax(logits: np.ndarray,
            dim: Optional[int] = -1) -> np.ndarray:
    """Conducts the softmax operation on the last dimension.
    Conducts the softmax operation on the last dimension and returns a numpy array.
    Args:
        logits (`np.ndarray`):
            An numpy array of integers containing the type of each logit.
        dim (`int`, `optional`, defaults to -1):
            An integer indicating the dimension for the softmax operation.
    Returns:
        `np.ndarray`:
            An numpy array representing the normalized probability of each logit␣
↪corresponding to each type of label.
    """
    logits = torch.tensor(logits)
    return torch.softmax(logits, dim=dim).numpy()
```

6.21.9 compute_accuracy

Computes the accuracy of the predictions by calculating the fraction of the true label prediction count and the entire number of data pieces.

Args:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.

Returns:

- A dictionary containing the calculation result of the accuracy.

```
def compute_accuracy(logits: np.ndarray,
                     labels: np.ndarray,
                     **kwargs) -> Dict[str, int]:
    """Compute the accuracy of the predictions.
    Compute the accuracy of the predictions by calculating the fraction of the true␣
↪label prediction count and the
    entire number of data pieces.
    Args:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels:
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
    Returns:
        `Dict[str: float]`:
            A dictionary containing the calculation result of the accuracy.
    """
    predictions = np.argmax(softmax(logits), axis=-1)
    accuracy = (predictions == labels).sum() / labels.shape[0]
    return {"accuracy": accuracy}
```

6.21.10 compute_mrc_F1

Computes the F1 score of the Machine Reading Comprehension (MRC) method. The prediction of the model is firstly decoded into strings, then the overall F1 score of the prediction could be calculated.

Args:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.

Returns:

- A dictionary containing the calculation result of F1 score.

```
def compute_mrc_F1(logits: np.ndarray,
                   labels: np.ndarray,
                   **kwargs) -> Dict[str, float]:
    """Computes the F1 score of the Machine Reading Comprehension (MRC) method.
    Computes the F1 score of the Machine Reading Comprehension (MRC) method. The␣
```

(continues on next page)

(continued from previous page)

```
↪prediction of the model is firstly
    decoded into strings, then the overall F1 score of the prediction could be␣
↪calculated.
    Args:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
    Returns:
        `Dict[str: float]`:
            A dictionary containing the calculation result of F1 score.
    """
    start_logits, end_logits = np.split(logits, 2, axis=-1)
    all_predictions, all_labels = make_predictions(start_logits, end_logits, kwargs[
↪"training_args"])
    micro_f1 = compute_mrc_F1_cls(all_predictions, all_labels)
    return {"micro_f1": micro_f1}
```

6.22 Convert Format

```
import json
import logging
import numpy as np

from typing import List, Dict, Union, Tuple
from sklearn.metrics import f1_score
from .metric import select_start_position, compute_unified_micro_f1
from ..input_engineering.input_utils import (
    get_left_and_right_pos,
    check_pred_len,
    get_ed_candidates,
    get_eae_candidates,
    get_event_preds,
    get_plain_label,
)
logger = logging.getLogger(__name__)
```

6.22.1 get_pred_per_mention

Get the predicted event type or argument role for each mention via the predictions of different paradigms. The predictions of Sequence Labeling, Seq2Seq, MRC paradigms are not aligned to each word. We need to convert the paradigm-dependent predictions to word-level for the unified evaluation. This function is designed to get the prediction for each single mention, given the paradigm-dependent predictions.

Args:

- pos_start: The start position of the mention in the sequence of tokens.
- pos_end: The end position of the mention in the sequence of tokens.

- `preds`: The predictions of the sequence of tokens.
- `id2label`: A dictionary that contains the mapping from id to textual label.
- `label`: The ground truth label of the input mention.
- `label2id`: A dictionary that contains the mapping from textual label to id.
- `text`: The text of the input context.
- `paradigm`: A string that indicates the paradigm.

Returns:

- A string which represents the predicted label.

```
def get_pred_per_mention(pos_start: int,
                         pos_end: int,
                         preds: List[Union[str, Tuple[str, str]]],
                         id2label: Dict[int, str] = None,
                         label: str = None,
                         label2id: Dict[str, int] = None,
                         text: str = None,
                         paradigm: str = "sl") -> str:
    """Get the predicted event type or argument role for each mention via the␣
↪predictions of different paradigms.
    The predictions of Sequence Labeling, Seq2Seq, MRC paradigms are not aligned to each␣
↪word. We need to convert the
    paradigm-dependent predictions to word-level for the unified evaluation. This␣
↪function is designed to get the
    prediction for each single mention, given the paradigm-dependent predictions.
    Args:
        pos_start (`int`):
            The start position of the mention in the sequence of tokens.
        pos_end (`int`):
            The end position of the mention in the sequence of tokens.
        preds (`List[Union[str, Tuple[str, str]]]`):
            The predictions of the sequence of tokens.
        id2label (`Dict[int, str]`):
            A dictionary that contains the mapping from id to textual label.
        label (`str`):
            The ground truth label of the input mention.
        label2id (`Dict[str, int]`):
            A dictionary that contains the mapping from textual label to id.
        text (`str`):
            The text of the input context.
        paradigm (`str`):
            A string that indicates the paradigm.
    Returns:
        A string which represents the predicted label.
    """
    if paradigm == "sl":
        # sequence labeling paradigm
        if pos_start == pos_end or\
                pos_end > len(preds) or \
                id2label[int(preds[pos_start])] == "O" or \
                id2label[int(preds[pos_start])].split("-")[0] != "B":
```

(continues on next page)

(continued from previous page)

```
            return "NA"

        predictions = set()
        for pos in range(pos_start, pos_end):
            _pred = id2label[int(preds[pos])][2:]
            predictions.add(_pred)

        if len(predictions) > 1:
            return "NA"
        else:
            return list(predictions)[0]

    elif paradigm == "s2s":
        # seq2seq paradigm
        predictions = []
        word = text[pos_start: pos_end]
        for i, pred in enumerate(preds):
            if pred[0] == word:
                if pred[1] in label2id:
                    pred_label = pred[1]
                    predictions.append(pred_label)
        if label in predictions:
            pred_label = label
        else:
            pred_label = predictions[0] if predictions else "NA"

        # remove the prediction that has been used for a specific mention.
        if (word, pred_label) in preds:
            preds.remove((word, pred_label))

        return pred_label

    elif paradigm == "mrc":
        # mrc paradigm
        predictions = []
        for pred in preds:
            if pred[1] == (pos_start, pos_end - 1):
                pred_role = pred[0].split("_")[-1]
                predictions.append(pred_role)

        if label in predictions:
            return label
        else:
            return predictions[0] if predictions else "NA"
    else:
        raise NotImplementedError
```

6.22.2 get_trigger_detection_sl

Obtains the event detection prediction results of the ACE2005 dataset based on the sequence labeling paradigm, predicting the labels and calculating the micro F1 score based on the predictions and labels.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `data_file`: A string indicating the path of the testing data file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- `results`: A list of strings indicating the prediction results of event triggers.

```
def get_trigger_detection_sl(preds: np.array,
                                    labels: np.array,
                                    data_file: str,
                                    data_args,
                                    is_overflow) -> List[str]:
    """Obtains the event detection prediction results of the ACE2005 dataset based on␣
↪the sequence labeling paradigm.
    Obtains the event detection prediction results of the ACE2005 dataset based on the␣
↪sequence labeling paradigm,
    predicting the labels and calculating the micro F1 score based on the predictions␣
↪and labels.
    Args:
        preds (`np.array`):
            A list of strings indicating the predicted types of the instances.
        labels (`np.array`):
            A list of strings indicating the actual labels of the instances.
        data_file (`str`):
            A string indicating the path of the testing data file.
        data_args:
            The pre-defined arguments for data processing.
        is_overflow:
    Returns:
        results (`List[str]`):
            A list of strings indicating the prediction results of event triggers.
    """
    # get per-word predictions
    preds, labels = select_start_position(preds, labels, False)
    results = []
    label_names = []
    language = data_args.language

    with open(data_file, "r", encoding='utf-8') as f:
        lines = f.readlines()
        for i, line in enumerate(lines):
            item = json.loads(line.strip())

            if not is_overflow[i]:
                check_pred_len(pred=preds[i], item=item, language=language)
```

(continues on next page)

(continued from previous page)

```
            candidates, label_names_per_item = get_ed_candidates(item=item)
            label_names.extend(label_names_per_item)

            # loop for converting
            for candidate in candidates:
                left_pos, right_pos = get_left_and_right_pos(text=item["text"],␣
↪trigger=candidate, language=language)
                pred = get_pred_per_mention(left_pos, right_pos, preds[i], data_args.
↪id2type)
                results.append(pred)

    if "events" in item:
        micro_f1 = compute_unified_micro_f1(label_names=label_names, results=results)
        logger.info("{} test performance after converting: {}".format(data_args.dataset_
↪name, micro_f1))

    return results
```

6.22.3 get_argument_extraction_sl

Obtains the event argument extraction prediction results of the ACE2005 dataset based on the sequence labeling paradigm, predicting the labels of entities and negative triggers and calculating the micro F1 score based on the predictions and labels.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `data_file`: A string indicating the path of the testing data file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- `results`: A list of strings indicating the prediction results of event arguments.

```
def get_argument_extraction_sl(preds: np.array,
                                    labels: np.array,
                                    data_file: str,
                                    data_args,
                                    is_overflow) -> List[str]:
    """Obtains the event argument extraction results of the ACE2005 dataset based on the␣
↪sequence labeling paradigm.
    Obtains the event argument extraction prediction results of the ACE2005 dataset␣
↪based on the sequence labeling
    paradigm, predicting the labels of entities and negative triggers and calculating␣
↪the micro F1 score based on the
    predictions and labels.
    Args:
        preds (`np.array`):
            A list of strings indicating the predicted types of the instances.
```

(continues on next page)

(continued from previous page)

```
        labels (`np.array`):
            A list of strings indicating the actual labels of the instances.
        data_file (`str`):
            A string indicating the path of the testing data file.
        data_args:
            The pre-defined arguments for data processing.
        is_overflow:
    Returns:
        results (`List[str]`):
            A list of strings indicating the prediction results of event arguments.
    """
    # evaluation mode
    eval_mode = data_args.eae_eval_mode
    language = data_args.language
    golden_trigger = data_args.golden_trigger

    # pred events
    event_preds = get_event_preds(pred_file=data_args.test_pred_file)

    # get per-word predictions
    preds, labels = select_start_position(preds, labels, False)
    results = []
    label_names = []
    with open(data_file, "r", encoding="utf-8") as f:
        trigger_idx = 0
        eae_instance_idx = 0
        lines = f.readlines()
        for line in lines:
            item = json.loads(line.strip())
            text = item["text"]
            for event in item["events"]:
                for trigger in event["triggers"]:
                    true_type = event["type"]
                    pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                    trigger_idx += 1

                    if eval_mode in ['default', 'loose']:
                        if pred_type == "NA":
                            continue

                    if not is_overflow[eae_instance_idx]:
                        check_pred_len(pred=preds[eae_instance_idx], item=item,␣
↪language=language)

                    candidates, label_names_per_trigger = get_eae_candidates(item=item,␣
↪trigger=trigger)
                    label_names.extend(label_names_per_trigger)

                    # loop for converting
                    for candi in candidates:
                        if true_type == pred_type:
```

(continues on next page)

(continued from previous page)

```
                            # get word positions
                            left_pos, right_pos = get_left_and_right_pos(text=text,␣
↪trigger=candi, language=language)
                            # get predictions
                            pred = get_pred_per_mention(left_pos, right_pos, preds[eae_
↪instance_idx], data_args.id2role)
                        else:
                            pred = "NA"
                        # record results
                        results.append(pred)
                    eae_instance_idx += 1

            # negative triggers
            for trigger in item["negative_triggers"]:
                true_type = "NA"
                pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                trigger_idx += 1

                if eval_mode in ['default', 'strict']:  # loose mode has no neg
                    if pred_type != "NA":
                        if not is_overflow[eae_instance_idx]:
                            check_pred_len(pred=preds[eae_instance_idx], item=item,␣
↪language=language)

                        candidates, label_names_per_trigger = get_eae_
↪candidates(item=item, trigger=trigger)
                        label_names.extend(label_names_per_trigger)

                        # loop for converting
                        for candi in candidates:
                            # get word positions
                            left_pos, right_pos = get_left_and_right_pos(text=text,␣
↪trigger=candi, language=language)
                            # get predictions
                            pred = get_pred_per_mention(left_pos, right_pos, preds[eae_
↪instance_idx], data_args.id2role)
                            # record results
                            results.append(pred)

                        eae_instance_idx += 1

        assert len(preds) == eae_instance_idx

    pos_labels = list(set(label_names))
    pos_labels.remove("NA")
    micro_f1 = f1_score(label_names, results, labels=pos_labels, average="micro") * 100.0

    logger.info('Number of Instances: {}'.format(eae_instance_idx))
    logger.info("{} test performance after converting: {}".format(data_args.dataset_name,
↪ micro_f1))
    return results
```

6.22.4 get_argument_extraction_mrc

Obtains the event argument extraction prediction results of the ACE2005 dataset based on the MRC paradigm, predicting the labels of entities and negative triggers and calculating the micro F1 score based on the predictions and labels.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `data_file`: A string indicating the path of the testing data file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- `results`: A list of strings indicating the prediction results of event arguments.

```
def get_argument_extraction_mrc(preds, labels, data_file, data_args, is_overflow):
    """Obtains the event argument extraction results of the ACE2005 dataset based on the␣
↪MRC paradigm.
    Obtains the event argument extraction prediction results of the ACE2005 dataset␣
↪based on the MRC paradigm,
    predicting the labels of entities and negative triggers and calculating the micro F1␣
↪score based on the
    predictions and labels.
    Args:
        preds (`np.array`):
            A list of strings indicating the predicted types of the instances.
        labels (`np.array`):
            A list of strings indicating the actual labels of the instances.
        data_file (`str`):
            A string indicating the path of the testing data file.
        data_args:
            The pre-defined arguments for data processing.
        is_overflow:
    Returns:
        results (`List[str]`):
            A list of strings indicating the prediction results of event arguments.
    """

    # evaluation mode
    eval_mode = data_args.eae_eval_mode
    golden_trigger = data_args.golden_trigger
    language = data_args.language

    # pred events
    event_preds = get_event_preds(pred_file=data_args.test_pred_file)

    # get per-word predictions
    results = []
    all_labels = []
    with open(data_args.test_file, "r", encoding="utf-8") as f:
        trigger_idx = 0
        eae_instance_idx = 0
```

(continues on next page)

(continued from previous page)

```
        lines = f.readlines()
        for line in lines:
            item = json.loads(line.strip())
            text = item["text"]

            # preds per index
            preds_per_idx = []
            for pred in preds:
                if pred[-1] == trigger_idx:
                    preds_per_idx.append(pred)

            for event in item["events"]:
                for trigger in event["triggers"]:
                    true_type = event["type"]
                    pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                    trigger_idx += 1

                    if eval_mode in ['default', 'loose']:
                        if pred_type == "NA":
                            continue

                    # get candidates
                    candidates, labels_per_idx = get_eae_candidates(item, trigger)
                    all_labels.extend(labels_per_idx)

                    # loop for converting
                    for cid, candi in enumerate(candidates):
                        label = labels_per_idx[cid]
                        if pred_type == true_type:
                            # get word positions
                            left_pos, right_pos = get_left_and_right_pos(text=text,␣
↪trigger=candi, language=language)
                            # get predictions
                            pred_role = get_pred_per_mention(pos_start=left_pos, pos_
↪end=right_pos, preds=preds_per_idx,
                                                             label=label, paradigm='mrc')
                        else:
                            pred_role = "NA"
                        # record results
                        results.append(pred_role)
                    eae_instance_idx += 1

            # negative triggers
            for trigger in item["negative_triggers"]:
                true_type = "NA"
                pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                trigger_idx += 1

                if eval_mode in ['default', 'strict']:  # loose mode has no neg
                    if pred_type != "NA":
```

(continues on next page)

(continued from previous page)

```
                        # get candidates
                        candidates, labels_per_idx = get_eae_candidates(item, trigger)
                        all_labels.extend(labels_per_idx)

                        # loop for converting
                        for candi in candidates:
                            label = "NA"
                            # get word positions
                            left_pos, right_pos = get_left_and_right_pos(text=text,␣
↪trigger=candi, language=language)
                            # get predictions
                            pred_role = get_pred_per_mention(pos_start=left_pos, pos_
↪end=right_pos, preds=preds_per_idx,
                                                             label=label, paradigm='mrc')
                            # record results
                            results.append(pred_role)

                        eae_instance_idx += 1

    pos_labels = list(data_args.role2id.keys())
    pos_labels.remove("NA")
    micro_f1 = f1_score(all_labels, results, labels=pos_labels, average="micro") * 100.0

    logger.info('Number of Instances: {}'.format(eae_instance_idx))
    logger.info("{} test performance after converting: {}".format(data_args.dataset_name,
↪ micro_f1))
    return results
```

6.22.5 get_trigger_detection_s2s

Obtains the event detection prediction results of the ACE2005 dataset based on the Seq2Seq paradigm, predicting the labels and calculating the micro F1 score based on the predictions and labels.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `data_file`: A string indicating the path of the testing data file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- `results`: A list of strings indicating the prediction results of event triggers.

```
def get_trigger_detection_s2s(preds, labels, data_file, data_args, is_overflow):
    """Obtains the event detection prediction results of the ACE2005 dataset based on␣
↪the Seq2Seq paradigm.
    Obtains the event detection prediction results of the ACE2005 dataset based on the␣
↪Seq2Seq paradigm,
    predicting the labels and calculating the micro F1 score based on the predictions␣
↪and labels.
```

(continues on next page)

(continued from previous page)

```
    Args:
        preds (`np.array`):
            A list of strings indicating the predicted types of the instances.
        labels (`np.array`):
            A list of strings indicating the actual labels of the instances.
        data_file (`str`):
            A string indicating the path of the testing data file.
        data_args:
            The pre-defined arguments for data processing.
        is_overflow:
    Returns:
        results (`List[str]`):
            A list of strings indicating the prediction results of event triggers.
    """
    # get per-word predictions
    results = []
    label_names = []
    with open(data_file, "r", encoding='utf-8') as f:
        lines = f.readlines()
        for idx, line in enumerate(lines):
            item = json.loads(line.strip())
            text = item["text"]
            preds_per_idx = preds[idx]

            candidates, labels_per_item = get_ed_candidates(item=item)
            for i, label in enumerate(labels_per_item):
                labels_per_item[i] = get_plain_label(label)
            label_names.extend(labels_per_item)

            # loop for converting
            for cid, candidate in enumerate(candidates):
                label = labels_per_item[cid]
                # get word positions
                left_pos, right_pos = candidate["position"]
                # get predictions
                pred_type = get_pred_per_mention(pos_start=left_pos, pos_end=right_pos,␣
↪preds=preds_per_idx, text=text,
                                                 label=label, label2id=data_args.type2id,
↪ paradigm='s2s')
                # record results
                results.append(pred_type)

    if "events" in item:
        micro_f1 = compute_unified_micro_f1(label_names=label_names, results=results)
        logger.info("{} test performance after converting: {}".format(data_args.dataset_
↪name, micro_f1))

    return results
```

6.22.6 get_argument_extraction_s2s

Obtains the event argument extraction prediction results of the ACE2005 dataset based on the Seq2Seq paradigm, predicting the labels of entities and negative triggers and calculating the micro F1 score based on the predictions and labels.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `data_file`: A string indicating the path of the testing data file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- `results`: A list of strings indicating the prediction results of event arguments.

```
def get_argument_extraction_s2s(preds, labels, data_file, data_args, is_overflow):
    """Obtains the event argument extraction results of the ACE2005 dataset based on the␣
↪Seq2Seq paradigm.
    Obtains the event argument extraction prediction results of the ACE2005 dataset␣
↪based on the Seq2Seq paradigm,
    predicting the labels of entities and negative triggers and calculating the micro F1␣
↪score based on the
    predictions and labels.
    Args:
        preds (`np.array`):
            A list of strings indicating the predicted types of the instances.
        labels (`np.array`):
            A list of strings indicating the actual labels of the instances.
        data_file (`str`):
            A string indicating the path of the testing data file.
        data_args:
            The pre-defined arguments for data processing.
        is_overflow:
    Returns:
        results (`List[str]`):
            A list of strings indicating the prediction results of event arguments.
    """

    # evaluation mode
    eval_mode = data_args.eae_eval_mode
    golden_trigger = data_args.golden_trigger

    # pred events
    event_preds = get_event_preds(pred_file=data_args.test_pred_file)

    # get per-word predictions
    results = []
    all_labels = []
    with open(data_args.test_file, "r", encoding="utf-8") as f:
        trigger_idx = 0
        eae_instance_idx = 0
        lines = f.readlines()
```

(continues on next page)

(continued from previous page)

```
        for line in lines:
            item = json.loads(line.strip())
            text = item["text"]

            for event in item["events"]:
                for trigger in event["triggers"]:
                    true_type = event["type"]
                    pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                    trigger_idx += 1

                    if eval_mode in ['default', 'loose']:
                        if pred_type == "NA":
                            continue

                    # preds per index
                    preds_per_idx = preds[eae_instance_idx]
                    # get candidates
                    candidates, labels_per_idx = get_eae_candidates(item, trigger)
                    for i, label in enumerate(labels_per_idx):
                        labels_per_idx[i] = get_plain_label(label)
                    all_labels.extend(labels_per_idx)

                    # loop for converting
                    for cid, candidate in enumerate(candidates):
                        label = labels_per_idx[cid]
                        if pred_type == true_type:
                            # get word positions
                            left_pos, right_pos = candidate["position"]
                            # get predictions
                            pred_role = get_pred_per_mention(pos_start=left_pos, pos_
↪end=right_pos, preds=preds_per_idx,
                                                             text=text, label=label,␣
↪label2id=data_args.role2id,
                                                             paradigm='s2s')
                        else:
                            pred_role = "NA"
                        # record results
                        results.append(pred_role)
                    eae_instance_idx += 1

            # negative triggers
            for trigger in item["negative_triggers"]:
                true_type = "NA"
                pred_type = true_type if golden_trigger or event_preds is None else␣
↪event_preds[trigger_idx]
                trigger_idx += 1

                if eval_mode in ['default', 'strict']:  # loose mode has no neg
                    if pred_type != "NA":
                        # preds per index
                        preds_per_idx = preds[eae_instance_idx]
```

(continues on next page)

(continued from previous page)

```
                        # get candidates
                        candidates, labels_per_idx = get_eae_candidates(item, trigger)
                        for i, label in enumerate(labels_per_idx):
                            labels_per_idx[i] = get_plain_label(label)
                        all_labels.extend(labels_per_idx)

                        # loop for converting
                        for cid, candidate in enumerate(candidates):
                            label = labels_per_idx[cid]
                            # get word positions
                            left_pos, right_pos = candidate["position"]
                            # get predictions
                            pred_role = get_pred_per_mention(pos_start=left_pos, pos_
↪end=right_pos, preds=preds_per_idx,
                                                             text=text, label=label,␣
↪label2id=data_args.role2id,
                                                             paradigm='s2s')
                            # record results
                            results.append(pred_role)

                        eae_instance_idx += 1

        assert len(preds) == eae_instance_idx

    pos_labels = list(data_args.role2id.keys())
    pos_labels.remove("NA")
    micro_f1 = f1_score(all_labels, results, labels=pos_labels, average="micro") * 100.0

    logger.info("Number of Instances: {}".format(eae_instance_idx))
    logger.info("{} test performance after converting: {}".format(data_args.dataset_name,
↪ micro_f1))
    return results
```

6.23 Dump Results

```
import jsonlines
import json
import numpy as np
from tqdm import tqdm
from collections import defaultdict
from typing import List, Dict, Union, Tuple
from .convert_format import get_pred_per_mention
from .metric import select_start_position
from ..input_engineering.input_utils import check_pred_len, get_left_and_right_pos
```

6.23.1 get_sentence_arguments

Gets the predicted arguments from a sentence in the Sequence Labeling paradigm.

Args:

- input_sentence: A list of dictionaries each of which contains the word and the corresponding bio-role.

Returns:

- arguments: A list of dictionaries each of which contains the word and the corresponding role.

```
def get_sentence_arguments(input_sentence: List[Dict[str, str]]) -> List[Dict[str, str]]:
    """Get the predicted arguments from a sentence in the Sequence Labeling paradigm.
    Args:
        input_sentence (`List[Dict[str, str]]`):
            A list of dictionaries each of which contains the word and the corresponding␣
↪bio-role.
    Returns:
        arguments (`List[Dict[str, str]]`):
            A list of dictionaries each of which contains the word and the corresponding␣
↪role.
    """
    input_sentence.append({"role": "NA", "word": "<EOS>"})
    arguments = []

    previous_role = None
    previous_arg = ""
    for item in input_sentence:
        if item["role"] != "NA" and previous_role is None:
            previous_role = item["role"]
            previous_arg = item["word"]

        elif item["role"] == previous_role:
            previous_arg += item["word"]

        elif item["role"] != "NA":
            arguments.append({"role": previous_role, "argument": previous_arg})
            previous_role = item["role"]
            previous_arg = item["word"]

        elif previous_role is not None:
            arguments.append({"role": previous_role, "argument": previous_arg})
            previous_role = None
            previous_arg = ""

    return arguments
```

6.23.2 get_maven_submission

Converts the predictions to the submission format of the MAVEN dataset and dumps the predictions into a json file.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `instance_ids`: A list of strings containing the id of each instance to be predicted.
- `result_file`: A string indicating the path to place the written json file.

```
def get_maven_submission(preds: Union[np.array, List[str]],
                         instance_ids: List[str],
                         result_file: str) -> None:
    """Converts the predictions to the submission format of the MAVEN dataset.
    Converts the predictions to the submission format of the MAVEN dataset and dumps the␣
↪predictions into a json file.
    Args:
        preds (`List[str]`):
            A list of strings indicating the predicted types of the instances.
        instance_ids (`List[str]`):
            A list of strings containing the id of each instance to be predicted.
        result_file (`str`):
            A string indicating the path to place the written json file.
    """
    all_results = defaultdict(list)
    for i, pred in enumerate(preds):
        example_id, candidate_id = instance_ids[i].split("-")
        all_results[example_id].append({
            "id": candidate_id,
            "type_id": int(pred)
        })
    with open(result_file, "w") as f:
        for data_id in all_results.keys():
            format_result = dict(id=data_id, predictions=[])
            for candidate in all_results[data_id]:
                format_result["predictions"].append(candidate)
            f.write(json.dumps(format_result) + "\n")
```

6.23.3 get_maven_submission_sl

Obtains the instances' predictions in the test file of the MAVEN dataset based on the sequence labeling paradigm and converts the predictions to the dataset's submission format. The converted predictions are dumped into a json file for submission.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `result_file`: A string indicating the path to place the written json file.
- `type2id`: A dictionary containing the correspondences between event types and ids.
- `config`: The configurations of the model.

```
def get_maven_submission_sl(preds: Union[np.array, List[str]],
                            labels: Union[np.array, List[str]],
                            is_overflow,
                            result_file: str,
                            type2id: Dict[str, int],
                            config) -> None:
    """Converts the predictions to the submission format of the MAVEN dataset based on␣
↪the sequence labeling paradigm.
    Obtains the instances' predictions in the test file of the MAVEN dataset based on the␣
↪sequence labeling paradigm and
    converts the predictions to the dataset's submission format. The converted␣
↪predictions are dumped into a json file
    for submission.
    Args:
        preds (`List[str]`):
            A list of strings indicating the predicted types of the instances.
        labels (`List[str]`):
            A list of strings indicating the actual labels of the instances.
        is_overflow:
        result_file (`str`):
            A string indicating the path to place the written json file.
        type2id (`Dict[str, int]`):
            A dictionary containing the correspondences between event types and ids.
        config:
            The configurations of the model.
    """
    # get per-word predictions
    preds, _ = select_start_position(preds, labels, False)
    results = defaultdict(list)
    language = config.language

    with open(config.test_file, "r") as f:
        lines = f.readlines()
        for i, line in enumerate(lines):
            item = json.loads(line.strip())
            text = item["text"]

            # check for alignment
            if not is_overflow[i]:
                check_pred_len(pred=preds[i], item=item, language=language)

            for candidate in item["candidates"]:
                # get word positions
                word_pos_start, word_pos_end = get_left_and_right_pos(text=text,␣
↪trigger=candidate, language=language)
                # get predictions
                pred = get_pred_per_mention(word_pos_start, word_pos_end, preds[i],␣
↪config.id2type)
                # record results
                results[item["id"]].append({
                    "id": candidate["id"].split("-")[-1],
                    "type_id": int(type2id[pred]),
                })
```

(continues on next page)

(continued from previous page)

```
    # dump results
    with open(result_file, "w") as f:
        for id, preds_per_doc in results.items():
            results_per_doc = dict(id=id, predictions=preds_per_doc)
            f.write(json.dumps(results_per_doc)+"\n")
```

6.23.4 get_maven_submission_seq2seq

Obtains the instances' predictions in the test file of the MAVEN dataset based on the Sequence-to-Sequence (Seq2Seq) paradigm and converts the predictions to the dataset's submission format. The converted predictions are dumped into a json file for submission.

Args:

- `preds`: The textual predictions of the Event Type or Argument Role. A list of tuple lists, in which each tuple is (argument, role) or (trigger, event_type)
- `save_path`: A string indicating the path to place the written json file.
- `data_args`: The pre-defined arguments for data processing.

```
def get_maven_submission_seq2seq(preds: List[List[Tuple[str, str]]],
                                 save_path: str,
                                 data_args) -> None:
    """Converts the predictions to the submission format of the MAVEN dataset based on␣
↪the Seq2Seq paradigm.
    Obtains the instances' predictions in the test file of the MAVEN dataset based on the␣
↪Sequence-to-Sequence (Seq2Seq)
    paradigm and converts the predictions to the dataset's submission format. The␣
↪converted predictions are dumped into
    a json file for submission.
    Args:
        preds (`List[List[Tuple[str, str]]]`):
            The textual predictions of the Event Type or Argument Role.
            A list of tuple lists, in which each tuple is (argument, role) or (trigger,␣
↪event_type)
        save_path (`str`):
            A string indicating the path to place the written json file.
        data_args:
            The pre-defined arguments for data processing.
    """
    type2id = data_args.type2id
    results = defaultdict(list)
    with open(data_args.test_file, "r") as f:
        lines = f.readlines()
        for idx, line in enumerate(lines):
            item = json.loads(line.strip())
            text = item["text"]
            preds_per_idx = preds[idx]

            for candidate in item["candidates"]:
                label = "NA"
                left_pos, right_pos = candidate["position"]
```

(continues on next page)

(continued from previous page)

```
                # get predictions
                pred_type = get_pred_per_mention(pos_start=left_pos, pos_end=right_pos,␣
↪preds=preds_per_idx, text=text,
                                                 label=label, label2id=type2id, paradigm=
↪'s2s')

                # record results
                results[item["id"]].append({"id": candidate["id"].split("-")[-1], "type_
↪id": int(type2id[pred_type])})
    # dump results
    with open(save_path, "w") as f:
        for id, preds_per_doc in results.items():
            results_per_doc = dict(id=id, predictions=preds_per_doc)
            f.write(json.dumps(results_per_doc) + "\n")
```

6.23.5 get_leven_submission

Converts the predictions to the submission format of the LEVEN dataset and dumps the predictions into a json file.

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `instance_ids`: A list of strings containing the id of each instance to be predicted.
- `result_file`: A string indicating the path to place the written json file.

Returns:

- The parameters of the input are passed to the `get_maven_submission()` method for further predictions.

```
def get_leven_submission(preds: Union[np.array, List[str]],
                         instance_ids: List[str],
                         result_file: str) -> None:
    """Converts the predictions to the submission format of the LEVEN dataset.
    Converts the predictions to the submission format of the LEVEN dataset and dumps the␣
↪predictions into a json file.
    Args:
        preds (`List[str]`):
            A list of strings indicating the predicted types of the instances.
        instance_ids (`List[str]`):
            A list of strings containing the id of each instance to be predicted.
        result_file (`str`):
            A string indicating the path to place the written json file.
    Returns:
        The parameters of the input are passed to the `get_maven_submission()` method for␣
↪further predictions.
    """
    return get_maven_submission(preds, instance_ids, result_file)
```

6.23.6 get_leven_submission_sl

Obtains the instances' predictions in the test file of the LEVEN dataset based on the sequence labeling paradigm and converts the predictions to the dataset's submission format. The converted predictions are dumped into a json file for submission.

Args:

- `preds`: A list of strings indicating the predicted type of the instances.
- `labels`: A list of strings indicating the actual label of the instances.
- `result_file`: A string indicating the path to place the written json file.
- `type2id`: A dictionary containing the correspondences between event types and ids.
- `config`: The configurations of the model.

Returns:

- The parameters of the input are passed to the `get_maven_submission_sl()` method for further predictions.

```
def get_leven_submission_sl(preds: Union[np.array, List[str]],
                            labels: Union[np.array, List[str]],
                            is_overflow,
                            result_file: str,
                            type2id: Dict[str, int],
                            config):
    """Converts the predictions to the submission format of the LEVEN dataset based on␣
↪the sequence labeling paradigm.
    Obtains the instances' predictions in the test file of the LEVEN dataset based on the␣
↪sequence labeling paradigm and
    converts the predictions to the dataset's submission format. The converted␣
↪predictions are dumped into a json file
    for submission.
    Args:
        preds (`List[str]`):
            A list of strings indicating the predicted type of the instances.
        labels (`List[str]`):
            A list of strings indicating the actual label of the instances.
        is_overflow:
        result_file (`str`):
            A string indicating the path to place the written json file.
        type2id (`Dict[str, int]`):
            A dictionary containing the correspondences between event types and ids.
        config:
            The configurations of the model.
    Returns:
        The parameters of the input are passed to the `get_maven_submission_sl()` method␣
↪for further predictions.
    """
    return get_maven_submission_sl(preds, labels, is_overflow, result_file, type2id,␣
↪config)
```

6.23.7 get_leven_submission_seq2seq

Obtains the instances' predictions in the test file of the LEVEN dataset based on the Sequence-to-Sequence (Seq2Seq) paradigm and converts the predictions to the dataset's submission format. The converted predictions are dumped into a json file for submission.

Args:

- `preds`: The textual predictions of the Event Type or Argument Role. A list of tuple lists, in which each tuple is (argument, role) or (trigger, event_type)
- `save_path`: A string indicating the path to place the written json file.
- `data_args`: The pre-defined arguments for data processing.

Returns:

- The parameters of the input are passed to the `get_maven_submission_seq2seq()` method for further predictions. The formats of LEVEN and MAVEN are the same.

```
def get_leven_submission_seq2seq(preds: List[List[Tuple[str, str]]],
                                 save_path: str,
                                 data_args) -> None:
    """Converts the predictions to the submission format of the LEVEN dataset based on␣
↪the Seq2Seq paradigm.
    Obtains the instances' predictions in the test file of the LEVEN dataset based on the␣
↪Sequence-to-Sequence (Seq2Seq)
    paradigm and converts the predictions to the dataset's submission format. The␣
↪converted predictions are dumped into
    a json file for submission.
    Args:
        preds (`List[List[Tuple[str, str]]]`):
            The textual predictions of the Event Type or Argument Role.
            A list of tuple lists, in which each tuple is (argument, role) or (trigger,␣
↪event_type)
        save_path (`str`):
            A string indicating the path to place the written json file.
        data_args:
            The pre-defined arguments for data processing.
    Returns:
        The parameters of the input are passed to the `get_maven_submission_seq2seq()`␣
↪method for further predictions.
        The formats of LEVEN and MAVEN are the same.
    """
    return get_maven_submission_seq2seq(preds, save_path, data_args)
```

6.23.8 get_duee_submission_sl

Args:

- `preds`: A list of strings indicating the predicted types of the instances.
- `labels`: A list of strings indicating the actual labels of the instances.
- `result_file`: A string indicating the path to place the written json file.
- `config`: The configurations of the model.

Returns:

– `all_results`: A list of dictionaries containing the predictions of events.

```
def get_duee_submission_sl(preds: Union[np.array, List[str]],
                           labels: Union[np.array, List[str]],
                           is_overflow,
                           result_file: str,
                           config) -> List[Dict[str, Union[str, Dict]]]:
    """Converts the predictions to the submission format of the DuEE dataset based on␣
↪the sequence labeling paradigm.
    Obtains the instances' predictions in the test file of the DuEE dataset based on the␣
↪sequence labeling paradigm and
    converts the predictions to the dataset's submission format. The converted␣
↪predictions are dumped into a json file
    for submission.
    Args:
        preds (`List[str]`):
            A list of strings indicating the predicted types of the instances.
        labels (`List[str]`):
            A list of strings indicating the actual labels of the instances.
        is_overflow:
        result_file (`str`):
            A string indicating the path to place the written json file.
        config:
            The configurations of the model.
    Returns:
        all_results (`List[Dict[str, Union[str, Dict]]]`):
            A list of dictionaries containing the predictions of events.
    """
    # trigger predictions
    ed_preds = json.load(open(config.test_pred_file))

    # get per-word predictions
    preds, labels = select_start_position(preds, labels, False)
    all_results = []

    with open(config.test_file, "r", encoding='utf-8') as f:
        trigger_idx = 0
        example_idx = 0
        lines = f.readlines()
        for line in tqdm(lines, desc='Generating DuEE1.0 Submission Files'):
            item = json.loads(line.strip())

            item_id = item["id"]
            event_list = []

            for tid, trigger in enumerate(item["candidates"]):
                pred_event_type = ed_preds[trigger_idx]
                if pred_event_type != "NA":
                    if not is_overflow[example_idx]:
                        if config.language == "English":
                            assert len(preds[example_idx]) == len(item["text"].split())
                        elif config.language == "Chinese":
```

(continues on next page)

(continued from previous page)

```
                            assert len(preds[example_idx]) == len("".join(item["text"].
↪split()))  # remove space token
                        else:
                            raise NotImplementedError

                    pred_event = dict(event_type=pred_event_type, arguments=[])
                    sentence_result = []
                    for cid, candidate in enumerate(item["candidates"]):
                        if cid == tid:
                            continue
                        char_pos = candidate["position"]
                        if config.language == "English":
                            word_pos_start = len(item["text"][:char_pos[0]].split())
                            word_pos_end = word_pos_start + len(item["text"][char_
↪pos[0]:char_pos[1]].split())
                        elif config.language == "Chinese":
                            word_pos_start = len([w for w in item["text"][:char_pos[0]]␣
↪if w.strip('\n\xa0 ')])
                            word_pos_end = len([w for w in item["text"][:char_pos[1]] if␣
↪w.strip('\n\xa0 ')])
                        else:
                            raise NotImplementedError
                        # get predictions
                        pred = get_pred_per_mention(word_pos_start, word_pos_end,␣
↪preds[example_idx], config.id2role)
                        sentence_result.append({"role": pred, "word": candidate["trigger_
↪word"]})

                    pred_event["arguments"] = get_sentence_arguments(sentence_result)
                    if pred_event["arguments"]:
                        event_list.append(pred_event)

                    example_idx += 1

                trigger_idx += 1

            all_results.append({"id": item_id, "event_list": event_list})

    # dump results
    with jsonlines.open(result_file, "w") as f:
        for r in all_results:
            jsonlines.Writer.write(f, r)

    return all_results
```

6.24 Evaluation Utils

```
import os
import json
import shutil
import logging
import jsonlines
import numpy as np

from tqdm import tqdm
from pathlib import Path
from typing import List, Dict, Union, Tuple
from transformers import PreTrainedTokenizer

from ..trainer import Trainer
from ..trainer_seq2seq import Seq2SeqTrainer
from ..arguments import DataArguments, ModelArguments, TrainingArguments
from ..input_engineering.seq2seq_processor import extract_argument
from ..input_engineering.base_processor import EDDataProcessor, EAEDataProcessor
from ..input_engineering.mrc_converter import make_predictions, find_best_thresh

from .convert_format import get_trigger_detection_sl, get_trigger_detection_s2s

logger = logging.getLogger(__name__)
```

6.24.1 dump_preds

Save the Event Detection predictions for further use in the Event Argument Extraction task.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: The tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `output_dir`: The file path to dump the event detection predictions.
- `model_args`: The pre-defined arguments for model configuration.
- `data_args`: The pre-defined arguments for data processing.
- `training_args`: The pre-defined arguments for training event detection model.
- `mode`: The mode of the prediction, can be 'train', 'valid' or 'test'.

```
def dump_preds(trainer: Union[Trainer, Seq2SeqTrainer],
               tokenizer: PreTrainedTokenizer,
               data_class: type,
               output_dir: Union[str,Path],
               model_args: ModelArguments,
               data_args: DataArguments,
               training_args: TrainingArguments,
               mode: str = "train",
               ) -> None:
```

(continues on next page)

(continued from previous page)

```
    """Dump the Event Detection predictions for each token in the dataset.
    Save the Event Detection predictions for further use in the Event Argument␣
↪Extraction task.
    Args:
        trainer:
            The trainer for event detection.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
        data_class:
            The processor of the input data.
        output_dir (`str`):
            The file path to dump the event detection predictions.
        model_args (`ModelArguments`):
            The pre-defined arguments for model configuration.
        data_args (`DataArguments`):
            The pre-defined arguments for data processing.
        training_args (`TrainingArguments`):
            The pre-defined arguments for training event detection model.
        mode (`str`):
            The mode of the prediction, can be 'train', 'valid' or 'test'.
    Returns:
        None
    """
    if mode == "train":
        data_file = data_args.train_file
    elif mode == "valid":
        data_file = data_args.validation_file
    elif mode == "test":
        data_file = data_args.test_file
    else:
        raise NotImplementedError

    logits, labels, metrics, dataset = predict(trainer=trainer, tokenizer=tokenizer,␣
↪data_class=data_class,
                                               data_args=data_args, data_file=data_file,
                                               training_args=training_args)
    logger.info("\n")
    logger.info("{}-Dump Preds-{}{}".format("-" * 25, mode, "-" * 25))
    logger.info("Test file: {}, Metrics: {}, Split_Infer: {}".format(data_file, metrics,␣
↪data_args.split_infer))

    preds = get_pred_s2s(logits, tokenizer) if model_args.paradigm == "seq2seq" else np.
↪argmax(logits, axis=-1)

    if model_args.paradigm == "token_classification":
        pred_labels = [data_args.id2type[pred] for pred in preds]
    elif model_args.paradigm == "sequence_labeling":
        pred_labels = get_trigger_detection_sl(preds, labels, data_file, data_args,␣
↪dataset.is_overflow)
    elif model_args.paradigm == "seq2seq":
        pred_labels = get_trigger_detection_s2s(preds, labels, data_file, data_args,␣
↪None)
```

(continues on next page)

(continued from previous page)

```
    else:
        raise NotImplementedError

    save_path = os.path.join(output_dir, "{}_preds.json".format(mode))

    json.dump(pred_labels, open(save_path, "w", encoding='utf-8'), ensure_ascii=False)
    logger.info("ED {} preds dumped to {}\n ED finished!".format(mode, save_path))
```

6.24.2 get_pred_s2s

Converts Seq2Seq output logits to textual Event Type Prediction in Event Detection task, or to textual Argument Role Prediction in Event Argument Extraction task.

Args:

- `logits`: The decoded logits of the Seq2Seq model.
- `tokenizer`: A string indicating the tokenizer proposed for the tokenization process.
- `pred_types`: The event detection predictions, only used in Event Argument Extraction task.

Returns:

- `preds`: The textual predictions of the Event Type or Argument Role. A list of tuple lists, in which each tuple is (argument, role) or (trigger, event_type)

```
def get_pred_s2s(logits: np.array,
                 tokenizer: PreTrainedTokenizer,
                 pred_types: List[str] = None,
                 ) -> List[List[Tuple[str, str]]]:
    """Convert Seq2Seq output logits to textual Event Type Prediction or Argument Role␣
↪Prediction.
    Convert Seq2Seq output logits to textual Event Type Prediction in Event Detection␣
↪task,
        or to textual Argument Role Prediction in Event Argument Extraction task.
    Args:
        logits (`np.array`):
            The decoded logits of the Seq2Seq model.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
        pred_types (`List[str]`):
            The event detection predictions, only used in Event Argument Extraction task.
    Returns:
        preds (`List[List[Tuple[str, str]]]`):
            The textual predictions of the Event Type or Argument Role.
            A list of tuple lists, in which each tuple is (argument, role) or (trigger,␣
↪event_type)
    """

    decoded_preds = tokenizer.batch_decode(logits, skip_special_tokens=False)

    def clean_str(x_str):
        for to_remove_token in [tokenizer.eos_token, tokenizer.pad_token]:
            x_str = x_str.replace(to_remove_token, '')
```

(continues on next page)

(continued from previous page)

```
        return x_str.strip()

    preds = list()
    for i, pred in enumerate(decoded_preds):
        pred = clean_str(pred)
        pred_type = pred_types[i] if pred_types else "NA"
        arguments = extract_argument(pred, i, pred_type)
        tmp = list()
        for arg in arguments:
            tmp.append((arg[-1], arg[-2]))
        preds.append(tmp)

    return preds
```

6.24.3 get_pred_mrc

Converts MRC output logits to textual Event Type Prediction in Event Detection task, or to textual Argument Role Prediction in Event Argument Extraction task.

Args:

- `logits`: The logits output of the MRC model.
- `training_args`: The event detection predictions, only used in Event Argument Extraction task.

Returns:

- `preds`: The textual predictions of the Event Type or Argument Role. A list of tuple lists, in which each tuple is (argument, role) or (trigger, event_type)

```
def get_pred_mrc(logits: np.array,
                 training_args: TrainingArguments,
                 ) -> List[List[Tuple[str, str]]]:
    """Convert MRC output logits to textual Event Type Prediction or Argument Role␣
↪Prediction.
    Convert MRC output logits to textual Event Type Prediction in Event Detection task,
        or to textual Argument Role Prediction in Event Argument Extraction task.
    Args:
        logits (`np.array`):
            The logits output of the MRC model.
        training_args (`TrainingArguments`):
            The event detection predictions, only used in Event Argument Extraction task.
    Returns:
        preds (`List[List[Tuple[str, str]]]`):
            The textual predictions of the Event Type or Argument Role.
            A list of tuple lists, in which each tuple is (argument, role) or (trigger,␣
↪event_type)
    """

    start_logits, end_logits = np.split(logits, 2, axis=-1)
    all_preds, all_labels = make_predictions(start_logits, end_logits, training_args)

    all_preds = sorted(all_preds, key=lambda x: x[-2])
```

(continues on next page)

(continued from previous page)

```
    best_na_thresh = find_best_thresh(all_preds, all_labels)
    logger.info("Best thresh founded. %.6f" % best_na_thresh)

    final_preds = []
    for argument in all_preds:
        if argument[-2] < best_na_thresh:
            final_preds.append(argument[:-2] + argument[-1:])  # no na_prob

    return final_preds
```

6.24.4 predict

Predicts the test set of the event detection task. The prediction of logits and labels, evaluation metrics' results, and the dataset would be returned.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: The tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `data_args`: The pre-defined arguments for data processing.
- `data_file`: A string representing the file path of the dataset.
- `training_args`: The pre-defined arguments for training.

Returns:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.
- `metrics`: The evaluation metrics result based on the predictions and annotations.
- `dataset`: An instance of the testing dataset.

```
def predict(trainer: Union[Trainer, Seq2SeqTrainer],
            tokenizer: PreTrainedTokenizer,
            data_class: type,
            data_args: DataArguments,
            data_file: str,
            training_args: TrainingArguments,
            ) -> Tuple[np.array, np.array, Dict, Union[EDDataProcessor,␣
↪EAEDataProcessor]]:
    """Predicts the test set of the Event Detection task or Event Argument Extraction␣
↪task.
    Predicts the test set of the event detection task. The prediction of logits and␣
↪labels, evaluation metrics' results,
    and the dataset would be returned.
    Args:
        trainer:
            The trainer for event detection.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
```

(continues on next page)

(continued from previous page)

```
        data_class:
            The processor of the input data.
        data_args:
            The pre-defined arguments for data processing.
        data_file (`str`):
            A string representing the file path of the dataset.
        training_args (`TrainingArguments`):
            The pre-defined arguments for training.
    Returns:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels: (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
        metrics:
            The evaluation metrics result based on the predictions and annotations.
        dataset:
            An instance of the testing dataset.
    """

    if training_args.task_name == "ED":
        pred_func = predict_sub_ed if data_args.split_infer else predict_ed
        return pred_func(trainer, tokenizer, data_class, data_args, data_file)

    elif training_args.task_name == 'EAE':
        pred_func = predict_sub_eae if data_args.split_infer else predict_eae
        return pred_func(trainer, tokenizer, data_class, data_args, training_args)

    else:
        raise NotImplementedError
```

6.24.5 get_sub_files

Splits a large data file into several small data files for evaluation. Sometimes, the test data file can be too large to make prediction due to GPU memory constrain. Therefore, we split the large file into several smaller ones and make predictions on each.

Args:

- `input_test_file`: The path to the large data file that needs to split.
- `input_test_pred_file`: The path to the Event Detection Predictions of the input_test_file. Only used in Event Argument Extraction task.
- `sub_size`: The number of items contained each split file.

Returns:

- **if `input_test_pred_file` is not `None`: (Event Argument Extraction task)**
  - `output_test_files`, `output_pred_files`: The lists of paths to the split files.
- **else:**
  - `output_test_files`: The list of paths to the split files.

```
def get_sub_files(input_test_file: str,
                  input_test_pred_file: str = None,
                  sub_size: int = 5000,
                  ) -> Union[List[str], Tuple[List[str], List[str]]]:
    """Split a large data file into several small data files for evaluation.
    Sometimes, the test data file can be too large to make prediction due to GPU memory␣
↪constrain.
    Therefore, we split the large file into several smaller ones and make predictions on␣
↪each.
    Args:
        input_test_file (`str`):
            The path to the large data file that needs to split.
        input_test_pred_file (`str`):
            The path to the Event Detection Predictions of the input_test_file.
            Only used in Event Argument Extraction task.
        sub_size (`int`):
            The number of items contained each split file.
    Returns:
        if input_test_pred_file is not None: (Event Argument Extraction task)
            output_test_files, output_pred_files:
                The lists of paths to the split files.
        else:
            output_test_files:
                The list of paths to the split files.
    """
    test_data = list(jsonlines.open(input_test_file))
    sub_data_folder = '/'.join(input_test_file.split('/')[:-1]) + '/test_cache/'

    # clear the cache dir before split evaluate
    if os.path.isdir(sub_data_folder):
        shutil.rmtree(sub_data_folder)
        logger.info("Cleared Existing Cache Dir")

    os.makedirs(sub_data_folder, exist_ok=False)
    output_test_files = []

    pred_data, sub_pred_folder = None, None
    output_pred_files = []
    if input_test_pred_file:
        pred_data = json.load(open(input_test_pred_file, encoding='utf-8'))
        sub_pred_folder = '/'.join(input_test_pred_file.split('/')[:-1]) + '/test_cache/'
        os.makedirs(sub_pred_folder, exist_ok=True)

    pred_start = 0
    for sub_id, i in enumerate(range(0, len(test_data), sub_size)):
        test_data_sub = test_data[i: i + sub_size]
        test_file_sub = sub_data_folder + 'sub-{}.json'.format(sub_id)

        with jsonlines.open(test_file_sub, 'w') as f:
            for data in test_data_sub:
                jsonlines.Writer.write(f, data)

        output_test_files.append(test_file_sub)
```

(continues on next page)

(continued from previous page)

```
        if input_test_pred_file:
            pred_end = pred_start + sum([len(d['candidates']) for d in test_data_sub])
            test_pred_sub = pred_data[pred_start: pred_end]
            pred_start = pred_end

            test_pred_file_sub = sub_pred_folder + 'sub-{}.json'.format(sub_id)

            with open(test_pred_file_sub, 'w', encoding='utf-8') as f:
                json.dump(test_pred_sub, f, ensure_ascii=False)

            output_pred_files.append(test_pred_file_sub)

    if input_test_pred_file:
        return output_test_files, output_pred_files

    return output_test_files
```

6.24.6 predict_ed

Predicts the test set of the event detection task. The prediction of logits and labels, evaluation metrics' results, and the dataset would be returned.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: The tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `data_args`: The pre-defined arguments for data processing.
- `data_file`: A string representing the file path of the dataset.

Returns:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.
- `metrics`: The evaluation metrics result based on the predictions and annotations.
- `dataset`: An instance of the testing dataset.

```
def predict_ed(trainer: Union[Trainer, Seq2SeqTrainer],
               tokenizer: PreTrainedTokenizer,
               data_class: type,
               data_args,
               data_file: str,
               ) -> Tuple[np.array, np.array, Dict, EDDataProcessor]:
    """Predicts the test set of the event detection task.
    Predicts the test set of the event detection task. The prediction of logits and␣
↪labels, evaluation metrics' results,
    and the dataset would be returned.
    Args:
```

(continues on next page)

(continued from previous page)

```
        trainer:
            The trainer for event detection.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
        data_class:
            The processor of the input data.
        data_args:
            The pre-defined arguments for data processing.
        data_file (`str`):
            A string representing the file path of the dataset.
    Returns:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels: (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
        metrics:
            The evaluation metrics result based on the predictions and annotations.
        dataset:
            An instance of the testing dataset.
    """
    dataset = data_class(data_args, tokenizer, data_file)
    logits, labels, metrics = trainer.predict(
        test_dataset=dataset,
        ignore_keys=["loss"]
    )
    return logits, labels, metrics, dataset
```

6.24.7 predict_sub_ed

Predicts the test set of the event detection task of a list of datasets. The prediction of logits and labels are conducted separately on each file, and the evaluation metrics' results are calculated after concatenating the predictions together. Finally, the prediction of logits and labels, evaluation metrics' results, and the dataset would be returned.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: The tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `data_args`: The pre-defined arguments for data processing.
- `data_file`: A string representing the file path of the dataset.

Returns:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.
- `metrics`: The evaluation metrics result based on the predictions and annotations.
- `dataset`: An instance of the testing dataset.

```
def predict_sub_ed(trainer: Union[Trainer, Seq2SeqTrainer],
                   tokenizer: PreTrainedTokenizer,
                   data_class: type,
                   data_args: DataArguments,
                   data_file: str,
                   ) -> Tuple[np.array, np.array, Dict, EDDataProcessor]:
    """Predicts the test set of the event detection task of subfile datasets.
    Predicts the test set of the event detection task of a list of datasets. The␣
↪prediction of logits and labels are
    conducted separately on each file, and the evaluation metrics' results are calculated␣
↪after concatenating the
    predictions together. Finally, the prediction of logits and labels, evaluation␣
↪metrics' results, and the dataset
    would be returned.
    Args:
        trainer:
            The trainer for event detection.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
        data_class:
            The processor of the input data.
        data_args:
            The pre-defined arguments for data processing.
        data_file (`str`):
            A string representing the file path of the dataset.
    Returns:
        logits (`np.ndarray`):
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels: (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
        metrics:
            The evaluation metrics result based on the predictions and annotations.
        dataset:
            An instance of the testing dataset.
    """
    data_file_full = data_file
    data_file_list = get_sub_files(input_test_file=data_file_full,
                                   sub_size=data_args.split_infer_size)

    logits_list, labels_list = [], []
    for data_file in tqdm(data_file_list, desc='Split Evaluate'):
        data_args.truncate_in_batch = False
        logits, labels, metrics, _ = predict_ed(trainer, tokenizer, data_class, data_
↪args, data_file)
        logits_list.append(logits)
        labels_list.append(labels)

    logits = np.concatenate(logits_list, axis=0)
    labels = np.concatenate(labels_list, axis=0)

    metrics = trainer.compute_metrics(logits=logits, labels=labels,
```

(continues on next page)

(continued from previous page)

```
                                        **{"tokenizer": tokenizer, "training_args":␣
↪trainer.args})

    dataset = data_class(data_args, tokenizer, data_file_full)
    return logits, labels, metrics, dataset
```

6.24.8 predict_eae

Predicts the test set of the event argument extraction task. The prediction of logits and labels, evaluation metrics' results, and the dataset would be returned.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: A string indicating the tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `data_args`: The pre-defined arguments for data processing.
- `training_args`: The pre-defined arguments for the training process.

Returns:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.
- `metrics`: The evaluation metrics result based on the predictions and annotations.
- `test_dataset`: An instance of the testing dataset.

```
def predict_eae(trainer: Union[Trainer, Seq2SeqTrainer],
                tokenizer: PreTrainedTokenizer,
                data_class: type,
                data_args: DataArguments,
                training_args: TrainingArguments,
                ) -> Tuple[np.array, np.array, Dict, EAEDataProcessor]:
    """Predicts the test set of the event argument extraction task.
    Predicts the test set of the event argument extraction task. The prediction of␣
↪logits and labels, evaluation
    metrics' results, and the dataset would be returned.
    Args:
        trainer:
            The trainer for event detection.
        tokenizer (`PreTrainedTokenizer`):
            A string indicating the tokenizer proposed for the tokenization process.
        data_class:
            The processor of the input data.
        data_args:
            The pre-defined arguments for data processing.
        training_args:
            The pre-defined arguments for the training process.
    Returns:
        logits (`np.ndarray`):
```

(continues on next page)

(continued from previous page)

```
            An numpy array of integers containing the predictions from the model to be␣
↪decoded.
        labels: (`np.ndarray`):
            An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
        metrics:
            The evaluation metrics result based on the predictions and annotations.
        test_dataset:
            An instance of the testing dataset.
    """
    test_dataset = data_class(data_args, tokenizer, data_args.test_file, data_args.test_
↪pred_file)
    training_args.data_for_evaluation = test_dataset.get_data_for_evaluation()
    logits, labels, metrics = trainer.predict(test_dataset=test_dataset, ignore_keys=[
↪"loss"])

    return logits, labels, metrics, test_dataset
```

6.24.9 predict_sub_eae

Predicts the test set of the event detection task of a list of datasets. The prediction of logits and labels are conducted separately on each file, and the evaluation metrics' results are calculated after concatenating the predictions together. Finally, the prediction of logits and labels, evaluation metrics' results, and the dataset would be returned.

Args:

- `trainer`: The trainer for event detection.
- `tokenizer`: The tokenizer proposed for the tokenization process.
- `data_class`: The processor of the input data.
- `data_args`: The pre-defined arguments for data processing.
- `training_args`: The pre-defined arguments for the training process.

Returns:

- `logits`: An numpy array of integers containing the predictions from the model to be decoded.
- `labels`: An numpy array of integers containing the actual labels obtained from the annotated dataset.
- `metrics`: The evaluation metrics result based on the predictions and annotations.
- `test_dataset`: An instance of the testing dataset.

```
def predict_sub_eae(trainer: Union[Trainer, Seq2SeqTrainer],
                    tokenizer: PreTrainedTokenizer,
                    data_class: type,
                    data_args: DataArguments,
                    training_args: TrainingArguments,
                    ) -> Tuple[np.array, np.array, Dict, EDDataProcessor]:
    """Predicts the test set of the event detection task of subfile datasets.
    Predicts the test set of the event detection task of a list of datasets. The␣
↪prediction of logits and labels are
    conducted separately on each file, and the evaluation metrics' results are calculated␣
```

(continues on next page)

(continued from previous page)

```
    after concatenating the
        predictions together. Finally, the prediction of logits and labels, evaluation␣
↪metrics' results, and the dataset
        would be returned.
        Args:
            trainer:
                The trainer for event detection.
            tokenizer (`PreTrainedTokenizer`):
                A string indicating the tokenizer proposed for the tokenization process.
            data_class:
                The processor of the input data.
            data_args:
                The pre-defined arguments for data processing.
            training_args:
                The pre-defined arguments for the training process.
        Returns:
            logits (`np.ndarray`):
                An numpy array of integers containing the predictions from the model to be␣
↪decoded.
            labels: (`np.ndarray`):
                An numpy array of integers containing the actual labels obtained from the␣
↪annotated dataset.
            metrics:
                The evaluation metrics result based on the predictions and annotations.
            test_dataset:
                An instance of the testing dataset.
        """
        test_file_full, test_pred_file_full = data_args.test_file, data_args.test_pred_file
        test_file_list, test_pred_file_list = get_sub_files(input_test_file=test_file_full,
                                                            input_test_pred_file=test_pred_
↪file_full,
                                                            sub_size=data_args.split_infer_
↪size)

        logits_list, labels_list = [], []
        for test_file, test_pred_file in tqdm(list(zip(test_file_list, test_pred_file_list)),
↪ desc='Split Evaluate'):
            data_args.test_file = test_file
            data_args.test_pred_file = test_pred_file

            logits, labels, metrics, _ = predict_eae(trainer, tokenizer, data_class, data_
↪args, training_args)
            logits_list.append(logits)
            labels_list.append(labels)

        # TODO: concat operation is slow
        logits = np.concatenate(logits_list, axis=0)
        labels = np.concatenate(labels_list, axis=0)

        test_dataset_full = data_class(data_args, tokenizer, test_file_full, test_pred_file_
↪full)
        training_args.data_for_evaluation = test_dataset_full.get_data_for_evaluation()
```

(continues on next page)

(continued from previous page)

```
    metrics = trainer.compute_metrics(logits=logits, labels=labels,
                                      **{"tokenizer": tokenizer, "training_args":␣
↪training_args})

    data_args.test_file = test_file_full
    data_args.test_pred_file = test_pred_file_full

    test_dataset = data_class(data_args, tokenizer, data_args.test_file, data_args.test_
↪pred_file)
    return logits, labels, metrics, test_dataset
```