**Описание работы алгоритма:**

1. В файле `data\_process.py` определен класс `DataTransform`, который выполняет обработку текста и поиск сущностей.

2. Класс `DataTransform` инициализируется с заданными параметрами модели, пути к модели и токенизатору.

3. В методе `load\_model` модель и токенизатор загружаются в память и устанавливаются на доступное устройство (GPU, если доступен).

4. Метод `preprocess\_text` осуществляет предобработку текста: разбиение на слова, токенизацию и преобразование входных данных в нужный формат (тензоры).

5. В методе `split\_text\_with\_overlap` происходит разделение текста на части с перекрытием.

6. Метод `get\_entities` принимает на вход текст и возвращает найденные сущности и их индексы.

- С помощью метода `split\_text\_with\_overlap` текст разделяется на части с перекрытием.

- Каждая часть текста подвергается предобработке с помощью метода `preprocess\_text`.

- Тензоры входных данных передаются в модель для предсказания меток.

- Полученные предсказанные метки обрабатываются методом `get\_entities\_with\_labels` для объединения токенов в слова, с получением меток сущностей.

- Результаты для каждой части текста объединяются и возвращаются в виде словарей с позициями и списком меток.

7. Метод `get\_entities\_with\_labels` объединяет токены в сущности с метками.

- Используется особая обработка для токенов от токенизатора DebertaV2Tokenizer.

- Токены объединяются в сущности с метками, сохраняются в список.

- Производится определение основной метки и добавление кортежей (слово, метка, индекс) в результаты.

- Возвращаются словари с позициями меток и списки результатов.

8. Метод `merge\_chunks\_results` объединяет результаты из перекрывающихся частей в один результат.

- Переданные позиции меток объединяются без пустых значений.

- Метки результата объединяются с учетом оригинальной длины текста.

- Возвращаются объединенные метки и результаты.

9. Метод `predict\_entities` принимает на вход датафрейм и выполняет поиск сущностей в каждой строке.

- Для каждой строки датафрейма инициализируется входной текст и находятся истинные метки сущностей.

- Текст проходит через метод `get\_entities` для поиска сущностей и получения предсказанных меток и позиций.

- Полученные результаты добавляются в соответствующие списки.

- Истинные и предсказанные метки добавляются в датафрейм.

- Возвращается датафрейм с заполненными столбцами и списки с истинными и предсказанными метками.

10. В файле `main\_predict.py` приведен пример использования класса `DataTransform` для предсказания сущностей по входным данным.

- При инициализации экземпляра класса `DataTransform` задаются:

* + - model\_name=имя\_модели
    - model\_path=путь\_к\_предобученной\_модели
    - tokenizer=токенизатор
    - token\_classification=классификатор

класс `DataTransform` по умолчанию настроен для использования моделей семейства BERT с моделью, расположенной в каталоге './model'. В этом каталоге должны лежать все файлы, относящиеся к модели: предобученные веса, концигурации токенизатора и модели, словарь токенизатора.

Варианты инициализации экземпляра класса:

1. Используем настройки по умолчанию для BERT:

dps = DataTransform()

1. Задаем имя модели и путь к предобученной модели BERT:

bert\_name = 'bert-base-cased'

bert\_tuned = './results/checkpoint-last'

dps = DataTransform(model\_name=bert\_name, model\_path=bert\_tuned)

1. Задаем все параметры при использовании модели DeBERTa:

bert\_name = 'microsoft/mdeberta-v3-base'

bert\_tuned = './results/checkpoint-last'

dps = DataTransform(

model\_name=bert\_name,

model\_path=bert\_tuned,

tokenizer=DebertaV2Tokenizer,

token\_classification=DebertaV2ForTokenClassification

)

- Загружается тестовый датафрейм из файла gt\_test.csv'.

- Итерируясь по каждой строке датафрейма, извлекается входной текст.

- С использованием метода `get\_entities` предсказываются метки сущностей и сохраняются в датафрейм 'result'.

Таким образом, данный алгоритм использует модель BERT или DeBERTa для предсказания меток сущностей в тексте.

Класс DataTransform предоставляет методы для загрузки модели, препроцессинга текста, поиска сущностей и формирования предсказанных и истинных меток сущностей. Пример использования класса показывает процесс предсказания меток сущностей для текстов из заданного датафрейма и сохранение результата в файл.

**Инструкция по развертыванию и использованию модели.**

Работоспособность модели проверена на Python 3.10 и 3.11.

1. Предполагается, что файл с тестовыми данными лежит в data/ - каталоге с данными, туда же будет сформирован файл «submission.csv» с предсказанными метками слов.
2. В файле requirements.txt необходимые модули для работы.

pip install --no-cache-dir -r requirements.txt

1. Структура проекта при использовании моделей семейства BERT:

predict.bat – командный файл для получения сабмита

aeroplane/ - каталог проекта

├── index.html

├── api\_app.py

├── data\_process.py

├── main\_predict.py

├── main\_predict\_mdeberta.py

├── ner\_testing.py

├── data/ - каталог с данными

│ ├── gt\_test.csv - тестовые данные для предсказаний

├── model/ - каталог с моделью семейства BERT

│ ├── config.json

│ ├── model.safetensors

│ ├── special\_tokens\_map.json

│ ├── tokenizer\_config.json

│ ├── vocab.txt

├── model\_mdb/ - каталог с моделью DeBERTa

│ ├── config.json

│ ├── model.safetensors

│ ├── spm.model

│ ├── tokenizer\_config.json

├── requirements.txt

├── test.csv

└── Dockerfile

1. Получение предсказаний с помощью модели из файла gt\_test.csv:
   1. Получение предсказаний с помощью командного файла:

**predict.bat**

по умолчанию используется модель DeBERTa:

@cd aeroplane

@rem use model BERT:

@rem @python main\_predict.py <-- BERT

@rem use model DeBERTa:

@python main\_predict\_mdeberta.py <-- DeBERTa

* 1. Получение предсказаний с помощью модели BERT:

**python.exe main\_predict.py**

* 1. Получение предсказаний с помощью модели DeBERTa:

**python.exe main\_predict\_mdeberta.py**

В результате работы скрипта будет создан файл submission.csv.

1. Создание докер-контейнера:
   1. Перейдите в директорию с проектом и выполнить команду:

**docker build -t aeroplane .**

* 1. После создания Docker образа, запустите контейнер:

**docker run -d -p 8000:8000 --name aeroplane aeroplane\_app**

1. Проверка работоспособности приложения скриптом на Python:

**python.exe api\_app\_reqs.py**

1. Проверка работоспособности приложения с помощью curl:

curl -X POST "http://localhost:8000/ner" -H "Content-Type: application/json" -d "{\"text\": \"Ваш текст\"}"

1. Работа с приложением через браузер:

В браузере открыть файл «index.html», ввести текст, нажать кнопку «Submit» и получить список предсказанных меток.

**Документация по API.**

API приложения «api\_app.py» реализовано с помощью библиотеки FastAPI и содержит несколько строк кода, все необходимые для работы библиотеки указаны в файле «requirements.txt».

В качестве настойки можно добавить адреса в список «origins»:

# Разрешение CORS  
origins = [  
 "http://localhost",  
 "http://localhost:8000",  
 "http://127.0.0.1",  
 "http://127.0.0.1:8000",  
 "null" # Для покрытия случаев, когда файлы открыты напрямую в браузере  
]

Без сборки докер-контейнера приложение можно запустить командой:

uvicorn api\_app:app --reload

Проверка работоспособности приложения описана выше в пунктах 6 и 7.

**Тестирование модели.**

Проверка работоспособности модели реализована в скрипте ner\_testing.py. Используется набор тестов для класса DataTransform, написанный с использованием модуля unittest.

Для каждого метода класса DataTransform написан тест с несколькими проверками. В процессе написания тестов для методов класса исправлялись ошибки, возникающие при тестировании.

- test\_preprocess\_text: тест проверяет метод preprocess\_text. Создается текст, который нужно токенизировать, и сравнивается результат работы метода с ожидаемыми значениями. Проверяются слова, токены, идентификаторы и маска внимания.

- test\_split\_text\_with\_overlap: тест проверяет метод split\_text\_with\_overlap, который разделяет текст на части с перекрытием. Проверяется количество частей, их длина и соответствие ожидаемым значениям.

- test\_get\_words\_positions: проверяется метод get\_words\_positions, который ищет слова, соответствующие заданному шаблону (скидки) и возвращает их позиции.

- test\_get\_entities\_with\_labels: проверяется метод get\_entities\_with\_labels, который объединяет токены в сущности с метками.

- test\_merge\_chunks\_results: проверяется метод merge\_chunks\_results, который объединяет результаты из перекрывающихся частей текста.

- test\_transform\_text\_labels: проверяется метод transform\_text\_labels, который формирует список меток для каждого слова в тексте.

- test\_get\_entities: проверяется метод get\_entities, который ищет сущности в тексте и возвращает найденные метки и их позиции.

**Описание работы алгоритма процесса обучения модели:**

Код обучения NER-модели представлен в тетрадке jupiter notebook lct-task-16-berta-overlaps.ipynb - для моделей семейства BERT и

lct-task-16-mdeberta-overlaps.ipynb - для моделей семейства DeBERTa.

Этот код выполняет обучение модели для задачи последовательного классифицирования с использованием предобученной модели BERT. Он включает в себя этапы предварительной обработки данных, разделения на обучающие и валидационные наборы, токенизации и меток, и затем обучает модель на нескольких фолдах с использованием перекрестной проверки. Далее представлен пошаговый обзор работы кода:

### Импорт библиотек и установка параметров

Код импортирует необходимые библиотеки для работы с данными, обучения модели и проведения перекрестной проверки. Устанавливает параметры, такие как максимальная длина входной последовательности токенов `MAX\_LEN` и размер перекрытия `OVERLAP` предложений, устройство для обучения модели (GPU, если доступно), и название предобученной модели BERT.

### Чтение и подготовка данных

1. \*\*Чтение данных:\*\* Данные читаются из CSV файла с помощью `pandas`. Поле `target\_labels\_positions` преобразуется в словарь с помощью модуля `ast`.

2. \*\*Создание меток:\*\* Для каждого предложения создаются соответствующие метки на основе `target\_labels\_positions`.

3. \*\*Токенизация:\*\* Используется токенизатор BERT для преобразования слов в токены, сохраняя метки.

### Разделение предложений с перекрытием

Функция `split\_sentence\_with\_overlap` разделяет длинные предложения на части с перекрытием, чтобы длина одной части не превышала `MAX\_LEN` токенов. Это нужно для того, чтобы обрабатывать длинные предложения, которые не помещаются в ограничение длины модели BERT.

### Токенизация и метки

Функция `tokenize\_and\_preserve\_labels` токенизирует слова и сохраняет метки для каждого токена, чтобы они соответствовали токенам.

### Подготовка данных для обучения

1. \*\*Преобразование в индексы:\*\* Токенизированные тексты и метки преобразуются в индексы с помощью `pad\_sequences`, чтобы длина каждой последовательности была равна `MAX\_LEN`.

2. \*\*Создание масок внимания:\*\* Создаются маски внимания, чтобы модель учитывала только значимые токены, игнорируя padding.

### Определение и обучение модели

1. \*\*Инициализация модели и оптимизатора:\*\* Функция `get\_model` создает модель BERT для токен-классификации и инициализирует оптимизатор `AdamW`.

2. \*\*Тренировочная функция:\*\* Функция `train\_model` обучает модель на обучающих данных и оценивает ее на валидационных данных. Она сохраняет лучшую модель на основе значения F1-score.

### Перекрестная проверка

1. \*\*KFold разбиение:\*\* Данные делятся на 5 фолдов для проведения перекрестной проверки с помощью `KFold`.

2. \*\*Обучение на фолдах:\*\* Для каждого фолда данные делятся на обучающую и валидационную части, модель обучается, и результат сохраняется.

3. \*\*Оценка и сохранение модели:\*\* Лучшая модель сохраняется на диск. Процесс повторяется для каждого фолда и выбирается лучшая модель. Анализ логов обучения показал, что в более чем 80% случаев наибольший скор получался у моделей, обученных на первом фолде.