ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ»**

**(ФИНАНСОВЫЙ УНИВЕРСИТЕТ)**

Департамент анализа данных

и машинного обучения

***Дисциплина: «Технологии анализа данных и машинного обучения»***

*Направление подготовки: «Прикладная математика и информатика»*

*Профиль: «Анализ данных и принятие решений в экономике и финансах»*

*Факультет информационных технологий и анализа больших данных*

*Форма обучения очная*

*Учебный 2022/2023 год, 6 семестр*

**Курсовая работа на тему:**

«Использование нейросетевых технологий для решения задачи выделения атрибутов товаров из текста»

*Выполнил(а):*

студентка группы ПМ20-2

Короткова В. В.

*Научный руководитель:*

ассистент Блохин Н.В.

**Москва 2023**

СОДЕРЖАНИЕ

[**ВВЕДЕНИЕ** 2](#_Toc134698735)

[**ГЛАВА 1. ПРОБЛЕМА РАСПОЗНАВАНИЯ ИМЕНОВАННЫХ СУЩНОСТЕЙ И ПОДХОДЫ К ЕЕ РЕШЕНИЮ** 3](#_Toc134698736)

[1.1 Задача распознавания именованных сущностей (NER) 3](#_Toc134698737)

[1.2 Использование искусственных нейронных сетей для решения задачи машинного обучения 5](#_Toc134698738)

[1.3 Рекуррентные нейронные сети 8](#_Toc134698739)

[**ГЛАВА 2. РЕАЛИЗАЦИЯ ПРОГРАММНОГО КОМПЛЕКСА ДЛЯ ВЫДЕЛЕНИЯ ХАРАКТЕРИСТИК ИЗ НАЗВАНИЙ ТОВАРОВ** 13](#_Toc134698740)

[2.1 Подготовка набора данных с информацией о товарах 13](#_Toc134698741)

[2.2 Предобработка данных 15](#_Toc134698742)

[2.3 Использование предобученной модели BERT для решения задачи NER 16](#_Toc134698743)

[2.4 Разработка и настройка рекуррентной сети с использованием слоев LSTM и GRU для решения задачи NER. 19](#_Toc134698744)

[2.5 Результаты 21](#_Toc134698745)

[**ЗАКЛЮЧЕНИЕ** 25](#_Toc134698746)

[**СПИСОК ИСТОЧНИКОВ** 26](#_Toc134698747)

[**ПРИЛОЖЕНИЕ А. ОПИСАНИЕ ПРОЦЕССОРА** 27](#_Toc134698748)

[**ПРИЛОЖЕНИЕ Б. ГЕНЕРАЦИЯ ДАТАСЕТОВ** 28](#_Toc134698749)

[**ПРИЛОЖЕНИЕ В. ОСНОВНАЯ ПРОГРАММА** 29](#_Toc134698750)

# **ВВЕДЕНИЕ**

Для развития крупным компаниям требуется автоматизация различных задач. В том числе необходима обработка и анализ больших объемов текстовых данных. Данная работа посвящена использованию нейросетевых технологий для решения задачи выделения атрибутов товаров из текста. Это может, например, упростить для пользователей процесс поиска необходимых товаров на сайтах.

Будет рассмотрена постановка задачи NER, а также способы ее решения. Целью работы является самостоятельная разработка нескольких моделей глубокого обучения для автоматического распознавания именованных сущностей. Задачи, которые будут выполнены в работе, следующие: для решения задачи NER будут использоваться уже популярные слои LSTM и GRU, а также предобученная модель BERT. Для получения набора данных, необходимого для обучения модели, была проведена процедура разметки данных, алгоритм которой будет приведен в работе. Для проверки точности будут использоваться такие метрики, как accuracy и f1-score.

В последнее время большая часть работ по NLP в глубоком машинном обучении посвящены генерации текстов, классической задаче NER уделяется меньше внимания. Актуальность данной работы состоит в том, что ранее не было представлено модели, которая распознает характеристики товаров в их названиях.

Объектом исследования являются названия товаров с торговой площадки. А предметом исследования является задача распознавания именованных сущностей по этим данным.

# **ГЛАВА 1. ПРОБЛЕМА РАСПОЗНАВАНИЯ ИМЕНОВАННЫХ СУЩНОСТЕЙ И ПОДХОДЫ К ЕЕ РЕШЕНИЮ**

## Задача распознавания именованных сущностей (NER)

Named Entity Recognition (или выделение именованных сущностей) является одной из самых известных задач обработки естественного языка. Она заключается в выделении в тексте слов и их классификации по различным категориям. Для каждого набора данных эти категории различаются. Несколько самых частых [1] представлены на рис.1:

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 1. Самые популярные категории с примерами

Целью задачи NER является нахождение и идентификация каждой сущности по категориям. Поэтому ее можно разбить на три этапа:

1. определение категорий (классов) в датасете;
2. токенизация текста;
3. классификация токенов, т.е. слов или их последовательностей.

Распознавание именованных сущностей обычно используется для решения таких задач, как:

* создание вопросно-ответных систем – улучшается точность и качество поиска благодаря выводу только не целых страниц, а только тех частей, в которых находится искомая информация;
* Машинный перевод - определение терминологии, которую нужно переводить на другой язык, а также для определения контекста и смысла слов в предложении;
* Обработка текста - в грамматике могут использоваться именованные сущности для определения падежа, рода и числа существительных, а также для определения формы глаголов
* Анализ новостных статей - NER может помочь автоматически идентифицировать имена, места, даты и другие сущности в новостных статьях, что может быть полезно для анализа тенденций и событий.
* Автоматизированный анализ данных - в коммерческих целях может использоваться для анализа текста, включая анализ статей, отчетов и других документов, для локализации ключевой информации.

Но NER не используется повсеместно из-за существования разных сложностей ее внедрения:

* Неоднозначности, например, слово «август» может быть названием месяца или именем. Для решения этой проблемы нужен большой обучающий набор данных, что является второй проблемой.
* Необходимость большого количества размеченных текстов для обучения.
* Проблема с многоязычностью – так как различные языки имеют разные правила и особенности, поэтому модели NER, обученные на одном языке, могут работать гораздо хуже на другом.
* Именованные сущности редко состоят из одного слова, возникает проблема определения их границ. Чтобы избежать эту проблему при разметке обозначаются начало (и в некоторых случаях конец) сущности специальными символами, например, <NAME\_START>, >NAME\_END>.

В данной работе задача выделения именованных сущностей будет решаться с помощью глубокого обучения, но это не всегда было так. При появлении первых решений использовался подход, основанный на использовании набора правил для обнаружения и классификации именованных сущностей в тексте. Он может быть эффективным в случае, если структура данных ограничена определенным набором правил. Однако этот подход неэффективен в случае, если задача требует обработки большого объема текстов.

Несмотря на то, что сейчас для решения задачи NER используются архитектура Transformer, более классическим подходом является использование рекуррентных нейронных сетей, так как происходит работа с последовательностями. Такой подход основан на использовании таких архитектур глубоких систем, как LSTM и GRU. Решение данной задачи с помощью глубокого обучения является самым точным. В общих чертах, глубокое обучение для задачи NER предполагает обучение модели на большом корпусе текстов, где каждое слово помечено тегом, указывающим, является ли это слово именованной сущностью или нет. В процессе обучения модель изучает общие признаки, которые помогают ей определять именованные сущности в новых текстах.

## 1.2 Использование искусственных нейронных сетей для решения задачи машинного обучения

Искусственная нейронная сеть – это математическая модель, а также её реализации, построенная по образу и подобию нервных клеток живого организма. Она состоит из слоёв нейронов, которые принимают на вход информацию, обрабатывают её и передают дальше по сети, которая связывает слои нейронов. В своём учебнике Голдберг пишет, что глубокое обучение – это раздел машинного обучения, который использует нейронные сети [2]. Оно решает различные задачи, например: компьютерное зрение, обработка естественного языка, в т.ч. генерация текстов, распознавание образов и т.д.

Для обработки естественного языка существует множество методов, не использующие нейронные сети. Но только модели глубоко обучения могут «понимать» язык так, как это делает человек.

Использование нейронных сетей в NLP позволяет создавать более точные и эффективные модели для обработки естественного языка. Однако, для их использования необходимо иметь большой объем данных для обучения и достаточно мощные вычислительные ресурсы, потому что в задачах глубокого обучения язык рассматривается как последовательность слов.

Любая нейронная сеть состоит из слоев нейронов, функции активации, функции потерь и оптимизатора. Что это такое, а также какие виды перечисленных компонент используются для задачи NER будет рассмотрено далее в этой главе. На рис. 2 схематично представлен цикл обучения.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 2. Логика обучения нейронной сети

Функции активации – это нелинейности, вводимые в нейронную сеть для того, чтобы уловить сложные взаимосвязи данных. Они представляют собой различные математические операции, которые применяются к выходам нейронов на определенном скрытом слое. Для каждого слоя можно ввести разные функции активации. Без их использования нейронная сеть не смогла бы решать сложные задачи.

По информации из Википедии [3], на данный момент существует больше 20 разных функций активации, каждая из которых лучше других решает определенную задачу и имеет свои особенности. Самыми популярными являются следующие: сигмоидная функция, гиперболический тангенс, ReLU и SoftMax.

В данной работе, посвященной решению задачи NER, для активации линейного слоя используется ReLU из-за её популярности, простоты и возможности моделировать нелинейные связи. Она имеет следующий вид:

Функция потерь является мерой отклонения предсказанных значений от настоящих. Во время цикла обучения модель стремится уменьшить ее значение. Формально функция потерь сопоставляет числовую оценку (скаляр) предсказанному выходу yˆ, если известен истинный выход y.

Как и функций активации, функций потерь существует уже достаточно много. Самыми известными из них являются: Mean Squared Loss, Cross Entropy Loss, логарифмическая и т.д.

В данной работе используется преобразованная версия Cross Entropy Loss под названием Focal Loss [4]. Из-за особенностей задачи NER и исследуемого датасета в нем наблюдается достаточно значительный дисбаланс классов. Такая функция потерь придает больший вес тем классам, которые встречаются реже. А самому частому классу (тэг <O>, который обозначает, что токен является неинтересным для нас) присваивается минимальный вес. Благодаря этой функции увеличивается точность. Математически она рассчитывается с помощью формул (2.1 – 2.3):

В течение цикла обучения модель предсказывает значения целевой переменной, функция активации применяет математические операции к нейронам, функция потерь считает расхождения предсказанных и наблюдаемых значений. Оптимизаторы подсчитывают градиенты (частные производные), и на основе этих значений обновляет веса в конце цикла обучения. Основными примерами оптимизаторов являются стохастический градиентный спуск и Adam. На 2020 год было эмпирически доказано превосходство Адама над другими оптимизаторами [5]. Именно поэтому он используется в данной работе. Причины для этого следующие: он является самым оптимальным, устойчив к выбору гиперпараметров, имеет меньшую вероятность переобучения, а также имеет большую скорость, чем SGD.

## 1.3 Рекуррентные нейронные сети

Для решения задач обработки естественного языка используются рекуррентные нейронные сети. Они предназначены именно для работы с временными последовательностями данных (видео, звук), а не только текстами. Главной особенностью RNN является тот факт, что на вход каждому последующему скрытому слою подаются не только значения прошлого слоя, а всех предыдущих. Они созданы так, чтобы уметь запоминать важную информацию, а лишнюю отбрасывать. На рис.3 представлена схема RNN

Изображение выглядит как текст, часы

Автоматически созданное описание

Рисунок 3. Принцип устройства узла RNN

Рекуррентные сети – это семейство моделей глубокого машинного обучения. Простейшим вариантом является RNN Элмана, но в настоящее время она уже почти не используется. Популярными сейчас моделями являются слои LSTM и GRU, подробнее они будут описаны далее.

Стоит уточнить, что векторы различной длины нельзя объединить в тензор поэтому в данной работы все предложения были дополнены до максимальной длины с помощью таких тэгов, как <PAD> (padding).

При изучении рекуррентных нейронных сетей обязательно понимать, что такое эмбеддинги – векторные представления слов [6]. Обычно в модели для получения эмбеддингов токенов используется специальный слой. Этот слой преобразует каждый токен в предложении в вектор заданной нами размерности, что даёт модели возможность ощутить близость слов по значениям посредством расстояний между векторами.

Рекуррентные сети чаще всего имеют следующую архитектуру, данная работа ей соответствует: слой эмбеддингов -> модель -> регуляризация (необязательна) -> нормализация (необязательна) -> линейный слой -> функция активации -> линейный слой. Количество линейных скрытых слоев неограниченно, но стоит помнить, что после выходного линейного слоя функция активации не требуется.

Как говорилось выше, рекуррентные сети являются семейством моделей, которые работают с последовательностями. В последнее время чаще всего используются слои LSTM и GRU, именно им посвящен данный раздел работы.

LSTM (Long Short-Term Memory) – это модель долгой краткосрочной памяти. Она была создана в 1997 году Хохрейтером и Шмидхубером для решения проблемы затухающего градиента в рекуррентных нейронных сетях, а также для возможности повторно использовать прошлую информацию в обучении.

Пошаговый разбор работы ячейки LSTM представлен в статье Кристофа Олаха от 2015 года [7].

В ячейке LSTM три основных вида узлов: входной, забывающий и выходной. Основные шаги для работы:

1. Решить, какую информацию забыть
2. Решить, какую информацию запомнить
3. Обновить старое состояние ячейки в новое
4. Вывести отфильтрованную информацию

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 4. Структура ячейки LSTM

В 2014 году появилась еще одна версия модели GRU (Gated Recurrent Unit). Её главное отличие заключается в том, что блок забывания объединен с выходным блоком. Из-за этого модели со слоем GRU имеют меньше параметром, вследствие чего быстрее обучаются.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 5. Структура ячейки GRU

У ячеек LSTM и GRU есть и другие версии BI-LSTM и BI-GRU, то есть их двунаправленные варианты. По сути, это та же ячейка, но удвоенная. В первой сигнал идет в прямом направлении, а во второй – в обратном. Затем результаты объединяются в один выход. Это сделано для того, чтобы модель могла учитывать контекст с двух сторон последовательности. Логика работы таких ячеек представлена на рис. 6 и рис. 7

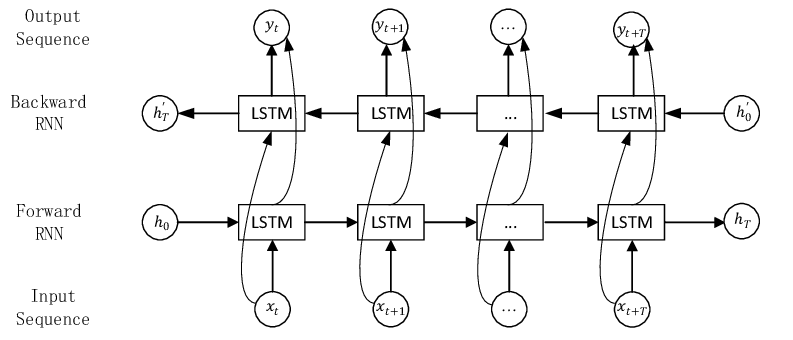


Рисунок 6. Логика работы BI-LSTM

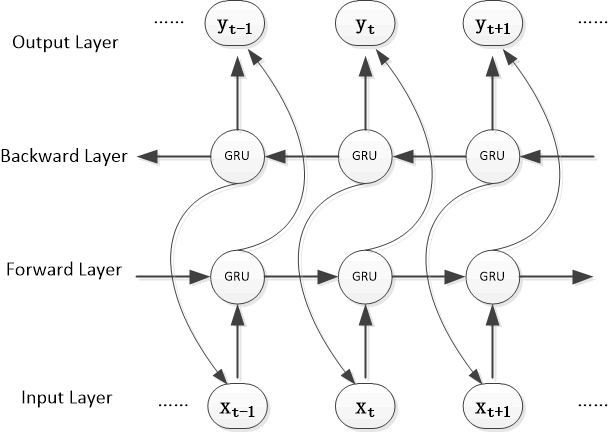


Рисунок 7. Логика работы BI-GRU

Кроме перечисленных выше слоев используются слои BI-LSTM-CRF и BI-GRU-CRF. CRF, или вероятностная целевая функция – один из самых популярных статистических методов, использующихся в данной работе. В статье “Bidirectional LSTM-CRF Models for Sequence Tagging” Zhiheng Huang подробно описано, как такие слои применяются, и какие они дают результаты [8].

CRF является марковской моделью и моделирует зависимости между метками последовательности учитывая контекст, что позволяет улучшить качество предсказаний и уменьшить количество ошибок.

Но в данной исследовательской работе такая модель не будет использоваться по причине отсутствия готовых реализаций этого слоя в библиотеке PyTorch.

# **ГЛАВА 2. РЕАЛИЗАЦИЯ ПРОГРАММНОГО КОМПЛЕКСА ДЛЯ ВЫДЕЛЕНИЯ ХАРАКТЕРИСТИК ИЗ НАЗВАНИЙ ТОВАРОВ**

## 2.1 Подготовка набора данных с информацией о товарах

Для выполнения данной работы использовался датасет, состоящий из названий товаров с сайта supl biz [11] (торговая площадка для покупателей и продавцов). Изначально он представляет собой json-файл со 157 тысячью товаров.

Так как эта работа посвящена задаче распознавания именованных сущностей – основной целью подготовки набора данных было выделить категории, к которым относятся токены. Это можно попытаться сделать с помощью характеристик товаров. Но, видимо, из-за того, что каждый продавец вносит информацию на сайт самостоятельно – формулировка атрибутов в информации о товарах почти всегда имеет разную структуру. Ниже представлено несколько примеров для лучшего понимания проблемы:

* "Title": "1112.5208500-02 Форсунки омывателя лобового стекла с креплением (кт.)", "Характеристики: Гарантийный срок: 12 Смотреть товар на сайте поставщика". В данном случае с помощью характеристик нельзя выделить ни один тэг
* "Title": "Внешний жесткий диск Transcend TS2TSJ25M3S 2Tb", Характеристики: Тип устройства: Внешний жесткий диск Цвет: None Смотреть товар на сайте поставщика". Для данного товара можно было бы выделить только что это за тип.
* "Title": "Аренда Автовышки 15 м", "Характеристики": "Высота подъема — 15м", "Грузоподъемность люльки — 200кг". Здесь можно было бы выделить высоту подъема, но она есть у очень маленького числа товаров, поэтому не имеет смысла.

Поэтому было принято решение провести разметку вручную по небольшой части датасета. В итоге случайным образом было выбрано около 3000 товаров, названия которых составляют в сумме 20000 слов, каждому из которых была присвоена своя категория вручную.

Для изучаемого датасета были выбраны следующие категории – тэги:

* ‘O’ – так называемое ‘ничто’, этот присваивается в случае, если токен не несет никакой полезной информации
* ‘CMPN’ – название компании-производителя
* ‘MDL’ – модель, артикул товара
* ‘COL’ – цвет товара
* ‘MAT’ – материал, из которого сделан товар или его часть
* ‘NUM’ – присваивается, если речь идет о количестве предметов
* ‘SIZE’ – все, что связано с размерами товара (высота, длина, ширина, радиус, глубина, вес и т.д.)
* ‘CAP’ – все, что связано с мощностью товара (вместимость, мощность, скорость, количество оборотов в минуту и т.д.)

Стоит также отметить, что если несколько токенов подряд относятся к одной категории, то первому из тэгов присваивается слово START, как и было упомянуто в теоретической части работы.

Перед разметкой все названия были поделены на токены с помощью токенизатора из библиотеки nltk. А затем был составлен файл ‘bert\_made.xlsx’, в котором в первом столбце название повторяется столько строк подряд, сколько в нем токенов, во втором столбце сами токены, а третий предназначен для тэгов (см. приложение Б). Пример разметки представлен на рис. 6:

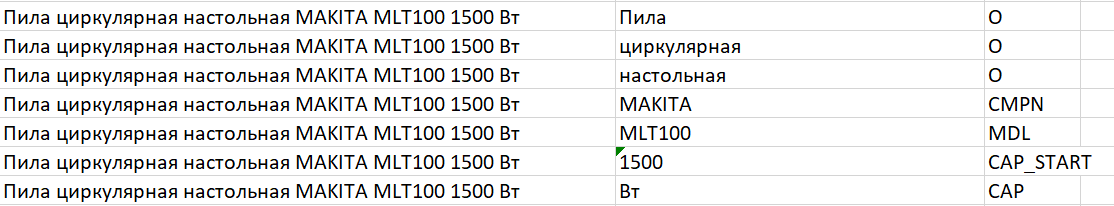


Рисунок 8. Пример разметки

## 2.2 Предобработка данных

Модели машинного обучения могут принимать на вход только числовые значения, поэтому все слова из названий и тэги нужно векторизовать. Это было сделано с помощью класса под названием VocabVectorize (см. приложение В). Во многих работах реализуются два отдельных класса Vocabulary и Vectorizer, но было принято решение их объединить для упрощения [9].

Перед началом дальнейшей работы датафрейм, импортированный из размеченного файла Эксель был сгруппирован по названиям. Теперь каждая строка представляет собой название и список с присвоенными токенам тэгами.

Далее для токенов из названий и для тэгов создаются словари, в которых содержаться все слова и категории по одному. В словаре с названиями также есть дополнительные <PAD>, с помощью которого названия дополняются до максимальной длины; и <UNK>, который используется в случае, если в словаре такого слова нет.

На следующем шаге с помощью методов text2seq и tag2seq каждому слову и тэгу присваиваются их индексы из соответствующих словарей. Методы seq2text и seq2tag выполняют обратное действие (см. приложение В).

В данном классе также есть функция tokenize, которая предназначена для того, чтобы разбивать новые названия (которые еще не видела модель) на токены. На этом же этапе происходит понижение регистра и лемматизация, т.е. приведение слова к начальной форме.

За ним следуют функции encode и decode, которые векторизуют и выполняют обратное действие, с помощью методов, описанных выше. Стоит отметить, что именно в этих функциях названия дополняются до максимальной длины с помощью пэддинга. Для приведения к привычному виду сначала удаляются все <PAD>, а затем применяются seq2text и seq2tag.

Как принято во всех работах по глубокому обучению в этом же классе находится функция \_\_len\_\_, которая подсчитывает длину словаря.

Затем идет реализация класса Dataset, который нужен, чтобы была возможность воспользоваться классом Dataloader из PyTorch. Функция \_\_len\_\_ возвращает длину текущего слова. А функция \_\_get\_item\_\_ нужна для того, чтобы возвращать слова (уже продленные пэддингами).

Далее объект класса Dataset делится на тренировочную и тестовую выборки по 80% и 20% соответственно. Далее тренировочная подразделяется еще и на валидирующую. Уже эти датасеты train, test, val подаются в класс Dataloader, работа с которым будет продолжена далее.

## 2.3 Использование предобученной модели BERT для решения задачи NER

Так как подготовленный датасет имеет совсем не большие размеры, было решено поступить следующим образом: сначала использовать предобученный BERT как первую модель, затем с помощью нее разметить еще 15000 товаров, а потом уже на новом наборе данных с использованием semi supervised learning настроить модели со слоями LSTM и GRU. Таким образом, в данной работе используются и передовые технологии в области NLP, так и классические, проверенные временем методы.

BERT (Bidirectional Encoder Representations from Transformers) – это языковая модель. Целью ее создания было улучшить поиск Google. Она представляет собой модель, которая использует технологию трансформеров для обработки текста и позволяет получать максимальную информацию из контекста на основе двунаправленной модели кодировщика. BERT является предварительно обученной моделью, которая использует большие объемы текстовых данных для обучения [10]. После этого модель может быть настроена на выполнение конкретных задач, таких как классификация текста или ответ на вопросы.

Так как архитектура и веса BERTа находятся в открытом доступе, можно скачать конкретную предобученную модель для решения необходимой задачи. В случае данной работы для решения задачи NER была использована модель под названием “BertForTokenClassification”. Архитектура данной модели показана на рис.7. При ее загрузке из библиотеки transformers предобученные веса для классической модели также загружаются. После чего вся модель (а не только несколько слоев) обучаются на исследуемом наборе данных. Так делается из-за того, что исходная модель предназначена для решения задач на английском языке, а в этой работе данные на русском.

Изображение выглядит как текст, человек, снимок экрана, документ

Автоматически созданное описание

Рисунок 9. Архитектура BERTForTokenClassification

Теперь можно перейти к реализации модели. Для начала создается класс BertModel в котором импортируется нужная модель с весами. Затем идет реализация цикла обучения (см. приложение В).

В качестве оптимизатора используется Adam. Как было описано в теоретической части работы, он нужен для обновления весов модели.

Так как данных немало, а архитектура модели не самая простая, цикл обучения длится достаточно долго. Поэтому для ускорения работы программы вычисления переносится на cuda, т.е. на графический процессор (см. приложение А).

Функция train\_loop проходит 30 эпох обучения, на каждой из которых происходит следующее:

* для тренировочного набора данных: модель выдает логиты, то есть для каждого из 33 токенов вероятность выбора каждого из 15 тэгов -> каждому токену присваивается тэг с большей вероятностью -> рассчитывается accuracy и loss
* для валидационного набора данных: происходит все то же самое, только дополнительно выводится f1-score. Этот этап нужен для того, чтобы во время обучения можно было наблюдать, как обучается модель, а не ждать до конца всего цикла.

Также для отображения того, сколько идет цикл, используется tqdm, который представляет собой progress bar.

На следующем этапе, для того чтобы показать, как модель справляется с новыми данными, которые в нее еще никогда не подавались, вызывается функция bert\_test. В ее начале выставляется параметр model.eval(), который отвечает за то, чтобы некоторые специфические слои (например, Dropout или BatchNorm), которые используются исключительно для процесса обучения, не выполнялись. Т.е. этот метод переводит модель в режим оценки.

На этом обучение первой модели заканчивается. Для дальнейшего semi supervised learning обучается несколько таких моделей на одном и том же, но перемешанном наборе данных. Их можно сохранить прямо в программе, чтобы модель и ее веса зафиксировались и их можно было использовать далее.

Важно отметить, что модель BERTForTokenClassification может работать, только если передавать в нее attention\_mask, так как в модели используется механизм attention. Она формируется сама, при условии использования специального токенайзера для BERTа. Но так как для разметки используется библиотека nltk реализацию attention\_mask нужно написать самостоятельно. Сделать это достаточно просто: для всех токенов в векторе добавляется 1, а для все пэддингов 0. Так мы определяем части предложений, на которые модели важно обратить внимание.

Создание этой маски происходит в классе VocabVectorize в функции encode, а в модель она передается в классе BertModel.

## 2.4 Разработка и настройка рекуррентной сети с использованием слоев LSTM и GRU для решения задачи NER.

Как и была написано выше, для дальнейшей работы было обучено 5 моделей BERT. Затем их результаты были импортированы и с их помощью была проведена разметка дополнительных 15000 случайно выбранных товаров. Это было сделано следующим образом: для каждого токена в каждом товаре для каждой из 5 моделей была выбрана категория с наибольшей вероятностью, затем для каждого токена сравнивались тэги, которые были присвоены в каждой из 5 моделей, в итоге для этого слова выбиралась категория с помощью функции моды, то есть самый популярный тэг. Благодаря такой операции была получена разметка, которая по точности не сильно уступает разметке вручную. Так как это достаточно время затратные операции – все вычисления снова были переведены на графический процессор.

Данные, собранные благодаря этому способу, были преобразованы в датафрейм, подобный тому, что подается в самом начале в модель BERT. Главное его отличие заключается в том, что добавился столбец ‘Handmade’. Он имеет тип данных int, и показывает, что если значение 1 – то название было размечено вручную, а при 0 – с помощью моделей BERT.

Так как количество названий увеличилось почти в 6 раз, объект класса VocabVectorize нужно создать заново, потому что увеличился не только сам словарь, но и максимальная длина одного названия товара.

Теперь можно перейти к процессу обучения. В качестве второй модели была выбрана архитектура со слоем BILSTM. Ее архитектура представлена на рис. 10. Стоит отметить, что для рекуррентных моделей нельзя забывать про параметр batch\_first, он отвечает за то, в каком порядке будут подаваться размерности данных. Если этого сделать, можно будет легко запутаться.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 10. Архитектура модели со слоем BILSTM

Цикл обучения принципиально не отличается от такой же функции для модели BERT (см. приложение В). Однако для соответствия задаче Semi Supervised Learning было внесено одно значимое изменение. Так как в данном случае модель должна меньше внимания на ошибки, связанные с данными, размеченными автоматически, был отредактирован подсчет loss. Теперь, ошибка, которая подсчитана на данных, размеченных автоматически будет иметь вес, умноженный на 0.2. То есть такие ошибки будут влиять на процесс обучения не полностью, а только на 20%. Данные, размеченные вручную, как и раньше будут иметь полный вес.

Для третьей модели, в которой используется слой BIGRU были выполнены аналогичные действия. Ее архитектура поменялась только в использовании другого слоя рекуррентной сети. Сделано это было для того, чтобы наблюдать отличия в точности схожих слоев, и сделать собственные выводы. Архитектура третьей модели представлена на рис. 11:

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 11. Архитектура модели с использованием слоя BIGRU

## 2.5 Результаты

Для определения точности в задаче NER будут использоваться метрики accuracy и f1-score. То, как они рассчитываются, представлено на формулах 3.1 и 3.2.

где TP – правильно предсказанный положительный результат, TN – правильно предсказанный негативный результат, FP – неправильно предсказанный положительный результат, FN – неправильно предсказанный негативный результат.

F мера чаще всего используется для тех данных, у которых классы не сбалансированы, что подходит к исследуемым данным. Гистограммы категорий представлены на рис. 12 и рис. 13 до дополнительной разметки и после соответственно.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 12. Гистограмма тэгов на данных, размеченных вручную

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 13. Гистограмма тэгов на дополнительно размеченных данных

Теперь можно перейти к визуализации результатов, полученных при использовании каждой из трех модели. В начале будут представлены графики изменения ошибки, точности и f1 – score по эпохам на тренировочном и валидационном наборе данных.

Таблица 1. Изменение метрик для модели BERT

|  |  |  |
| --- | --- | --- |
|  | Train | Val |
| Loss |  |  |
| Acc |  |  |
| F1 |  |  |

Аналогичные графики были построены для моделей BI-LSTM и BI-GRU:

Таблица 2. Изменение метрик для моделей BI-LSTM и BI-GRU

|  |  |  |
| --- | --- | --- |
|  | BI-LSTM | BI-GRU |
| Loss |  |  |
| Acc |  |  |
| F1 |  |  |

Но самыми показательными эти метрики являются, когда они подсчитаны на тестовом датасете. Потому что эти данные модель никогда не видела, значит, так мы поймем, как она будет справляться с новыми данными при дальнейшем использовании.

Таблица 3. Значения метрик на тестовых данных

|  |  |  |  |
| --- | --- | --- | --- |
|  | BERT | BI-LSTM | BI-GRU |
| Accuracy | 94,9% | 79,2% | 78,4% |
| F1-score | 65,4% | 76,8% | 76,9% |

Также, для визуализации решения задачи NER была реализована функция visualize\_ner. В нее подается одно название товара, и на выходе получаем предложение, каждый из токенов которого окрашен в цвет соответствующего тэга. Пример такой разметки представлен на рис. 14:

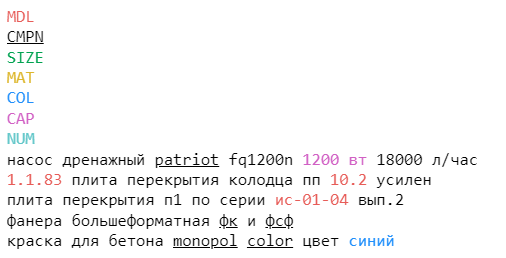


Рисунок 14. Пример визуализации решения задачи NER

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения данной работы была решена задача NER. Проблему с разным форматом изначальных данных удалось решить благодаря ручной разметке. Но набор данных был недостаточно большим, для его увеличения была использована предобученная модель, а затем с ее помощью размечен больший датасет. На выходе получена готовая реализация задачи распознавания именованных сущностей. Модели обучены на названиях товаров с торговой площадки, поэтому ее решение можно было бы использовать и на самом сайте supl biz, и в других интернет-магазинах, которые специализируется на строительных товарах. Более того, для полноты раскрытия темы была приведена теоретическая справка о том, что такое нейронные сети, функции активации, оптимизаторы, какие они бывают и как работают.

Как оказалось, большую точность имеет модель BI-GRU. Это было предсказуемо, так как она является развитой версией модели BI-LSTM.

# **СПИСОК ИСТОЧНИКОВ**

1. Bird S. Natural Language Processing with Python / Bird S., Klein E., Loper E. – California: O’REILLY, 2009. – 504 p.
2. Гольдберг Й. Нейросетевые методы в обработке естественного языка / пер. с анг. А. А. Слинкина. – Москва: ДМК Пресс, 2019. – 282 с.
3. Википедия. Функция активации. – Википедия, свободная энциклопедия. – URL: ru.wikipedia.org/wiki/Функция\_активации (дата обращения: 30.04.2023). – Текст: электронный.
4. Lavanya Gupta. Focal Loss – What, Why, and How? / Lavanya Gupta – 2021 – URL: <https://medium.com/swlh/focal-loss-what-why-and-how-df6735f26616> (дата обращения: 21.04.2023). – Текст: электронный.
5. Пойнтнер Ян. Программируем с PyTorch: Создание приложений глубокого обучения / Ян Пойнтнер. – СПб.: Питер, 2020. – 256 с.
6. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение / Я. Гудфеллоуб И. Бенджио, А. Курвилль. – 2-е изд., испр. – М.: ДМК Пресс, 2018. – 652 с.
7. Olah C. Understanding LSTM networks / Christopher Olah – 2015 – URL: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> (дата обращения: 30.04.2023). – Текст: электронный.
8. Zhiheng Huang. Bidirectional LSTM-CRF Models for Sequence Tagging / Zhiheng Huang, Wei Xu, Kai Yu. – arxiv.org, 2015. – 10 p. – URL: <https://arxiv.org/abs/1508.01991> (дата обращения: 02.05.2023). – Текст: электронный.
9. Макмахан Брайан. Знакомство с PyTorch: глубокое обучение при обработке естественного языка / Макмахан Брайан, Рао Делип. - СПб.: Питер, 2020. — 256 с
10. Devlin J. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova – arxiv.org, 2018. – 16 p. – URL: <https://arxiv.org/abs/1810.04805> (дата обращения: 02.05.2023). – Текст: электронный.
11. Supl.biz. URL: <https://supl.biz/> (дата обращения 30.03.2023). – Текст: электронный.

# **ПРИЛОЖЕНИЕ А. ОПИСАНИЕ ПРОЦЕССОРА**

Процессор - AMD Ryzen 7 4800HS with Radeon

Ядра - 16

Оперативная память - 16 Гб DDR4

Видеокарта - NVIDIA GeForce GTX 1660 Ti with Max-Q Design

Видеопамять - 6 Гб GDDR6

Тактовая частота - 2.90 GHz

Частота системной шины - 100 MHz

Объём кэша второго уровня - 512 Кб

# **ПРИЛОЖЕНИЕ Б. ГЕНЕРАЦИЯ ДАТАСЕТОВ**

import json

import nltk

import pymorphy2

import pandas as pd

from pymorphy2.shapes import is\_punctuation

df = pd.read\_json('supl\_biz\_157k.json')

df = df.iloc[200:] #первые 200 слов обрезаются, так как в начале ручной разметки они были перенесены в файл вручную

df = df.sample(frac=1, random\_state=1)

df.to\_json('shuffled.json')

df = pd.read\_json('shuffled.json')

df = df[200:]

df.loc[29620:].head(15000).to\_excel('for\_bert\_excel.xlsx', index=False)

new\_df = df.copy()

new\_df['Words'] = new\_df['Title'].apply(lambda x: list(filter(lambda y: not is\_punctuation(y), nltk.word\_tokenize(x))))

new\_df = new\_df[['Title', 'Words']].explode('Words') #каждое слово из названия на новой строке

new\_df[9997:19998].to\_excel('new\_shuffled\_excel.xlsx', index=False)

# **ПРИЛОЖЕНИЕ В. ОСНОВНАЯ ПРОГРАММА**

import nltk

import torch

import pymorphy2

import pandas as pd

import torch.nn as nn

from tqdm import tqdm

import matplotlib.pyplot as plt

from collections import defaultdict

from sklearn.metrics import f1\_score

from sklearn.metrics import accuracy\_score

from pymorphy2.shapes import is\_punctuation

from transformers import BertForTokenClassification

from torch.utils.data import Dataset, DataLoader, random\_split

df = pd.read\_excel('shuffled\_excel\_LAST.xlsx').drop\_duplicates()

df.columns = ["Title", "Words", "Tag"]

df['Tag'].value\_counts().plot.bar(figsize=(12, 4))

plt.title('Гистограмма тэгов')

plt.xlabel('Тэги')

plt.show()

first\_dataframe = df.groupby('Title')['Tag'].apply(list).reset\_index()

words = df.groupby('Title')['Words'].apply(list).reset\_index()['Words']

first\_dataframe['Words'] = words

class VocabVectorize:

pad = "<PAD>"

unk = "<UNK>"

titles = []

def \_\_init\_\_(self, titles, tags):

titles = titles.apply(lambda x: self.tokenize(x)).explode().value\_counts().index

max\_len = max([len(val) for val in titles])

self.alphabet = [VocabVectorize.pad, VocabVectorize.unk, \*titles]

self.max\_len = max\_len

text2seq = {word: i for i, word in enumerate(self.alphabet)}

self.seq2text = {i: word for i, word in enumerate(self.alphabet)}

unknown\_idx = text2seq[VocabVectorize.unk]

self.text2seq = defaultdict(lambda: unknown\_idx, text2seq)

tags = tags.explode().value\_counts().index

self.alphabet\_tags = [\*tags]

self.tag2seq = {word: i for i, word in enumerate(self.alphabet\_tags)}

self.seq2tag = {i: word for i, word in enumerate(self.alphabet\_tags)

def \_\_len\_\_(self):

return len(self.alphabet)

def tokenize(self, text):

return list(filter(lambda y: not is\_punctuation(y) and y != '``', nltk.word\_tokenize(text.lower())))

def encode(self, text):

no\_pad = list(map(lambda x: self.text2seq[x], self.tokenize(text)))

len\_pad = self.max\_len - len(no\_pad)

attention\_mask = torch.tensor([1] \* len(no\_pad) + [0] \* len\_pad).long()

return torch.tensor(no\_pad + [self.text2seq['<PAD>']]\*len\_pad), attention\_mask

def decode(self, encode\_text):

with\_pad = list(map(self.seq2text.get, encode\_text.tolist()))

return ' '.join(list(filter(lambda x: x != '<PAD>', with\_pad)))

#преобразует тэг в вектор

def tag\_encode(self, tags):

encoded\_tag = torch.tensor(list(map(self.tag2seq.get, tags + ['O']\*(self.max\_len - len(tags)))))

return encoded\_tag

def tag\_decode(self, encode\_tag, encode\_text):

decoded\_tag = list(map(self.seq2tag.get, encode\_tag[encode\_text != 0].tolist()))

return decoded\_tag

class DataSet(Dataset):

def \_\_init\_\_(self, data, vocab):

self.data = data

self.vocab = vocab

self.max\_len = self.vocab.max\_len

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, i):

x, attention\_mask = self.vocab.encode(self.data.iloc[i]['Title'])

y = self.vocab.tag\_encode(self.data.iloc[i]['Tag'])

return x, y, attention\_mask

dataset = DataSet(first\_dataframe, vocab)

ratio = 0.2

num\_test = int(ratio \* len(dataset))

num\_train = len(dataset) - num\_test

train\_dataset, test\_dataset = random\_split(dataset, lengths=[num\_train, num\_test])

num\_valid = int(ratio \* len(train\_dataset))

num\_train = len(train\_dataset) - num\_valid

train\_dataset, valid\_dataset = random\_split(train\_dataset, lengths=[num\_train, num\_valid])

batch\_size = 8

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True, drop\_last=True)

valid\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size, drop\_last=True)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, drop\_last=True)

use\_cuda = torch.cuda.is\_available()

device = torch.device("cuda:0" if use\_cuda else "cpu")

class BertModel(torch.nn.Module):

def \_\_init\_\_(self, unique\_labels):

super(BertModel, self).\_\_init\_\_()

self.bert = BertForTokenClassification.from\_pretrained('bert-base-cased', num\_labels=unique\_labels)

def forward(self, input\_id, mask, label):

output = self.bert(input\_ids=input\_id, attention\_mask=mask, labels=label, return\_dict=False)

return output

def train\_loop(model, train\_dataloader, val\_dataloader, batch\_size):

optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING\_RATE)

if use\_cuda:

model = model.cuda()

train\_losses = []

train\_acc = []

train\_f1 = []

val\_losses = []

val\_acc = []

val\_f1 = []

for epoch\_num in range(EPOCHS):

total\_acc\_train = 0

total\_loss\_train = 0

model.train()

true = []

pred = []

for train\_data, train\_label, attention\_mask in tqdm(train\_dataloader):

train\_label = train\_label.to(device)

mask = attention\_mask.to(device)

input\_id = train\_data.to(device)

optimizer.zero\_grad()

loss, logits = model(input\_id, mask, train\_label)

for i in range(logits.shape[0]):

predictions = logits[i].argmax(dim=1)

pred.append(predictions)

true.append(train\_label[i])

acc = (predictions == train\_label[i]).float().mean()

total\_acc\_train += acc

total\_loss\_train += loss.item()

loss.backward()

optimizer.step()

train\_losses.append(total\_loss\_train / (len(train\_dataloader) \* batch\_size))

train\_acc.append(total\_acc\_train.cpu() / (len(train\_dataloader) \* batch\_size))

f1 = f1\_score(torch.concat(true).cpu(), torch.concat(pred).cpu(), average='macro')

train\_f1.append(f1)

model.eval()

total\_acc\_val = 0

total\_loss\_val = 0

true = []

pred = []

for val\_data, val\_label, attention\_mask in val\_dataloader:

val\_label = val\_label.to(device)

mask = attention\_mask.to(device)

input\_id = val\_data.to(device)

loss, logits = model(input\_id, mask, val\_label)

for i in range(logits.shape[0]):

predictions = logits[i].argmax(dim=1)

pred.append(predictions)

true.append(val\_label[i])

acc = (predictions == val\_label[i]).float().mean()

total\_acc\_val += acc

total\_loss\_val += loss.item()

val\_accuracy = total\_acc\_val.cpu() / (len(val\_dataloader) \* batch\_size)

val\_loss = total\_loss\_val / (len(val\_dataloader) \* batch\_size)

f1 = f1\_score(torch.concat(true).cpu(), torch.concat(pred).cpu(), average='macro')

val\_losses.append(val\_loss)

val\_acc.append(val\_accuracy)

val\_f1.append(f1)

print(f'Epochs: {epoch\_num + 1} | Loss: {total\_loss\_train / (len(train\_dataloader) \* batch\_size): .3f} | Accuracy: {total\_acc\_train / (len(train\_dataloader) \* batch\_size): .3f} | F1 Score {train\_f1[-1]: .3f} | Val\_Loss: {val\_loss: .3f} | Val Accuracy: {val\_accuracy: .3f} | Val F1 Score: {f1: .3f}')

return train\_losses, train\_acc, train\_f1, val\_losses, val\_acc, val\_f1

LEARNING\_RATE = 0.00005

EPOCHS = 15

unique\_labels = len(vocab.tag2seq.keys())

def train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name):

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True, drop\_last=True)

valid\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size, drop\_last=True)

model = BertModel(unique\_labels)

train\_losses, train\_acc, train\_f1, val\_losses, val\_acc, val\_f1 = train\_loop(model, train\_loader, valid\_loader, batch\_size)

torch.save(model, file\_name)

torch.cuda.empty\_cache()

del train\_loader

del valid\_loader

del model

return train\_losses, train\_acc, train\_f1, val\_losses, val\_acc, val\_f1

train\_losses\_1, train\_acc\_1, train\_f1\_1, val\_losses\_1, val\_acc\_1, val\_f1\_1 = train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name='model\_1.pth')

train\_losses\_2, train\_acc\_2, train\_f1\_2, val\_losses\_2, val\_acc\_2, val\_f1\_2 = train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name='model\_2.pth')

train\_losses\_3, train\_acc\_3, train\_f1\_3, val\_losses\_3, val\_acc\_3, val\_f1\_3 = train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name='model\_3.pth')

train\_losses\_4, train\_acc\_4, train\_f1\_4, val\_losses\_4, val\_acc\_4, val\_f1\_4 = train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name='model\_4.pth')

train\_losses\_5, train\_acc\_5, train\_f1\_5, val\_losses\_5, val\_acc\_5, val\_f1\_5 = train\_Bert(train\_dataset, valid\_dataset, batch\_size, unique\_labels, file\_name='model\_5.pth')

def bert\_test(model, test\_dataloader, batch\_size):

model.eval()

total\_acc\_test = 0

total\_loss\_test = 0

true = []

pred = []

for test\_data, test\_label, attention\_mask in test\_dataloader:

test\_label = test\_label.to(device)

mask = attention\_mask.to(device)

input\_id = test\_data.to(device)

loss, logits = model(input\_id, mask, test\_label)

for i in range(logits.shape[0]):

predictions = logits[i].argmax(dim=1)

true.append(predictions)

pred.append(test\_label[i])

acc = (predictions == test\_label[i]).float().mean()

total\_acc\_test += acc

total\_loss\_test += loss.item()

test\_accuracy = total\_acc\_test / (len(test\_dataloader) \* batch\_size)

test\_loss = total\_loss\_test / (len(test\_dataloader) \* batch\_size)

f1 = f1\_score(torch.concat(true).cpu(), torch.concat(pred).cpu(), average='macro')

print(f'TEST | Loss: {test\_loss: .3f} | Accuracy: {test\_accuracy: .3f} | F1 Score: {f1: .3f}')

model\_1 = torch.load('model\_1.pth').to(device)

bert\_test(model\_1, test\_loader, batch\_size)

big = pd.read\_excel('for\_bert\_excel.xlsx')

titles = []

tags\_list = []

model\_1 = torch.load('model\_1.pth').to(device)

model\_2 = torch.load('model\_2.pth').to(device)

model\_3 = torch.load('model\_3.pth').to(device)

model\_4 = torch.load('model\_4.pth').to(device)

model\_5 = torch.load('model\_5.pth').to(device)

with torch.no\_grad():

for title in tqdm(big['Title']):

token, att\_mask = vocab.encode(title)

preds = []

for model in [model\_1, model\_2, model\_3, model\_4, model\_5]:

loss, pred = model(token.unsqueeze(0).to(device), att\_mask.unsqueeze(0).to(device), torch.zeros(1, len(token)).long().to(device))

preds.append(pred.argmax(dim=2)[0])

preds = torch.concat(preds).reshape(5, -1)

preds = torch.mode(preds, dim=0).values

tag = vocab.tag\_decode(preds, token)

titles.append(title)

tags\_list.append(tag)

torch.cuda.empty\_cache()

print(titles[0], tags\_list[0])

def visualize\_ner(words, tags):

colors = {

'MDL': '\033[31m',

'CMPN': '\033[4m',

'SIZE': '\033[32m',

'MAT': '\033[33m',

'COL': '\033[34m',

'CAP': '\033[35m',

'NUM': '\033[36m'

}

blank = '\033[0m'

res = ''.join([v + k + blank + '\n' for k, v in colors.items()])

for i in range(len(words)):

for token, label in zip(words[i], tags[i]):

tag = label.split('\_')[0]

res += colors.get(tag, blank) + token + blank + ' '

res += '\n'

print(res)

custom\_title = ['Насос дренажный PATRIOT FQ1200N 1200 Вт, 18000 л/час',

'1.1.83 Плита перекрытия колодца ПП 10.2 (усилен.)',

'Плита перекрытия П1 по серии ИС-01-04, вып.2',

'фанера большеформатная ФК и ФСФ',

'Краска для бетона Monopol Color цвет синий'

]

tokens = list(map(lambda x: vocab.tokenize(x), custom\_title))

X = torch.concat(list(map(lambda x: vocab.encode(x)[0], custom\_title))).reshape(len(custom\_title), -1)

mask = torch.concat(list(map(lambda x: vocab.encode(x)[1], custom\_title))).reshape(len(custom\_title), -1)

y = torch.zeros(len(custom\_title), len(X[0])).long()

\_, pred = model\_1(X.to(device), mask.to(device), y.to(device))

pred = pred.argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

\_, pred = model\_2(X.to(device), mask.to(device), y.to(device))

pred = pred.argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

\_, pred = model\_3(X.to(device), mask.to(device), y.to(device))

pred = pred.argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

\_, pred = model\_4(X.to(device), mask.to(device), y.to(device))

pred = pred.argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

\_, pred = model\_5(X.to(device), mask.to(device), y.to(device))

pred = pred.argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

new\_dataset = pd.DataFrame({'Title':titles, 'Tag':tags\_list})

new\_dataset['Words'] = new\_dataset['Title'].apply(lambda x: vocab.tokenize(x))

new\_dataset

new\_dataset.to\_excel('bert\_made.xlsx', index=False)

new\_dataset = pd.read\_excel('bert\_made.xlsx')

new\_dataset['Tag'] = new\_dataset['Tag'].apply(eval)

new\_dataset['Words'] = new\_dataset['Words'].apply(eval)

visualize\_ner(new\_dataset['Words'][14995:].values, new\_dataset['Tag'][14995:].values)

new\_dataset['HandMade'] = 0

first\_dataframe['HandMade'] = 1

big\_dataframe = pd.concat([first\_dataframe, new\_dataset], ignore\_index=True)

big\_dataframe.shape

big\_dataframe['Tag'].explode().value\_counts().plot.bar(figsize=(12, 4))

plt.title('Гистограмма тэгов нового датасета')

plt.xlabel('Тэги')

plt.show()

vocab = VocabVectorize(big\_dataframe['Title'], big\_dataframe['Tag'])

vocab.max\_len

class DataSet\_hm(Dataset):

def \_\_init\_\_(self, data, vocab):

self.data = data

self.vocab = vocab

self.max\_len = self.vocab.max\_len

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, i):

hm = self.data.iloc[i]['HandMade']

x, attention\_mask = self.vocab.encode(self.data.iloc[i]['Title'])

y = torch.tensor(list(map(self.vocab.tag2seq.get, self.data.iloc[i]['Tag'] + ['O']\*(self.max\_len - len(self.data.iloc[i]['Tag'])))))

return x, y, attention\_mask, hm

big\_dataset = DataSet\_hm(big\_dataframe, vocab)

class NER\_LSTM\_Model(nn.Module):

def \_\_init\_\_(self, emb\_size, input\_size, hidden\_size, num\_layers, n\_classes, input\_len):

super().\_\_init\_\_()

self.embed = nn.Embedding(input\_size + 1, emb\_size)

self.bilstm = nn.LSTM(emb\_size, hidden\_size=hidden\_size, num\_layers=num\_layers, bidirectional=True, batch\_first=True)

self.dropout = nn.Dropout(p=0.2)

self.norm = nn.BatchNorm1d(input\_len)

self.fc1 = nn.Linear(2\*hidden\_size, hidden\_size)

self.sigmoid = nn.Tanh()

self.fc2 = nn.Linear(hidden\_size, n\_classes)

def forward(self, x):

out = self.embed(x)

out = self.dropout(out)

out, \_ = self.bilstm(out)

out = self.norm(out)

out = self.fc1(out)

out = self.sigmoid(out)

out = self.fc2(out)

return out

class FocalLoss(nn.Module):

def \_\_init\_\_(self, alpha=1, gamma=2):

super(FocalLoss, self).\_\_init\_\_()

self.alpha = alpha

self.gamma = gamma

def forward(self, inputs, targets):

ce\_loss = nn.CrossEntropyLoss(reduction='none')(inputs, targets)

pt = torch.exp(-ce\_loss)

focal\_loss = self.alpha \* (1 - pt) \*\* self.gamma \* ce\_loss

return focal\_loss.mean()

def semi\_train(model, loader, loss\_function, optimizer):

model.train()

epoch\_loss = 0

epoch\_predict = []

epoch\_label = []

N = 0

beta = 0.2

for inputs, labels, \_, hm in tqdm(loader):

optimizer.zero\_grad()

outputs = model(inputs.to(device)).permute(0, 2, 1)

loss = 0

if hm.sum():

loss += loss\_function(outputs[hm.bool()], labels[hm.bool()].to(device))

if ~hm.sum():

loss += beta \* loss\_function(outputs[~hm.bool()], labels[~hm.bool()].to(device))

loss.backward()

optimizer.step()

epoch\_loss += loss.item()

mask = labels != vocab.text2seq.get(VocabVectorize.pad)

epoch\_predict.append(outputs.argmax(dim=1)[mask])

epoch\_label.append(labels[mask])

return epoch\_loss / len(loader), accuracy\_score(torch.concat(epoch\_label).cpu(), torch.concat(epoch\_predict).cpu()), f1\_score(torch.concat(epoch\_label).cpu(), torch.concat(epoch\_predict).cpu(), average='macro')

def semi\_test(model, loader, loss\_function):

with torch.no\_grad():

model.eval()

epoch\_predict = []

epoch\_label = []

for inputs, labels, \_, \_ in tqdm(loader):

outputs = model(inputs.to(device)).permute(0, 2, 1)

mask = labels != vocab.text2seq.get(VocabVectorize.pad)

epoch\_predict.append(outputs.argmax(dim=1)[mask])

epoch\_label.append(labels[mask])

return accuracy\_score(torch.concat(epoch\_label).cpu(), torch.concat(epoch\_predict).cpu()), f1\_score(torch.concat(epoch\_label).cpu(), torch.concat(epoch\_predict).cpu(), average='macro')

input\_size = len(vocab.alphabet)

emb\_size = 200

hidden\_size = 100

num\_layers = 3

input\_len = vocab.max\_len

n\_classes = len(vocab.alphabet\_tags)

model = NER\_LSTM\_Model(emb\_size, input\_size, hidden\_size, num\_layers, n\_classes, input\_len).to(device)

criterion = FocalLoss(alpha=1, gamma=2)

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)

ratio = 0.2

num\_test = int(ratio \* len(big\_dataset))

num\_train = len(big\_dataset) - num\_test

train\_dataset, test\_dataset = random\_split(big\_dataset, lengths=[num\_train, num\_test])

num\_valid = int(ratio \* len(train\_dataset))

num\_train = len(train\_dataset) - num\_valid

train\_dataset, valid\_dataset = random\_split(train\_dataset, lengths=[num\_train, num\_valid])

batch\_size = 8

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True, drop\_last=True)

valid\_loader = DataLoader(valid\_dataset, batch\_size=batch\_size, drop\_last=True)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, drop\_last=True)

num\_epochs = 10

train\_losses = []

train\_accs = []

train\_f1s = []

valid\_accs = []

valid\_f1s = []

for epoch in range(num\_epochs):

train\_loss, train\_acc, train\_f1 = semi\_train(model, loader=train\_loader, loss\_function=criterion, optimizer=optimizer)

valid\_acc, valid\_f1 = semi\_test(model, loader=valid\_loader, loss\_function=criterion)

print(f'Epoch {epoch}')

print(f'Train Loss {train\_loss:.6f}, Train Acc {train\_acc:.6f}, Train F1 {train\_f1:.6f}')

print(f'Valid Acc {valid\_acc:.6f}, Valid F1 {valid\_f1:.6f}')

print('---')

train\_losses.append(train\_loss)

train\_accs.append(train\_acc)

train\_f1s.append(train\_f1)

valid\_accs.append(valid\_acc)

valid\_f1s.append(valid\_f1)

torch.cuda.empty\_cache()

test\_acc, test\_f1 = semi\_test(model=model, loader=test\_loader, loss\_function=criterion)

print(test\_acc, test\_f1)

torch.save(model, 'final\_lstm\_model.pth')

tokens = list(map(lambda x: vocab.tokenize(x), custom\_title))

X = torch.concat(list(map(lambda x: vocab.encode(x)[0], custom\_title))).reshape(len(custom\_title), -1)

pred = model(X.to(device)).argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)

class NER\_GRU\_Model(nn.Module):

def \_\_init\_\_(self, emb\_size, input\_size, hidden\_size, num\_layers, n\_classes, input\_len):

super().\_\_init\_\_()

self.embed = nn.Embedding(input\_size + 1, emb\_size)

self.bilstm = nn.GRU(emb\_size, hidden\_size=hidden\_size, num\_layers=num\_layers, bidirectional=True, batch\_first=True)

self.dropout = nn.Dropout(p=0.2)

self.norm = nn.BatchNorm1d(input\_len)

self.fc1 = nn.Linear(2\*hidden\_size, hidden\_size)

self.sigmoid = nn.Tanh()

self.fc2 = nn.Linear(hidden\_size, n\_classes)

def forward(self, x):

out = self.embed(x)

out = self.dropout(out)

out, \_ = self.bilstm(out)

out = self.norm(out)

out = self.fc1(out)

out = self.sigmoid(out)

out = self.fc2(out)

return out

input\_size = len(vocab.alphabet)

emb\_size = 200

hidden\_size = 100

num\_layers = 3

input\_len = vocab.max\_len

n\_classes = len(vocab.alphabet\_tags)

model = NER\_GRU\_Model(emb\_size, input\_size, hidden\_size, num\_layers, n\_classes, input\_len).to(device)

criterion = FocalLoss(alpha=1, gamma=2)

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)

num\_epochs = 10

train\_losses = []

train\_accs = []

train\_f1s = []

valid\_accs = []

valid\_f1s = []

for epoch in range(num\_epochs):

train\_loss, train\_acc, train\_f1 = semi\_train(model, loader=train\_loader, loss\_function=criterion, optimizer=optimizer)

valid\_acc, valid\_f1 = semi\_test(model, loader=valid\_loader, loss\_function=criterion)

print(f'Epoch {epoch}')

print(f'Train Loss {train\_loss:.6f}, Train Acc {train\_acc:.6f}, Train F1 {train\_f1:.6f}')

print(f'Valid Acc {valid\_acc:.6f}, Valid F1 {valid\_f1:.6f}')

print('---')

train\_losses.append(train\_loss)

train\_accs.append(train\_acc)

train\_f1s.append(train\_f1)

valid\_accs.append(valid\_acc)

valid\_f1s.append(valid\_f1)

torch.cuda.empty\_cache()

test\_acc, test\_f1 = semi\_test(model=model, loader=test\_loader, loss\_function=criterion)

print(test\_acc, test\_f1)

torch.save(model, 'final\_gru\_model.pth')

tokens = list(map(lambda x: vocab.tokenize(x), custom\_title))

X = torch.concat(list(map(lambda x: vocab.encode(x)[0], custom\_title))).reshape(len(custom\_title), -1)

pred = model(X.to(device)).argmax(dim=2)

tags = [vocab.tag\_decode(pred[i], X[i]) for i in range(len(custom\_title))]

visualize\_ner(tokens, tags)