СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С помощью естественного языка можно выразить любую мысль, любую идею. Любое изображение или звук можно описать словами. Текст является всеобъемлющим средством передачи информации. Что означает, что обработка текстов на естественных языках (Natural Language Processing, NLP) является крайне важной и актуальной проблемой.

Существует большое множество задач в области обработки текстов, например:

- перевод с одного языка на другой (например, перевод с русского на японский или обратно);
- или же монолингвистический перевод (перевод с языка в него же), как, например, упрощение текстов (понижение сложности слов, выражений, грамматики, сохраняя при этом исходный смысл текста);
- классификация текстов (положительный или отрицательный отзыв, фильтрация спама и т. д.);
- генерация текстов (например, из заданного заголовка сгенерировать статью);
- реферирование текстов (из большого по объёму документа или набора документов выделить ёмкую основную мысль);

Для японского и китайского языков особый интерес представляет задача упрощения текстов, так как эти языки используют иероглифическую письменность, где для чтения текстов нужно знать чтение и значение отдельных иероглифов (в японском языке большинство иероглифов имеют несколько чтений, порой даже больше 10). Это может значительно сузить круг возможных читателей какой-либо текста — дети изучают иероглифы, начиная с первого класса школы и до самого выпуска. То же касается и иностранцев, имеющих довольно ограниченное знание иероглифов. Причём даже взрослые японцы и китайцы могут испытывать трудности с иероглифами, особенно связанные с юридическими документами. Количество иероглифов довольно высоко, в среднем, взрослый японец знает порядка 2 000 иероглифов, взрослый китаец — 8 000 (хотя самих иероглифов значительно больше — не менее 80 000, — но большинство из них используются крайне редко). Поэтому есть высокая потребность в упрощении текстов для увеличения количества их потенциальных читателей.

Более того, упрощение текстов может повысить эффективность других задач NLP, как, например, реферирование, извлечение информации, машинный перевод и т. д.

Целью данной научно-исследовательской работы является исследование и обзор предметной области и теоретической части предстоящей магистерской работы.

Для достижения данной цели необходимо выполнить следующие задачи:

- раскрыть предметную область — нюансы работы с японским языком, существующие решения в упрощении и сервисы, в них нуждающиеся;
- описать компьютерную обработку естественных языков с помощью искусственных нейронных сетей, в частности, с помощью модели Transformer;
- выбрать и описать используемый инструментарий для практической реализации.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ РАБОТЫ

1.1. Этапы обработки естественного языка

Как правило, в обработке текстов обычно выделяют следующие этапы [9, с. 9]:

1. Графематический анализ. Здесь осуществляется анализ на уровне символов, в том числе и токенизация, то есть разбиение набора символов (текста) на последовательность отдельных структурированных частей (слово, знак препинания, число, гиперссылка, адрес электронной почты и т. д.).
2. Морфологический анализ. Здесь происходит анализ на уровне слов (не токенов). Стоит выделить следующие процессы, проходящие на этом этапе:
   - Лемматизация — это нахождение нормальной (начальной) формы слова (леммы), к примеру, лемма у слова «сбегать» — «бегать». В японском это относится в основном к глаголам, так как у существительных, как правило, всего одна словоформа.
   - Приписывание граммем. Граммемы — грамматическая характеристика, например, род, падеж, число. Граммемы могут помочь в разрешении неоднозначностей, которые возникают в морфологии.
3. Фрагментационный анализ. Осуществляется на уровне фраз, частей предложений. Этот этап неразрывно связан с синтаксическим анализом, а иногда и вовсе говорят о них, как об одном целом. Сюда, например, может входить обработка причастных или деепричастных оборотов.
4. Синтаксический анализ. Осуществляется на уровне предложений. Здесь, как правило, строится дерево зависимостей одних слов от других, а также исключается морфологическая неоднозначность.
5. Семантический анализ. Осуществляется на уровне всего текста. Самый сложный и неоднозначный из этапов, здесь появляется формальное представление смысла текста, как правило, в виде сементического графа. На сегодняшний день задачи семантического анализа чаще всего решаются искусственными нейронными сетями, о чём мы и поговорим далее.

1.2. Искусственные нейронные сети

Вкратце, искусственная нейронная сеть (ИНС) представляет собой систему соединённых и взаимодействующих между собой простых нейронов. Каждый нейрон получает на вход несколько чисел (входные данные или же выходы других нейронов), суммирует эти числа с определёнными коэффициентами (нахождение оптимальных коэффициентов — обучение ИНС), после чего применяет к сумме функцию активации (любую нелинейную функцию) и передаёт результат на вход другому нейрону (или же на выход ИНС).

Существует большое множество различных архитектур ИНС (свёрточные, рекуррентные, рекурсивные, графовые и т. д.), однако в данной работе будет сконцентрировано внимание на так называемых Transformer'ах, используемых, как правило, для языковой обработки, в частности для задач перевода и упрощения текстов.

1.3. Какие модели используют для обработки естественных языков

Существует большое количество моделей, использующихся для обработки естественных языков. Одними из наиболее популярных (до появления Transformer'ов) были следующие:

- Рекуррентные нейронные сети (RNN) (1982 г.):
  - Обработка последовательностей (например, текста).
  - На каждый слой передаётся текущий элемент (слово) + результат предыдущего слоя.
  - Причём есть обратные связи — поэтому рекуррентные.
  - Очень медленные — из-за последовательной природы нельзя распараллелить.
- Долгая краткосрочная память (LTSM) (1997 г.):
  - Разновиднсть RNN с элементом «забывания».
  - Ещё медленнее (из-за более сложного устройства каждого слоя модели).

Для больших объёмов данных эти модели подходят не очень хорошо из-за своей последовательной природы обработки (требует огромных вычислительных ресурсов). На замену им пришли так называемые Transformer'ы (которые мы обсудим в следующем разделе).

1.4. О модели Transformer

Transformer — относительно новая (2017 г.) архитектура глубоких ИНС, разработанная в Google Brain. Так же, как и рекуррентные ИНС (РНС), Transformer'ы предназначены для обработки последовательностей (к примеру, текста), то есть Transformer'ы относятся к sequence-to-sequence (seq2seq) моделям. В отличие от РНС, Transformer'ы не требуют обработки последовательностей по порядку, благодаря чему они распараллеливаются легче, чем РНС, и могут быстрее значительно обучаться.

1.4.1. Где применяются Transformer'ы

Используются Transformer'ы, например, в Яндексе (там его используют для лучшего ранжирования запросов, то есть поиск идёт не только по тексту, как обычной строке, но и по смыслу этого текста), во многих переводчиках (Яндекс, Google, DeepL и т. д.), а также в GPT-3 — самой большой на сегодняшний день модели генерации текстов на английском языке.

1.4.2. Устройство Transformer'а

Transformer состоит из encoder'а и decoder'а. Encoder получает на вход последовательность слов в виде векторов (word2vec). Decoder получает на вход часть этой последовательности и выход encoder'а. Encoder и decoder состоят из слоев. Слои encoder'а последовательно передают результат следующему слою в качестве его входа. Слои decoder'а последовательно передают результат следующему слою вместе с результатом encoder'а в качестве его входа.

Каждый encoder состоит из механизма внимания (attention) (вход из предыдущего слоя) и ИНС с прямой связью (вход из механизма внимания). Каждый decoder состоит из механизма внимания (вход из предыдущего слоя), механизма внимания к результатам encoder'а и ИНС с прямой связью (вход из механизма внимания).

1.4.3. Механизм внимания

Каждый механизм внимания параметризован матрицами весов запросов W_Q, весов ключей W_K, весов значений W_V. Для вычисления внимания входного вектора

X к вектору Y, вычисляются вектора $Q = W_Q X$, $K = W_K X$, $V = W_V Y$. Эти вектора используются для вычисления результата внимания по формуле (1.1).

Если коротко, то внимание устанавливает взаимоотношения слов в предложении. Оно показывает нам насколько важен каждый элемент (по отдельности).

$$\text{Attention}(Q, K, V) = \underbrace{\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)}_{\text{scores}} V, \tag{1.1}$$

где

- scores «оценивают» важность элементов (там лежат значения от 0 до 1).
- Q (Query), K (Key), V (Value) — матрицы входных элементов.
- d_k — нижняя размерность одной из этих матриц (длина части embedding'a).

1.4.4. MultiHead Attention

«Сердце» Transformer'ов — MultiHead Attention. Несколько слоёв внимания позволяют «следить» за разными частями входной последовательности (независимо от других). Добавляя больше этих слоёв, мы «следим» за бóльшим количеством частей последовательности.

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right) W_0, \tag{1.2}$$

где

- $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.
- W_i — матрицы коэффициентов для обучения.

1.4.5. Positional Encoding

Так как в Transformer'е нет ни рекурренции (recurrence), ни свёртки, нам нужно что-то, что будет использовать порядок элементов в последовательности.

$$PE(p, 2i) = \sin\left(\frac{p}{10\,000^{2i/d_{\text{model}}}}\right), \tag{1.3}$$

$$PE(p, 2i+1) = \cos\left(\frac{p}{10\,000^{(2i+1)/d_{\text{model}}}}\right), \tag{1.4}$$

где

- p (position) — позиция,
- i (dimension) — размер предложения.

1.4.6. Обзор архитектуры Transformer'a

В оригинальной статье [8] представлена следующая архитектура Transformer'a (см. рис.1.1).

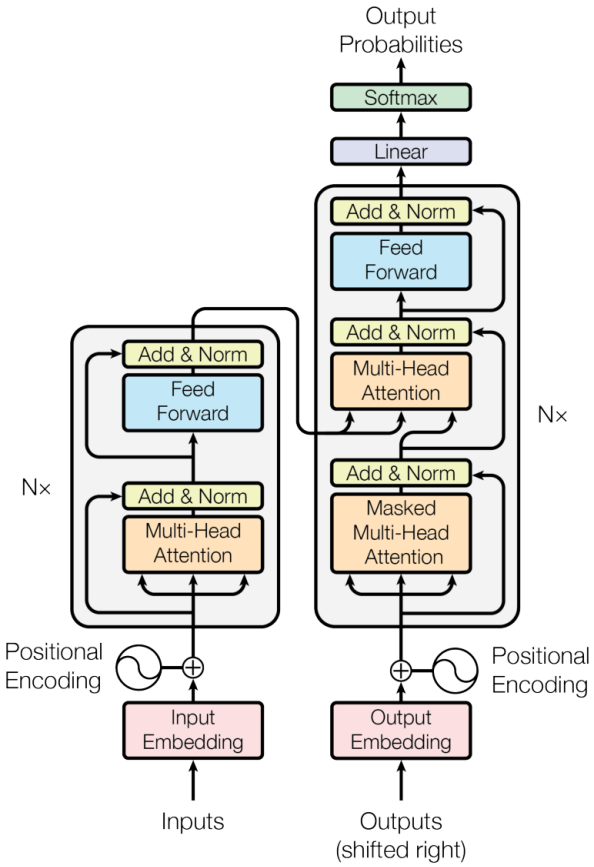

Рис.1.1. Архитектура Transformer'a

На рис.1.1 мы видим, что входные данные превращаются в эмбеддинги, после чего к ним суммируются positional encoding, после чего результат этой операции прогоняется через механизм MultiHead Attention, и, в итоге, через линейный слой со слоем Softmax мы получаем выходные вероятности элементов (слов в предложении).

1.4.7. Верхнеуровневый взгляд на Transformer

Какой высокоуровневый смысл несут encoder и decoder? Encoder представляет собой модель, «изучающую» исходный язык, его контекст (какие слова встечаются рядом в предложении, как следуют предложения друг за другом). То есть обучая encoder, мы обучаем модель «понимать» язык. Decoder же, в свою очередь, учится превращать контекст, полученный encoder'ом в какой-то осмыс-

ленный набор слов (в том числе и на другом языке). Иными словами, encoder получает информацию о предложениях, а decoder превращает эту информацию во что-то полезное (переведённое предложение, упрощённый текст, ответ на вопрос пользователя и т.д.)

О том, как это обучение происходит и за счёт чего encoder обучается контексту, мы поговорим в следующей главе.

ГЛАВА 2. ДЕТАЛИ ПРАКТИЧЕСКОЙ РЕАЛИЗАЦИИ

2.1. Выбор инструментов

В данной работе используются следующие инструменты:

1. Машинное обучение. Для обучения модели будет использоваться Python в связке с фреймворком для машинного обучения PyTorch [5], предоставляющий широкие возможности для реализации нейронных сетей, в том числе, там присутствует поддержка ранее упомянутых Transformer'ов.
2. Токенизация. Для токенизации японского текста будет использоваться библиотека MeCab [4].
3. Эмбеддинги. Готовые модели с эмбеддингами могут быть взяты из Python–библиотеки FastText [2] (в том числе там есть эмбеддинги для японского языка).
4. Сервер. Тут, опять же, будет использован Python с фреймворком Falcon [1], позволяющим создавать легковесный back-end. То есть будет реализован REST API сервер.
5. Веб-приложение. Будет использоваться TypeScript [6] с библиотекой Lit [3] (библиотека для веб-компонентов).

Система будет доступна в браузере в виде простого веб-приложения, то есть модель будет обучена на Python, а пользоваться обученной моделью можно будет в любом современном браузере (пользователю ничего не нужно будет устанавливать).

2.2. Как устроен Transformer изнутри

Для ускорения вычислений в PyTorch используются не матрицы размерности $N \times M$, а тензоры размерности $B \times M \times N$ (где B — размер батча), то есть матрицы обрабатываются батчами размера B. Ускорение происходит за счёт оптимизированного вычисления батчей на видеокартах в PyTorch. Однако для простоты изложения будем считать, что работаем мы с матрицами.

2.2.1. Механизм внимания

Вернёмся к формуле (1.1). Программно её можно реализовать следующим образом:

```
def scaledDotProductAttention(
  query: Tensor,
  key: Tensor,
  value: Tensor,
  mask: Optional[Tensor] = None
) -> Tensor:
  # Считаем scale, на который будем делить
  scale = query.size(-1) ** 0.5
  # Перемножаем матрицы query и key, делим их на scale
  temp = query.bmm(key.transpose(1, 2)) / scale

  # Применяем маску, если она есть
  if (mask is not None):
    temp += mask

  # Применяем softmax к измерению embedding'ов
  softmax = f.softmax(temp, dim=-1)
  # Перемножаем softmax с матрицей value
  return softmax.bmm(value)
```

Обратим внимание на то, что в коде используется некая маска. О том, что это и зачем она нужна, поговорим в следующем разделе.

2.2.2. Маска в механизме внимания

Как же выглядит маска? Просто создаётся треугольная матрица формы size×size, в левой части которой «0», а в правой — «$-\infty$». Нужно это для того, чтобы при обучении не показывать полностью переведённые (упрощённые) предложения модели (защита от переобучения). То есть «$-\infty$» при суммировании маски со scores заставляет модель «принебречь» частями предложения. Вид матрицы представлен на рис.2.1.

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.1}$$

Реализовать создание такой матрицы довольно несложно, в PyTorch это можно сделать следующим образом:

```
def generateSquareSubsequentMask(size: int, device: torch.device) -> Tensor:
  # Создаём треугольную матрицу
  mask = (torch.triu(torch.ones((size, size), device=device)) == 1).transpose(0, 1)
  # Переводим её в формат float с 0-ми и -inf
  mask = mask.float() \
```

```
      .masked_fill(mask == 0, float("-inf")) \
      .masked_fill(mask == 1, float(0.))
    return mask
```

2.2.3. Positional Encoding

Реализовать positional encoding тоже не составляет большого труда:

```
def positionalEncoding(
  sequenceLength: int,
  dModel: int,
  device: torch.device
) -> Tensor:
  # Тензор [[[0.], [1.], [2.], ...]]
  pos = torch \
    .arange(sequenceLength, dtype=torch.float, device=device) \
    .reshape(1, -1, 1)
  # Тензор [[[0., 1., 2., ...]]]
  dim = torch.arange(dModel, dtype=torch.float, device=device).reshape(1, 1, -1)
  # Фаза (аргумент для cos/sin) =
  # [
  #   [
  #     [0., 0., 0., ...],
  #     [1., 1., 1., ...],
  #     [2., 2., 2., ...],
  #     ...
  #   ]
  # ]
  phase = pos / 10000 ** (dim // dModel)

  # [[[sin(...),  cos(...), sin(...),  cos(...), ...], ...]]
  return torch.where(dim.long() % 2 == 0, torch.sin(phase), torch.cos(phase))
```

2.2.4. Как encoder учится понимать контекст

Чтобы научить encoder понимать смысл текста на каком-либо языке, нужно его каким-то образом обучить (предоставить данные для обучения). Отличная новость заключается в том, что данные для обучения можно получить относительно несложно, причём в очень больших объёмах. В модели BERT [7], к примеру, сделали следующее: для большого набора языков выгрузили все статьи википедии для каждого из них. После этого процесс обучения заключается в том, чтобы передавать encoder'у предложения, но не в исходном виде, а с замаскированными словами (вспомним маску, о которой мы говорили ранее), чтобы encoder пытался угадать, какое слово должно стоять в предложении. Тоже самое делается и с предложениями — модели подаются 2 предложения, и она должна определить,

следует ли оно предложение за другим. Вся прелесть в том, что весь этот процесс обучения не нуждается в ручной обработке — маскировка случайных слов и подставление двух случайных предложений реализуется программно, причём довольно просто.

После чего эти большие объёмы данных прогоняются через encoder, обучая его, а на выходе мы получаем модель, имеющую довольно обширное знание о законах языка, о том, как строятся слова в предложениях, а также как строятся сами предложения. Далее нужно лишь дообучить decoder для решения нужной нам задачи (fine-tuning), что можно сделать даже на относительно небольшом массиве данных. В итоге мы получаем модель, превосходящую современные решения в мире NLP.

2.3. Сервер

Как уже было сказано ранее, сервер использует фреймворк Falcon. Имеется лишь один путь — `/processJapaneseText`, по которму пользовательское приложение передаёт запрос с японским текстом, после чего сервер, используя функции `parse` и `simplify`, упрощает это предложение и возвращает результат приложению.

2.3.1. Настройка сервера

Чтобы использовать сервер, его необходимо сначала настроить. Так как написан он на Python, нам, естественно, понадобится установленный Python и менеджер пакетов pip, который поставляется вместе с ним.

Далее требуется установить необходимые зависимости через pip:

- `pip install falcon` — фреймворк Falcon,
- `pip install mecab` — библиотека MeCab,
- `pip install pytorch` — библиотека PyTorch,
- `pip install fasttext` — библиотека FastText,
- `pip install waitress` — библиотека для запуска сервера (для Windows),
- `pip install gunicorn` — библиотека для запуска сервера (для UNIX).

Теперь сервер может быть запущен командой:

- `waitress-serve --port=8000 server:app` — на Windows,
- `gunicorn --port=8000 server:app` — на UNIX.

2.4. Пользовательское приложение

Пользовательское приложение выглядит довольно минималистично — есть форма ввода предложения, после нажатия на кнопку «Simplify» или нажатия «Enter» отправляется запрос на сервер (по пути `/processJapaneseText`), после чего ответ отображается в виде, как на рис.2.1.

Есть также возможность посмотреть перевод предложений (исходного и упрощённого) — при нажатии на ссылку «translation» открывается страница Google Translate с выбранным предложением — это может использоваться как некая опорная линия упрощения (что смысл не потерялся).

Цвета в предложениях (исходном и упрощённом) на рис.2.1 указывают на часть речи какого-либо токена (слова).

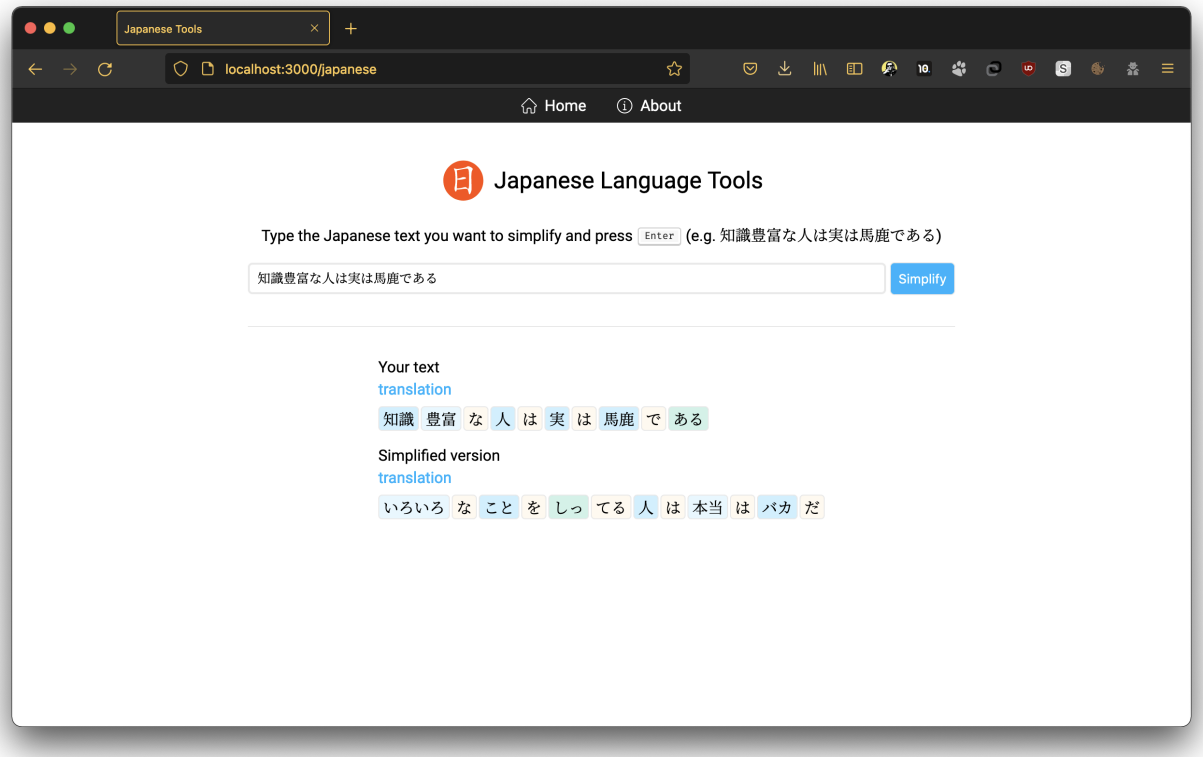

Рис.2.1. Пользовательское приложение

2.5. Объяснение выбранной архитектуры

Описанная архитектура «приложение-сервер» выбрана неслучайно. Дело в том, что в отличие, к примеру, от настольного приложения, пользователю ничего не нужно скачивать — он просто открывает веб-приложение и может пользоваться

приложением. Обученная модель используется на сервере, что позволяет её чаще обновлять, а возможно даже и заменить на более удачную.

Это также может позволить другим разработчикам взять готовую часть решения — например, взять уже существующее приложение и использовать там свою модель для упрощения. Или же, наоборот, использовать в своём существующем приложении сервер, представленный в данной работе, просто делая к нему запрос по ранее упомянутому пути (так как сервер не имеет привязки к приложению, сделать это довольно просто, нужно лишь настроить домены, которым будет дан доступ к серверу).

ЗАКЛЮЧЕНИЕ

В данной научно-исследовательской работе была выполнена поставленная цель — создание прототипа будущей системы для упрощения текста. Для достижения этой цели были выполнены следующие задачи:

- обзор общих этапов обработки естественных языков в контексте японского языка;
- краткий обзор искусственных нейронных сетей, различных моделей, использующихся в задачах перевода (в частности, упрощения) текстов на естественных языках;
- обзор архитектуры модели Transformer, как с теоритической, так и с практической точек зрения;
- описание архитектуры реализуемой системы, обзор инструментов для разработки.

Задачей для следующего этапа является окончательная программная реализация описанной в данной работе системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Falcon. Python web framework. — URL: https://falcon.readthedocs.io/en/stable/ (дата обращения: 06.02.2022).

2. FastText. Python llibrary for efficient text classification and representation learning. — URL: https://fasttext.cc (дата обращения: 06.02.2022).

3. Lit. JS web-components library. — URL: https://lit.dev (дата обращения: 06.02.2022).

4. MeCab. Japanese language tokenization. — URL: https://github.com/taku910/mecab (дата обращения: 21.05.2021).

5. PyTorch. Python machine learning framework. — URL: https://pytorch.org (дата обращения: 06.02.2022).

6. TypeScript. Superset of JavaScript with types. — URL: https://www.typescriptlang.org (дата обращения: 06.02.2022).

7. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / J. Devlin [и др.] // CoRR. — 2018. — Т. abs/1810.04805. — arXiv: 1810.04805. — URL: http://arxiv.org/abs/1810.04805.

8. *Maruyama T.*, *Yamamoto K.* Extremely Low Resource Text simplification with Pre-trained Transformer Language Model. — 2019. — DOI 10.1109/IALP48816.2019.9037650.

9. *Батура Т. В.* Математическая лингвистика и автоматическая обработка текстов на естественном языке. — Новосибирск: Новосиб. гос. ун-т., 2016. — 166 с.