МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

ГЕНЕРАЦИЯ ЗАГОЛОВКА ПО КРАТКОМУ СОДЕРЖАНИЮ ТЕКСТА

КУРСОВАЯ РАБОТА

студентки 3 курса 341 группы

направления 02.03.01 — Математическое обеспечение и администрирование информационных систем

факультета КНиИТ

Загудалиной Вероники Павловны

Научный руководитель

ст. пр. _____ А. А. Казачкова

Заведующий кафедрой

доцент, к. ф.-м. н. _____ М. В. Огнева

Саратов 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современную эпоху характеризует стремительное развитие вычислительных технологий, глобальных сетей и цифровых коммуникаций, что ведёт к экспоненциальному росту числа пользователей Интернета и объёмов создаваемых данных. Согласно прогнозам, к 2025 году мировой объём цифровой информации достигнет 463 миллиардов гигабайт [1]. Столь масштабный рост требует новых подходов к обработке, анализу и визуализации данных, которые сегодня обеспечиваются, в первую очередь, технологиями искусственного интеллекта (ИИ), машинного обучения (ML) и обработки естественного языка (NLP). Эти инструменты автоматизируют работу с текстовой информацией, делая взаимодействие человека с цифровой средой более эффективным [2].

Особенно остро эта необходимость проявляется в сфере текстового контента — новостей, научных статей, художественных произведений и пользовательских публикаций, объёмы которых ежедневно увеличиваются. Для облегчения восприятия и повышения доступности такой информации требуются эффективные инструменты её обработки. Одним из ключевых среди них является автоматическая генерация заголовков, то есть создание краткого, выразительного текста, отражающего основную идею документа.

В новых условиях заголовки эволюционировали: из простого средства привлечения внимания они превратились в самостоятельный инструмент влияния на восприятие контента. Теперь они выполняют такие функции, как усиление вовлечённости, повышение кликабельности, поисковая оптимизация и адаптация под алгоритмы социальных сетей. Эта трансформация тесно связана с феноменом «заголовочного чтения», когда пользователи зачастую ограничиваются лишь заголовком, не обращаясь к полному тексту [3].

Целью курсовой работы является проектирование и разработка системы автоматической генерации заголовков для текстовых документов.

Задачи:

1. Изучить современные методы автоматической генерации заголовков и обзоры аналогичных исследований.
2. Рассмотреть библиотеки и инструменты для обработки естественного языка (NLP), включая парсинг текстов и подготовку данных.
3. Сбор и обработка текстовых данных:
    а) Парсинг сайтов с литературными произведениями и публикациями.
    б) Формирование датасета с текстами и соответствующими заголовками.
4. Разработка и обучение моделей генерации заголовков:
    а) Реализация базовых моделей.
    б) Реализация и обучение Seq2Seq-модели.
5. Оценка качества работы моделей.

1 Теоретические основы моделей генерации

1.1 Сбор данных

Парсинг веб-страниц – это процесс извлечения данных с веб-страниц, обычно с использованием специальных программ или скриптов.

Веб-страницы обычно написаны на языке HTML, который определяет структуру и содержание страницы. При парсинге веб-страниц данные извлекаются из HTML-кода, обрабатываются и преобразуются в удобный для использования формат, такой как текст, таблицы, JSON или XML.

Парсинг веб-страниц может включать в себя различные этапы, такие как загрузка HTML-кода страницы, поиск нужной информации среди различных тегов и атрибутов, обработка данных и сохранение их в нужном формате. Для парсинга веб-страниц часто используются специализированные библиотеки и инструменты, такие как BeautifulSoup, lxml, Scrapy, Selenium и другие.

Извлеченные данные могут использоваться для различных целей, таких как анализ, отслеживание изменений, автоматизация задач, создание персонализированных приложений или уведомлений.

Однако при использовании парсинга веб-страниц важно учитывать правила использования сайта и законы о защите данных, чтобы не нарушать авторские права или правила конфиденциальности [4].

1.2 Предобработка и анализ текста

Обработка естественного языка начинается с работы с сырыми данными, которые зачастую являются неструктурированными.

Эти данные часто содержат ошибки, неоднозначности или избыточность, что делает их сложными для анализа. Поэтому важным этапом является предварительная обработка текста, которая включает удаление лишних символов, преобразование регистра, устранение стоп-слов и другие шаги для приведения данных к стандартному виду.

Нормализация текста означает его преобразование в более удобную, стан-

дартную форму. Например, большая часть того, что мы собираемся делать с языком, основывается на выделении или токенизации слов из текста - задача токенизации [5].

Токенизация – это процесс разбиения фразы, предложения, абзаца или всего текстового документа на более мелкие единицы, например, отдельные слова или термины. Каждое из этих меньших подразделений называется токенами.

Перед обработкой естественного языка нужно определить слова, которые составляют строку символов. В связи с этим токенизация является основным шагом для работы с NLP. Важность токенизации обусловлена тем, что значение текста можно легко интерпретировать, анализируя слова, присутствующие в тексте.

Токенизированную форму можно использовать для подсчета базовых статистик, например, количества слов в тексте или частоты слова, как необходимый шаг перед более сложными шагами обработки текста [6].

Другой частью нормализации текста является лемматизация - задача определения того, что два слова имеют один и тот же корень, несмотря на их поверхностные различия. Например, слова пел, спетый и петь являются формами глагола петь. Слово петь является общей леммой этих слов, и лемматизатор переводит их все в «петь». Лемматизация необходима для обработки морфологически сложных языков, например, для стемминга русского языка. Под стеммингом понимается более простая версия лемматизации, в которой мы в основном просто удаляем суффиксы с конца слова.

В текстах часто встречаются слова, не влияющие на смысл и лишь создающие помехи при анализе эмоциональной окраски. Эти слова, известные как стоп-слова, включают в себя:

- Имена
- Числовые значения
- Вводные конструкции

- Служебные части речи (предлоги, частицы, союзы)
- Местоимения
- Междометия
- Ссылки
- Пунктуационные знаки

Для их фильтрации применяются следующие подходы:

Методы обработки

1. Регулярные выражения — эффективны для удаления пунктуации, цифр и других легко идентифицируемых элементов.
2. Предопределённые списки — популярные стоп-слова (предлоги, союзы, междометия и т.д.) заранее сохраняются в файлы, а затем загружаются в множество Python.

Алгоритм удаления

1. Текст разбивается на токены методом tokenize.
2. Каждый токен проверяется на вхождение в сножество стоп-слов.
3. Совпадающие токены исключаются из дальнейшего анализа.

Этот подход минимизирует шум в данных и повышает точность обработки текста [7].

1.3 Метрики оценки качества генерации текста

1.3.1 BLEU

Метрика BLEU (Bilingual Evaluation Understudy) была предложена в работе Папинени. В качестве метода автоматической оценки качества. Основная идея метрики заключается в том, чтобы измерять степень совпадения n-грамм (последовательностей из n слов) между кандидатом и одним или несколькими эталонами, выполненными человеком [8].

Ключевые компоненты алгоритма BLEU включают:

1. Модифицированная n-граммная точность: для каждого размера n-граммы (обычно от 1 до 4) вычисляется точность. В отличие от обычной точ-

ности, модифицированный вариант предотвращает искусственное завышение оценки за счёт многократного учёта одинаковых слов в кандидате. Количество совпадений для каждой n-граммы ограничивается максимальным количеством её появлений в любом одном эталонном переводе.

2. Штраф за краткость: чтобы наказать слишком короткие кандидаты, которые могут иметь высокую n-граммную точность, но не передавать полный смысл, вводится штраф за краткость (Brevity Penalty, BP).
3. Агрегация: итоговая оценка BLEU представляет собой взвешенное геометрическое среднее модифицированных точностей для разных n, умноженное на штраф за краткость.

   Итоговая формула для корпуса текстов имеет вид:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{1}$$

   где:

— p_n — модифицированная точность для n-грамм,
— w_n — вес, обычно $w_n = 1/N$ для равномерного взвешивания,
— N — максимальная длина n-граммы (по умолчанию 4),
— BP — штраф за краткость.

1.3.2 Perplexity

Для оценки качества языковых моделей часто используется показатель Perplexity (PPL) — функция вероятности, отражающая способность модели предсказывать слова в тестовом наборе данных. Чем лучше модель предсказывает последовательность слов, тем выше вероятность, которую она присваивает каждому слову, и тем ниже значение perplexity.

Идеальная языковая модель при этом должна для каждого слова в корпусе назначать вероятность 1 (для правильного слова) и 0 — для всех остальных. Однако на практике мы используем не абсолютные вероятности, а их нормированную форму, поскольку общая вероятность тестового набора убывает с

увеличением его длины.

Поэтому Perplexity служит нормированной метрикой, которая позволяет сравнивать модели, обученные на текстах различной длины. Она определяется как обратная вероятность тестового набора, нормированная по количеству слов (или токенов).

Для тестового корпуса $W = w_1, w_2, \ldots, w_N$ perplexity вычисляется по формуле:

$$\text{Perplexity}(W) = P(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}} \tag{2}$$

Используя правило цепочки вероятностей, можно разложить вероятность последовательности W на условные вероятности отдельных слов:

$$\text{Perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i \mid w_1, \ldots, w_{i-1})}} \tag{3}$$

Из формулы (3) следует, что чем выше вероятность последовательности слов, предсказанная моделью, тем ниже perplexity. Следовательно, минимизация perplexity эквивалентна максимизации вероятности тестового корпуса, что делает метрику удобным показателем качества языковой модели.

Использование обратной вероятности связано с исходным определением perplexity, происходящим из коэффициента перекрёстной энтропии в теории информации. То есть perplexity имеет обратную зависимость от вероятности предсказанной последовательности [9].

1.3.3 ROUGE

Развитием метрик семейства BLEU, ориентированных преимущественно на показатель точности, стало появление группы метрик ROUGE, которые, помимо точного совпадения элементов текста, уделяют особое внимание измерению полноты (recall). Показатель полноты отражает долю n-грамм, совпадающих

между оригинальным и сгенерированным текстом, относительно общего числа n-грамм в исходном тексте.

Формально метрика ROUGE-N представляет собой recall на уровне n-грамм между сгенерированным (candidate) резюме и набором эталонных (reference) резюме. Значение ROUGE-N вычисляется следующим образом:

$$\text{ROUGE-N} = \frac{\sum_{S \in \{\text{Reference Summaries}\}} \sum_{\text{gram}_n \in S} \text{Count}_{match}(\text{gram}_n)}{\sum_{S \in \{\text{Reference Summaries}\}} \sum_{\text{gram}_n \in S} \text{Count}(\text{gram}_n)} \tag{4}$$

где:

— n — длина n-граммы;

— gram_n — конкретная n-грамма длины n;

— $\text{Count}_{match}(\text{gram}_n)$ — максимальное количество вхождений данной n-граммы, встречающихся одновременно в сгенерированном тексте и в эталонных резюме;

— $\text{Count}(\text{gram}_n)$ — общее количество появлений этой n-граммы в эталонных текстах [10].

Помимо классической метрики ROUGE-N, существует целый ряд модификаций, позволяющих оценивать различные аспекты текстового сходства. Наиболее распространённые варианты включают:

— ROUGE-L, основанный на длине наибольшей общей подпоследовательности слов (LCS). Данный вариант учитывает порядок слов и хорошо подходит для оценки структурного сходства предложений.

— ROUGE-S, использующий совпадение так называемых «скип-биграмм» — пар слов, которые могут быть разделены произвольным количеством других слов. Такой подход позволяет обнаруживать частично совпадающие структуры, даже если порядок слов в тексте был изменён [11].

1.3.4 METEOR

Ещё одним инструментом для оценки качества автоматически создаваемого текста является метрика METEOR (Metric for Evaluation of Translation with Explicit ORdering). По своей сути она близка к ROUGE, однако отличается тем, что применяет заранее заданные правила выравнивания текста и учитывает совпадения не только на уровне точных слов, но и их синонимов. Такой подход позволяет более корректно оценивать случаи, когда система использует разные слова с одинаковым значением, а также принимать во внимание порядок слов в предложении, что делает оценку более точной.

В отличие от BLEU, метрика METEOR опирается на несколько типов соответствия — точное совпадение слов, совпадение по корням и совпадение по синонимам, что делает её более чувствительной к качеству генерации на уровне фраз и предложений. METEOR была создана как ответ на ограничения BLEU и стремится обеспечить более высокую согласованность автоматической оценки с человеческими суждениями.

По результатам экспериментов, использование METEOR на уровне словосочетаний показывает корреляцию с оценками экспертов 0,964, тогда как BLEU на тех же данных достигает лишь 0,817. На уровне отдельных предложений максимальная корреляция составляет 0,403 [11, 12].

1.4 Эволюция подходов генерации

1.4.1 Шаблонные методы генерации заголовков

Одним из первых подходов к автоматической генерации текста и заголовков были шаблонные (template-based) системы, широко применявшиеся в 1960–1990-х годах. Принцип их работы заключался в том, что задание структуры текста и заголовка осуществлялось человеком заранее, в виде набора правил или шаблонов. В дальнейшем система лишь подставляла в эти шаблоны фактические данные.

Основные особенности подхода:

1. Жёсткая структура: заголовок строился по заранее заданной грамматической схеме.
2. Подстановка данных: в качестве переменных использовались ключевые сущности текста — названия команд, счёт, место, время, имена участников.
3. Упрощение текста: для получения краткого заголовка из длинного текста часто применялись эвристики:
   — удаление предлогов, местоимений и вводных слов,
   — сохранение существительных и глаголов,
   — сокращение до ключевых событий.

   Примеры ранних систем:

1. ELIZA (1966) — имитация психотерапевта, основанная на шаблонах и правилах подстановки. Хотя она не генерировала заголовки, её принцип работы показывает ограниченность и прямолинейность rule-based подхода [13].
2. PARRY (1972) — симулятор пациента с паранойей, также использовавший заранее прописанные сценарии [14].
3. SHRDLU (1972) — система Терри Винограда, демонстрировавшая взаимодействие на естественном языке в ограниченном «мире блоков»; применялась грамматика и правила синтаксиса [15].

Шаблонные методы активно использовались в первых новостных агрегаторах и спортивных отчётах. Такие заголовки генерировались автоматически на основе статистики матчей и заранее определённого набора шаблонов.

1.4.2 Языковые модели на основе n-грамм

Классические статистические языковые модели, основанные на n-граммах, опираются на подсчёт частот последовательностей из n токенов в корпусе. N-граммные языковые модели используют предположение Маркова, которое утверждает, что в контексте языкового моделирования вероятность следующе-

го слова в последовательности зависит только от предыдущего(их) слова(слов) В простейшей форме вероятность появления токена w_i при заданном контексте $w_{i-(n-1):i-1}$ оценивается как

$$P_n(w_i \mid w_{i-(n-1):i-1}) = \frac{\mathrm{cnt}(w_{i-(n-1):i-1} w_i \mid \mathcal{D})}{\mathrm{cnt}(w_{i-(n-1):i-1} \mid \mathcal{D})}, \tag{1}$$

где $\mathrm{cnt}(\mathbf{w} \mid \mathcal{D})$ — количество появлений n-граммы $\mathbf{w}$ в обучающем корпусе $\mathcal{D}$, а n — заранее заданный гиперпараметр. Для $n = 1$ контекст $w_{i-(n-1):i-1}$ определяется как пустая строка ε, количество которой равно $|\mathcal{D}|$.

Однако на практике такая наивная модель сталкивается с проблемой разреженности данных: числитель в формуле может быть равен нулю, что приводит к бесконечной перплексии. Одним из способов решения этой проблемы является стратегия backoff [16]: если числитель равен нулю, уменьшают n на единицу и повторяют процесс до тех пор, пока числитель не станет положительным. Следует отметить, что backoff не формирует корректное распределение $P_n(* \mid w_{i-(n-1):i-1})$, поскольку эффективное n зависит от конкретного w_i, поэтому требуется дополнительное снижение вероятностей для нормализации распределения, например, метод Katz [17].

Исторически n-граммовые модели реализовывались через построение таблицы подсчёта n-грамм по обучающему корпусу. Каждая уникальная n-грамма сохраняется вместе с числом её появлений. Размер таких таблиц растёт почти экспоненциально с увеличением n. Например, оценка показывает, что таблица для 5-грамм на корпусе из 1,4 триллиона токенов займёт примерно 28 ТиБ дискового пространства. Поэтому традиционные n-граммовые модели ограничены малыми значениями n, чаще всего $n = 5$ [18, 19]. Малое значение n приводит к потере более богатого контекста, что снижает предсказательную способность таких моделей [20].

1.4.3 Feedforward Neural Network Language Model (FNNLM)

Одним из первых и наиболее влиятельных подходов к построению нейронных языковых моделей является нейронная вероятностная языковая модель (Neural Probabilistic Language Model), предложенная Yoshua Bengio и соавторами в 2003 году. Эта модель является прямой (feedforward) нейронной сетью, обучаемой предсказывать вероятность следующего слова по фиксированному контексту из n-1 предыдущих слов.

Основная проблема традиционных моделей типа n-грамм заключается в «проклятии размерности»: число возможных последовательностей слов растёт экспоненциально, поэтому большинство контекстов в тестовых данных отсутствуют в обучающей выборке. В результате требуется агрессивное сглаживание или сокращение контекста, что ухудшает качество генерации. Bengio указывает, что n-граммы фактически «склеивают» короткие фрагменты текста и не умеют учитывать сходство слов или более дальние зависимости между ними.

Для решения этих ограничений Bengio и соавторы предложили два ключевых механизма:

1. Обучение распределённых представлений слов: Каждому слову сопоставляется обучаемый вектор признаков — embedding. Тем самым слова с похожими синтаксическими и семантическими ролями оказываются близко друг к другу в пространстве признаков.
2. Моделирование вероятности следующего слова с помощью feedforward-сети Вероятность следующего слова вычисляется как гладкая функция от векторных представлений слов контекста. Поскольку функция непрерывна, небольшое изменение входа ведёт к небольшому изменению вероятности, что обеспечивает генерализацию на новые сочетания слов.

Feedforward-LM состоит из двух основных частей:

1. Слой представлений слов

   Матрица C размера $|V| \times m$, где каждая строка — embedding слова.

Для входного окна $(w_{t-n+1}, \ldots, w_{t-1})$ векторы конкатенируются: $x = (C(w_{t-1}), \ldots$

2. Нейронная сеть предсказания

   Сеть с одним скрытым слоем и функцией активации tanh: $y = b + Wx + U \tanh(d + Hx)$, где y_i — логиты вероятности для слова i.

   Выход нормализуется через softmax:

$$P(w_t = i | w_{t-1}, \ldots) = \frac{e^{y_i}}{\sum_j e^{y_j}}.$$

Таким образом, модель одновременно обучает:

— параметры сети,

— распределённые векторные представления слов.

Работа показывает, что использование FFNN-LM позволяет:

1. эффективно бороться с проклятием размерности,
2. учитывать дальние зависимости (до 5–6 слов в окне),
3. переносить знания за счёт сходства слововых embedding'ов (пример: предложение с «dog» имеет высокую вероятность, даже если обучалась модель на варианте с «cat») .

На реальных корпусах (Brown, AP News) FFNN-модели значительно снизили перплексию по сравнению с n-граммами — до 20–30%, что сделало подход архитектурно революционным [21].

2 Работа с данными

2.1 Парсинг данных

Веб-скрапинг (web-scraping) – это автоматизированное извлечение информации с веб-страниц. Например, сбор контактов из онлайн-каталога можно отнести к скрапингу. Поскольку ручной сбор больших массивов данных неэффективен, для этой задачи используют специальные программы – веб-скраперы.

С правовой точки зрения веб-скрапинг, как правило, не считается нарушением закона, так как он работает с общедоступными данными.

С ростом цифровизации значение веб-скрапинга увеличилось. Согласно исследованию компании, специализирующейся на информационной безопасности, более 50% (52%) интернет-трафика (без учета аудио- и видеопотоков) генерируется ботами – автоматизированными системами [22].

Для построения и обучения моделей машинного обучения требуется корпус данных, соответствующий предметной области исследования. Так как готовые наборы данных не всегда удовлетворяют требованиям по объёму, актуальности и тематике, было принято решение осуществить автоматизированный сбор текстовой информации из открытых источников. Основная цель парсинга — формирование репрезентативного набора текстов, который может быть использован для последующей предобработки и обучения моделей.

В качестве источников данных были выбраны:

1. Сайты с произведениями:
   - Проза.ру
   - ЛитПричал
2. Литрес — сайт по продаже книг, в котором для каждой книги писалась аннотация.
3. Брифли — сайт с краткими пересказами произведений.

Критериями выбора послужили: доступность данных, достаточный объём информации, а также структурированность представления материалов. Для

разработки парсеров были использованы следующие инструменты и библиотеки Python:

- requests — для получения HTML-кода страниц;
- BeautifulSoup — для анализа и извлечения информации из HTML;
- fake_useragent — для имитации различных User-Agent при отправке запросов и снижения риска блокировки;
- json — для сохранения собранных данных в структурированном формате;
- time, datetime — для управления задержками между запросами и работы с датами публикаций;
- tqdm — для визуализации прогресса выполнения парсинга.

Для каждого сайта был написан отдельный парсер. Перед написанием была изучена структура HTML документа каждого из сайтов для корректного сбора нужной информации. Рассмотрим подробно каждый.

2.1.1 Парсинг Проза.ру

Основной целью парсинга являлось получение текстов вместе с метаданными для последующей предобработки и обучения моделей машинного обучения.

Парсер реализован как класс ProzaRuParser со следующими основными методами:

Метод __init__: при создании объекта класса задаются базовый URL, заголовки запроса, время для ожидания ответа от сайта, задержка для запросов, чтобы не перегружать сайт, и файл для сохранения данных:

```
def __init__(self, base_url: str = "https://proza.ru/texts/list.html", delay: float =
↪   1.5):
    self.base_url = base_url
    self.headers = {"User-Agent": UserAgent().random}
    self.timeout = 10
    self.delay = delay
    self.output_file = "../data/temp_data/json/data_proza_ru.json"

    with open(self.output_file, 'w', encoding='utf-8') as f:
        json.dump({}, f, ensure_ascii=False, indent=4)
```

Метод _get_page: загружает страницу и возвращает BeautifulSoup-объект.

```
time.sleep(self.delay)
response = requests.get(url, headers=self.headers, timeout=self.timeout)
response.raise_for_status()
return BeautifulSoup(response.content, "html.parser")
```

Метод get_all_forms(): получает все категории, представленные на сайте Proza.ru.

Сначала он загружает HTML-код основной страницы с помощью вспомогательного метода _get_page(). Далее из полученного документа извлекается блок с разделами произведений, после чего метод проходит по каждому подразделу и собирает ссылки на все категории. Для каждой категории сохраняется название и полная ссылка на соответствующую страницу. В результате метод возвращает словарь, где ключом является название категории, а значением — URL страницы.

```
def get_all_forms(self) -> Dict[str, str]:
    """Получает названия и ссылки на разделы с малыми формами."""
    soup = self._get_page(self.base_url)
    if not soup:
        return {}

    works_block = soup.find('ul', attrs={'type': 'square', 'style':
    ↪  'color:#404040'})
    all_forms = works_block.find_all('ul', attrs={'type': 'square'})
    data_all_forms = {}
    for form in all_forms:
        category = form.find_all('a')
        for link in category:
            title = link.text.strip()
            full_link = "https://www.proza.ru" + link['href']
            data_all_forms[title] = full_link

    return data_all_forms
```

Метод clean_text() выполняет очистку HTML-контента и извлечение чистого текста из страницы.

Сначала метод создаёт объект BeautifulSoup для анализа HTML-кода. Затем удаляются все нерелевантные элементы, такие как: iframe, img, script, style, рекламные блоки (div.ads, div.advertisement) и блоки видео (div.video-blk, div.video-block). Дополнительно удаляются скрытые элементы с классом,

содержащим 'hidden'. После удаления всех лишних элементов текст извлекается с сохранением разделения строк, а пустые строки удаляются. Метод возвращает очищенный текст в виде строки.

```
def clean_text(self, html: str) -> str:
    soup = BeautifulSoup(html, 'html.parser')
    for element in soup(['iframe', 'img', 'script', 'style',
                         'div.video-blk', 'div.video-block',
                         'div.ads', 'div.advertisement']):
        element.decompose()

    for div in soup.find_all('div', class_=lambda x: x and 'hidden' in x):
        div.decompose()

    clean_text = soup.get_text(separator='\n', strip=True)
    lines = [line.strip() for line in clean_text.split('\n') if line.strip()]
    return '\n'.join(lines)
```

Метод get_works() выполняет сбор всех произведений в рамках выбранной категории на сайте Proza.ru.

Сначала метод загружает HTML-страницу категории с помощью вспомогательного метода _get_page(). Затем из документа извлекаются блоки с перечнем произведений. Для каждого произведения метод получает:

1. имя автора (author);
2. заголовок произведения (title);
3. ссылку на полную страницу (link);
4. текст произведения (text), который загружается и очищается с помощью метода get_text().

Собранные данные сохраняются в словарь, где ключом является имя автора, а значением — словарь с заголовком, ссылкой и текстом произведения. После обработки каждого произведения данные также добавляются в JSON-файл с помощью метода _update_output_file(), а между запросами делается задержка для предотвращения блокировки.

Метод возвращает словарь с произведениями категории.

```
def get_works(self, url: str, category_title: str) -> Dict[str, Dict[str, str]]:
    soup = self._get_page(url)
```

```
        if not soup:
            return {}

        works_block = soup.find_all('ul', attrs={'type': 'square',
        ↪  'style':'color:#404040'})
        data_small_works = {}

        for works_list in works_block:
            for work in works_list.find_all('li'):
                work_data = work.find('a')
                if not work_data:
                    continue
                try:
                    author = work.find('a', attrs={'class':'poemlink'}).text.strip()
                    title = work_data.text.strip()
                    link = "https://www.proza.ru" + work_data['href']
                    text = self.get_text(link)

                    data_small_works[author] = {
                        'link': link,
                        'title': title,
                        'text': text
                    }
                    self._update_output_file(category_title, title, data_small_works[title])
                    time.sleep(self.delay)
                except Exception as e:
                    print(f"Ошибка при обработке произведения: {e}")
                    continue

        return data_small_works
```

Метод parse_by_dates() выполняет сбор произведений за указанный период по дате публикации.

На вход метод получает:

1. start_date — дата начала периода в формате 'YYYY-MM-DD';
2. end_date — дата окончания периода в том же формате;
3. category_title — название категории произведений;
4. topic — идентификатор темы на сайте.

Метод преобразует даты в объекты datetime и организует цикл, который проходит по каждому дню периода в обратном порядке. Для каждой даты формируется URL с указанием дня, месяца, года и темы. Затем вызывается метод get_works(), который собирает все произведения на этой странице. Собранные данные добавляются в общий словарь result. В конце работы метод возвращает

словарь со всеми произведениями за указанный период, структурированный по авторам и названиям.

```
def parse_by_dates(self, start_date: str, end_date: str, category_title: str, topic: str)
↪   -> Dict[str, dict]:
    current_date = datetime.strptime(start_date, "%Y-%m-%d")
    end_date = datetime.strptime(end_date, "%Y-%m-%d")
    result = {}

    while current_date >= end_date:
        date_str = current_date.strftime("%Y-%m-%d")
        print(f"\nОбработка даты: {date_str}")

        day = current_date.strftime("%d")
        month = current_date.strftime("%m")
        year = current_date.strftime("%Y")

        url = f"{self.base_url}?day={day}&month={month}&year={year}&topic={top
        ↪   ic}"
        data_day = self.get_works(url, category_title)
        result.update(data_day)

        current_date -= timedelta(days=1)

    return result
```

Метод get_all_work() обеспечивает полный сбор всех форм и произведений внутри них на сайте Proza.ru.

Сначала метод получает список всех категорий форм с помощью метода get_all_forms(). Далее для каждой категории выполняются следующие действия:

1. вызывается метод get_works() для получения всех произведений, доступных на странице категории;
2. извлекается идентификатор темы (topic) из URL категории;
3. вызывается метод parse_by_dates() для сбора произведений за определённый период по дате публикации;
4. объединяются результаты обоих методов в один словарь merged_works, который содержит все произведения категории;
5. обновлённый словарь добавляется в общий словарь all_data, где ключом является название категории.

В результате метод возвращает полный словарь, структурированный по категориям, авторам и названиям произведений.

```
def get_all_work(self) -> Dict[str, Dict[str, Dict[str, str]]]:
    """Получает все малые формы и произведения внутри них."""
    all_data = {}
    all_forms = self.get_all_forms()

    for all_form_title, all_form_link in all_forms.items():
        print(f"\nОбработка категории: {all_form_title}")
        works = self.get_works(all_form_link, all_form_title)
        topic = re.search(r'topic=(\d+)', all_form_link).group(1)
        works_by_data = self.parse_by_dates("2025-07-18", "2025-07-11",
        ↪  all_form_title, topic)
        merged_works = {**works, **works_by_data}
        all_data[all_form_title] = merged_works
        time.sleep(self.delay)

    return all_data
```

Метод _update_output_file() отвечает за обновление JSON-файла при добавлении нового произведения.

Он выполняет следующие действия:

1. считывает существующие данные из файла JSON;
2. проверяет, существует ли категория произведения, и при необходимости создаёт новый словарь для неё;
3. добавляет или обновляет запись с заголовком произведения и его данными (author, title, text, link);
4. сохраняет обновлённый словарь обратно в JSON-файл.

В случае ошибки при чтении или записи файла метод выводит сообщение об ошибке.

```
def _update_output_file(self, category: str, title: str, work_data: dict):
    """Обновляет JSON-файл, добавляя новое произведение"""
    try:
        with open(self.output_file, 'r', encoding='utf-8') as f:
            existing_data = json.load(f)

        if category not in existing_data:
            existing_data[category] = {}
        existing_data[category][title] = work_data
```

```
            with open(self.output_file, 'w', encoding='utf-8') as f:
                json.dump(existing_data, f, ensure_ascii=False, indent=4)

        except Exception as e:
            print(f" Ошибка при обновлении файла: {e}")
```

Метод save_to_json() сохраняет весь собранный корпус данных в отдельный JSON-файл.

Он принимает на вход словарь с данными и имя файла для сохранения. В случае успешного сохранения выводится сообщение с подтверждением. Если возникает ошибка при записи файла, метод выводит сообщение об ошибке.

```
def save_to_json(self, data: dict, filename: str = "data/data_proza_ru.json"):
    """Сохраняет данные в JSON-файл."""
    try:
        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=4)
        print(f"\nДанные сохранены в {filename}")
    except Exception as e:
        print(f"Ошибка при сохранении JSON: {e}")
```

2.1.2 Парсинг ЛитПричал

Данный парсер реализует сбор текстов и метаданных с сайта «ЛитПричал» и принципиально отличается от парсинга Proza.ru рядом особенностей архитектуры ресурса. Основным объектом структуры являются жанры, а не категории. Это определяет общую логику обхода страниц.

Парсер реализован в виде класса LitPrichalParser, где в конструкторе задаются базовый URL, заголовки и таймауты запросов

```
def __init__(self, base_url: str = "https://www.litprichal.ru"):
    self.base_url = base_url
    self.headers = {"User-Agent": UserAgent().random}
    self.timeout = 10
```

Ключевым отличием является метод get_genres(). Он извлекает с главной страницы списка прозы все доступные жанровые разделы. Жанры представлены в блоках:

```
<div class="col-sm-6 col-md-4">...</div>
```

Каждый жанр содержит одну или несколько ссылок на подразделы. Метод собирает названия и формирует абсолютные ссылки через urljoin.

```
def get_genres(self) -> Dict[str, Dict[str, str]]:
    url = f"{self.base_url}/prose.php"
    soup = self._get_page(url)

    genre_blocks = soup.find_all("div", class_="col-sm-6 col-md-4")
    for block in genre_blocks:
        for genre_link in block.find_all("a"):
            name = genre_link.text.strip()
            link = urljoin(self.base_url, genre_link.get("href"))
            genres[name] = {"link": link}
```

В отличие от Proza.ru, где категории группировались в отдельные HTML-списки, здесь структура линейно распределена по bootstrap-блокам.

Сайт ЛитПричал использует постраничную навигацию внутри жанров. Для определения количества страниц применяется метод _get_page_count().

```
def _get_page_count(self, genre_url: str) -> int:
    pagination = soup.find("ul", class_="pagination")
    pages = pagination.find_all("li")
    last_page = int(pages[-1].find("a").get("href").strip("/").split("/")[-1].replace("p",
    ↪  ""))
```

Таким образом, логика сбора на ЛитПричале строится на последовательном переборе страниц.

Произведения собираются методом get_books(). На каждой странице ищутся элементы:

```
<div class="col-md-6 x2">...</div>
```

Из каждого извлекаются заголовок, ссылка и имя автора.

Для извлечения текста используется метод get_text(). Он выбирает второй блок div.col-md-12 x2, где находится основной текст.

```
text_blocks = soup.find_all("div", class_="col-md-12 x2")
return self.clean_text(str(text_blocks[1]))
```

На Proza.ru текстовая часть извлекалась из более сложной структуры страницы; здесь доступ проще и локализован в одном блоке.

Метод clean_text() структурно похож на аналогичный в Proza.ru, отличается отсутствием обработки некоторых специфичных элементов.

В отличие от Proza.ru отсутствует инкрементальное обновление JSON-файла. Все данные собираются в оперативной памяти и записываются по завершении.

Финальный сбор производится методом parse_all_in_genre(). Он получает все жанры, затем вызывает get_books() для каждого:

```
for genre_name, genre_data in genres.items():
    books = self.get_books(genre_data["link"])
    result[genre_name] = books
```

В целом логика данного парсинга адаптирована под архитектуру сайта ЛитПричал и нацелена на сбор произведений по жанрам с учётом пагинации, без анализа дат публикации и без инкрементального сохранения данных.

2.1.3 Парсинг Литрес

Парсер сайта ЛитРес предназначен для сбора метаданных о книгах и получения коротких текстовых аннотаций со страниц произведений. В отличие от сайтов Proza.ru и ЛитПричал, ресурс ЛитРес содержит преимущественно коммерческий контент и использует более сложную верстку, что усложняет извлечение информации.

Парсер реализован в виде класса LitresParser. При инициализации задаются базовый URL жанровой страницы, заголовки HTTP-запросов, таймауты и путь для сохранения результатов в JSON-файл.

```
def init(self, url: str):
    self.litres_url = “https://www.litres.ru”
    self.base_url = url
    self.headers = {“User-Agent”: UserAgent().random}
    self.timeout = 10
    self.filename = “/data/temp_data/litres.json”
```

Основной вспомогательный метод _get_page() отвечает за загрузку HTML-страницы. Он осуществляет запрос к серверу с задержкой, обрабатывает сетевые ошибки и возвращает объект BeautifulSoup:

```
def _get_page(self, url: str) -> Optional[BeautifulSoup]:
    time.sleep(1.5)
    response = requests.get(url, headers=self.headers, timeout=self.timeout)
    response.raise_for_status()
    return BeautifulSoup(response.content, "html.parser")
```

Для обхода каталога и извлечения списка книг используется метод get_books(). Он поддерживает пагинацию: на вход передается целевое количество страниц, после чего метод формирует URL вида: <base_url>?page=<номер>

На каждой странице ищутся карточки произведений:

```
<div class="Art-module__3wrtfG__content
↪  Art-module__3wrtfG__content_full">...</div>
```

Из каждого блока извлекаются:

— заголовок книги;

— ссылка на страницу произведения;

— текст аннотации (через метод get_text()).

При этом предусмотрена проверка на дублирование: если книга уже присутствует в ранее сохраненных данных, повторная обработка не выполняется.

```
for book in books:
    info = book.find("a", class_="ArtInfo-module__Y-DtKG__title")
    title = info.text.strip()
    link = self.litres_url + info.get("href")
    if title in books_data:
        continue
```

Извлечение текстовой информации организовано в методе get_text(). Аннотация находится внутри блока:

```
<div class="BookDetailsAbout-module__p8ABVW__truncate">...</div>
```

и затем уточняется вложенный элемент:

```
truncated = block.find("div", class_="Truncate-module__FwxwPG__truncated")
return truncated.text if truncated else None
```

В отличие от Proza.ru и ЛитПричала, на ЛитРес нельзя получить полный текст произведения из-за авторских прав и ограничений доступа. Поэтому парсер извлекает только доступные публичные аннотации.

Сохранение результатов производится инкрементально после каждой обработанной страницы. Метод save_to_json() записывает текущий словарь в файл:

```
def save_to_json(self, data: Dict, filename: str = None) -> None:
    with open(filename or self.filename, "w", encoding="utf-8") as f:
        json.dump(data, f, ensure_ascii=False, indent=4)
```

Перед обработкой выполняется загрузка уже имеющегося JSON-файла:

```
def _load_existing_data(self) -> Dict:
    try:
        with open(self.filename, "r", encoding="utf-8") as f:
            return json.load(f)
    except:
        return {}
```

Таким образом, структура данных постепенно дополняется, что позволяет продолжать сбор даже при прекращении работы или сетевых ошибках.

В целом логика парсинга ЛитРес характеризуется следующими особенностями:

— опора на пагинацию по параметру page;

— сохранение данных после каждой итерации;

— невозможность получения полного текста произведения;

— извлечение только открытых аннотаций;

— проверка на дубликаты по названию книги.

Такой подход обеспечивает устойчивость обработки и минимальные потери данных при длинных сериях запросов.

2.1.4 Парсинг Briefly

Парсер, реализованный для ресурса Briefly.ru, предназначен для извлечения кратких пересказов произведений мировой литературы.

Функциональность организована в классе ParsingBriefly. В конструкторе задаются базовый URL, заголовки HTTP-запросов, сетевые параметры и путь сохранения результатов:

```
def init(self, url: str):
    self.briefly_url = “https://briefly.ru”
    self.base_url = url
    self.headers = {“User-Agent”: UserAgent().random}
    self.timeout = 5
    self.filename = “../data/temp_data/json/briefly.json”
```

Получение HTML-страниц осуществляется вспомогательным методом _get_pag который генерирует задержки между запросами для имитации поведения человека:

```
def _get_page(self, url: str) -> Optional[BeautifulSoup]:
    self.human_delay()
    response = requests.get(url, headers=self.headers, timeout=self.timeout)
    return BeautifulSoup(response.content, “html.parser”)
```

Использование искусственных задержек снижает риск блокировки со стороны сервера.

Метод get_all_data() обходит карточки культур (например, русская, французская, античная), находящиеся на корневой странице Briefly.ru. Для каждой культуры извлекаются имена авторов:

```
cultures_cards = soup.find_all(“a”, class_=“visited-hidden”)
for culture in cultures_cards[7:]:
    name = culture.get_text(strip=True)
    link = self.briefly_url + culture.get(“href”)
    data_culture = self.get_authors(link, name)
```

Таким образом обеспечивается последовательный обход каталога.

Парсер использует метод get_works(), который распознаёт две возможные структуры страницы:

— works_index — содержит список полных и кратких пересказов;

— author_works — альтернативное оформление профиля автора.

Дополнительно выполняется фильтрация технических названий (например, «глава», «том», «действие»), чтобы не загружать фрагменты произведений:

```
if any(word in title.lower() for word in [“глава”, “том”, “действие”]):
    continue
```

Извлечение текста реализовано в методе get_text(), который учитывает различные варианты вёрстки страниц:

— краткие пересказы (pending) находятся в div.microsummary__content;

— полные пересказы (published) — в p.microsummary__content;

— если формат иной — применяется резервный поиск в divtext.

Перед сохранением нежелательные рекламные элементы удаляются:

```
for ad in main_div.find_all("div", class_="honey"):
    ad.decompose()
```

Завершающий текст формируется объединением всех параграфов.

Отличительной особенностью является метод human_delay(), который:

— создаёт случайные задержки между запросами;

— с небольшой вероятностью инициирует длинную паузу;

— делает парсер менее «роботоподобным».

```
delay = random.uniform(base, base + var)
if random.random() < long_pause_prob:
    time.sleep(random.uniform(5, 15))
```

Подобный механизм снижает риск антибот-ограничений.

Для сохранения прогресса используется подход постепенной записи в JSON-файл:

```
self.save_to_json(all_data)
```

В случае прерывания процесса данные не теряются, и загрузка продолжается с последнего сохранённого состояния.

Разработка обладает рядом отличительных черт:

— глубокая вложенность навигации (культура → автор → произведения);

— обработка нескольких шаблонов вёрстки страниц;

— фильтрация неполных текстов (глав, томов, действий);

— удаление рекламных блоков;

— антибот-стратегия (случайные паузы);

— инкрементальное сохранение данных.

Благодаря этому достигается устойчивость работы при обработке большого количества страниц и минимизируется риск блокировок.

2.2 Предобработка и анализ данных

На каждом из этих сайтов тексты имели различный формат представления, поэтому на первом этапе требовалось привести их к единому виду и очистить от лишней информации.

Были написаны вспомогательные функции для открытия и преобразования данных из формата JSON в DataFrame.

Были реализованы две функции:

1. open_json() — открывает JSON-файл и возвращает его содержимое в виде словаря.

```
def open_json(input_file: str) -> dict:
    with open(input_file) as json_file:
        data = json.load(json_file)
    return data
```

2. json_to_csv() и json_to_csv_lp() — преобразуют данные разных форматов (в зависимости от сайта-источника) в таблицу с двумя основными столбцами:
   — title — название произведения,
   — text — текст произведения или его описание.

```
def json_to_csv_lp(json_data: dict) -> DataFrame:
    rows = []
    for category, works in json_data.items():
        for author, work_data in works.items():
            rows.append({
                "title": work_data.get("title", ""),
                "text": work_data.get("text", "")
            })

    df = pd.DataFrame(rows)
    return df

def json_to_csv(json_data: dict) -> DataFrame:
    rows = []
    for title, text in json_data.items():
        rows.append({
```

```
            "title": title,
            "text": text
        })

    df = pd.DataFrame(rows)
    return df
```

Далее все таблицы были объединены в один общий датасет:

```
data = pd.concat([briefly, litprichal, proza_ru, litres], ignore_index=True)
```

Проверка количества записей показала, что данные объединились без потерь.

На этапе первичной очистки все текстовые данные были приведены к нижнему регистру и очищены от лишних пробелов:

```
data = data.apply(lambda col: col.str.lower().str.strip())
```

Затем были удалены пропущенные значения (NaN) и пустые строки. Это позволило избавиться от записей, в которых отсутствовали либо текст, либо название.

Для текстов описаний была создана функция clean_text(), которая выполняла следующие действия:

— удаляла ссылки (http), HTML-теги и спецсимволы;

```
text = re.sub(r"http\S+", "", text)
```

— заменяла несколько пробелов на один;

```
text = re.sub(r"<[^>]+>", "", text)
```

— оставляла только буквы, цифры и основные знаки пунктуации.

```
text = re.sub(r"[^\w\s,.!?-]", " ", text)
```

Результат — тексты были приведены к единому читаемому виду, пригодному для анализа и подачи модели.

Во многих названиях встречались элементы нумерации, такие как «том 2», «глава 5», «эпизод 3» и т.п. Для их удаления была реализована функция clean_title_completely.

clean_title_completely(title) — функция для «полной» очистки названия произведения от любых форм нумерации (тома, главы, части, эпизоды и т.п.), а также от сопутствующего шумового мусора (скобки с числами, служебные символы, лишняя пунктуация и т.д.). Результат — компактная читабельная строка, пригодная как целевая метка для обучения модели генерации заголовков.

```
cleaned = remove_complex_volume_numbers(title)
cleaned = str(cleaned).lower().strip()
```

Complex_patterns используется для удаления более сложных конструкций нумерации, включающих сочетания нескольких типов меток (том, часть, эпизод, серия, глава, книга) и двух и более числовых значений.

```
complex_patterns = [
    r'\(?\s*\d+\s+(?:эпизод|серия|глава|часть)\s+\d+\s+(?:том|книга|т\.)\s*\d
    ↪  *\s*\.?\)?',
    r'\(?\s*\d+\s+(?:том|книга|т\.)\s+\d+\s+(?:эпизод|серия|глава|часть)\s*\d
    ↪  *\s*\.?\)?',
    r'\b\d+\s+\d+\s+(?:том|часть|книга|эпизод|серия)\b',
    r'\b(?:том|часть|книга|эпизод|серия)\s+\d+\s+\d+\b',
    r'\b\d+[-\.,]\s*\d+\s+(?:том|часть|книга)',
    r'\b(?:том|часть|книга)\s+\d+[-\.,]\s*\d+\b',
]

for pattern in complex_patterns:
    cleaned = re.sub(pattern, '', cleaned, flags=re.IGNORECASE)
```

1. \(?\s*\d+\s+(?:эпизод|серия|глава|часть)\s+\d+\s+(?:том|книга|т\.)\s*\d*\s*\.?\)?
   - — Ищет сложные конструкции вида: число + тип контента + число + тип издания;
   - — Например, (12 серия 3 том), 5 глава 2 книга.
2. \(?\s*\d+\s+(?:том|книга|т\.)\s+\d+\s+(?:эпизод|серия|глава|часть)\s*\d*\s*\.?\)?
   - — Обратный порядок: число + тип издания + число + тип контента;
   - — Например, (3 том 12 серия), 2 книга 5 глава.
3. \b\d+\s+\d+\s+(?:том|часть|книга|эпизод|серия)\b
   - — Ищет два числа подряд + тип контента/издания;

— Например, 12 3 том, 5 2 книга.

4. \b(?:том|часть|книга|эпизод|серия)\s+\d+\s+\d+\b
   — Обратный порядок: тип контента/издания + два числа;
   — Например, том 12 3, книга 5 2.
5. \b\d+[-\.,]\s*\d+\s+(?:том|часть|книга)
   — Ищет числа с разделителями + тип издания;
   — Например, 12-3 том, 5.2 книга.
6. \b(?:том|часть|книга)\s+\d+[-\.,]\s*\d+\b
   — Обратный порядок: тип издания + числа с разделителями;
   — Например, том 12-3, книга 5.2, часть 10,1.

Basic_patterns используется для того, чтобы удалить распространённые сочетания слов-меток (том, часть, глава, книга, эпизод, серия, выпуск, т.) вместе с последующими/предшествующими цифрами или римскими цифрами.

```
basic_patterns = [
    r'\s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)\s*[ivxlcdm0-9]+',
    r'\s*[ivxlcdm0-9]+\s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)',
    r'\s*\d+[-\.,]?\s*(?:том|часть|книга|глава)',
    r'\s*(?:том|часть|книга|глава)[-\.,]?\s*\d+',
]
for pattern in basic_patterns:
    cleaned = re.sub(pattern, '', cleaned, flags=re.IGNORECASE)
```

1. \s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)\s*[ivxlcdm0-9]+
   — Ищет слово типа том (или сокращение т., vol.) + пробелы + число (арабское) или римское (ivxlcdm);
   — \s* до и после — чтобы удалить предшествующий пробел и не оставлять двойных пробелов.
2. \s*[ivxlcdm0-9]+\s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)
   — Обратная форма: число перед словом;
   — Например, 2 том, iii том.
3. \s*\d+[-\.,]?\s*(?:том|часть|книга|глава)

— Захватывает варианты с разделителями между числом и словом;

— Например, 10-том, 10, том, 10. том.

4. \s*(?:том|часть|книга|глава)[-\.,]?\s*\d+

— Аналогичный случай, когда разделитель стоит после слова;

— Например, том-10, том. 10.

Для того чтобы удалить текстовые порядковые обозначения (глава вторая, часть первая и т.д.) был написан паттерн:

```
number_words = ["первая", "вторая", "третья", ... , "двадцатая"]
text_num_pattern = r'\b(?:глава|часть|том|эпизод|серия|книга|выпуск)\s+(?:' +
↪   "|".join(number_words) + r')\b'
cleaned = re.sub(text_num_pattern, '', cleaned, flags=re.IGNORECASE)
```

Для удаления форм с русскими окончаниями (падежные и порядковые суффиксы): 2-я глава, 3й том, 4ая часть, том 5-ая и т.п., были написаны паттерны:

```
cleaned = re.sub(
    r'\b\d+[-–]?(?:я|й|е|ой|ая|ое|ые|ых)?\s+(?:глава|часть|том|книга|серия|эпизод|в
    ↪   ыпуск)\b',
    '', cleaned, flags=re.IGNORECASE
)
cleaned = re.sub(
    r'\b(?:глава|часть|том|книга|серия|эпизод|выпуск)\s+\d+[-–]?(?:я|й|е|ая|ое|ые|
    ↪   ых)?\b',
    '', cleaned, flags=re.IGNORECASE
)
```

1. [-–]? учитывает разные дефисы/тире (обычный - и длинный –);
2. (?:я|й|е|ой|ая|ое|ые|ых)? — возможные окончания, которые часто встречаются у порядковых числительных.

Удаление символа «№»:

```
cleaned = re.sub(r'№', '', cleaned)
```

Также были удалены скобки с числами и одиночных чисел на конце/начале

```
cleaned = re.sub(r'\([^)]*\d+[^)]*\)', '', cleaned)
cleaned = re.sub(r'\s+\d+\s*\.?$', '', cleaned)
cleaned = re.sub(r'^\d+\s+', '', cleaned)
```

1. \([^)]*\d+[^)]*\) — удаляет любые круглые скобки, в которых есть цифры, например (том 2), (№10, переработанное);

2. \s+\d+\s*\.?$ — удаляет числа, стоящие в конце строки, возможно с точкой;
3. ^\d+\s+ — удаляет ведущие числа в начале строки.

После всего были удалены прочие нежелательные символы и нормализированы пробелы

```
cleaned = re.sub(r'[^\w\s.,!?-]', '', cleaned)
cleaned = re.sub(r'\s+', ' ', cleaned)
cleaned = cleaned.strip(' ,.-')
```

1. r'[^\w\s.,!?-]' — удаляет всё, кроме букв / цифр / подчёрки (\w), пробелов, и набора разрешённых знаков . , ! ? -. Это убирает лишние спецсимволы, кавычки, другие скобки, символы валют и т.п.
2. Затем сводим последовательные пробелы к одному и убираем ведущие / замыкающие пробелы и пунктуацию , . -.

В результате очистки было изменено 8666 названий из 32085.

После предобработки были просмотрены пропуски в данных. Просто NaN не было, зато встречались тексты и названия, в которых не было никаких символов. Также в текстах часто встречалась заглушка «описание отсутствует». Все такие строки были удалены.

```
data = data[data.text != ""].reset_index(drop=True)
data = data[data.title != ""].reset_index(drop=True)
data = data[data.text != "описание отсутствует"].reset_index(drop=True)
```

В некоторых строках встречался нерусский текст. Так как датасет собирается для модели генерации на русском языке, такие строки были почищены, чтобы не вносить модели лишний шум. Для этого была написана функция keep_only_russian, которая оставляла только русские символы, цифры и пунктуацию в тексте. Если после этого появлялись пропуски, то они удалялись.

```
def keep_only_russian(text):
    if pd.isna(text):
        return ""
    text = re.sub(r"[^А-Яа-яЁё0-9\s.,!?-]", "", text)
    text = re.sub(r"\s+", " ", text).strip()
```

```
    return text

data.text = data.text.apply(keep_only_russian)
data.cleaned_title = data.cleaned_title.apply(keep_only_russian)

data = data[(data.text != "") & (data.cleaned_title != "")].reset_index(drop=True)
```

После первичной обработки данных были дополнительно проверены строки, содержащие текст. В датасете встречались случаи, когда строка формально не была пустой, однако не содержала никаких значимых символов — только пробелы или пунктуацию. Такие строки не несут полезной информации и потенциально могут добавить шум при обучении языковой модели.

Чтобы отфильтровать подобные случаи, была написана вспомогательная функция is_meaningful, которая проверяет, содержит ли текст хотя бы один букво-цифровой символ (на русском или английском языке).

Также при парсинге на некоторых сайтах в начале текста встречалась лишняя ведущая пунктуация (например, «. . . », «——», «***»). Для устранения этого была реализована функция clean_leading_punct, которая удаляет начальные небуквенные символы с помощью регулярного выражения.

После применения этих функций датасет очищается от нерелевантных строк. Далее выполняется нормализация текста:

1. Замена всех символов, не относящихся к слову, пробелу или базовой пунктуации, на пробел;
2. Устранение избыточных пробелов.

В результате тексты становятся более однородными и удобными для дальнейшей обработки.

```
def is_meaningful(text):
    return bool(re.search(r"[А-Яа-яA-Za-z0-9]", text))

def clean_leading_punct(text):
    return re.sub(r"^[^\wА-Яа-я0-9]+", "", text).strip()

data = data[data.text.apply(is_meaningful)].reset_index(drop=True)
data.text = data.text.str.replace(r"[^\w\s,.!?-]", " ", regex=True)
data.text = data.text.str.replace(r"\s+", " ", regex=True).str.strip()
```

После нормализации была проведена проверка на дубликаты. Сначала было посчитано их общее количество (430), затем — количество повторяющихся значений отдельно по тексту (762) и названию (2127).

Оказалось, что дубликаты названий встречаются достаточно часто — это допустимо, так как одни и те же произведения могут быть размещены на разных сайтах и иметь одинаковые заголовки.

Однако более критичны случаи, когда совпадает текст, а названия различаются. Это означает, что одно и то же произведение было спарсено несколько раз с разных источников. Такие строки следует удалить, чтобы не дублировать обучающие данные и не вносить перекос в модель.

Для этого были удалены все повторяющиеся тексты, оставив от каждого дубликата только один экземпляр.

```
data.duplicated().sum()
data.text.duplicated().sum(), data.title.duplicated().sum()

data = data[~data.text.duplicated(keep=False)].reset_index(drop=True)
data.duplicated().sum()
```

Таким образом, датасет был очищен от нерелевантных строк и повторяющихся текстов, что положительно скажется на качестве обучения генеративной модели.

На следующем этапе был проведён анализ длины заголовков (названий произведений). Сначала было вычислено распределение частот появления каждого уникального заголовка, после чего для каждого названия была рассчитана длина — количество слов, содержащихся в нём. Это позволило сгруппировать данные и оценить, какие длины встречаются чаще всего.

Полученное распределение визуализировано в виде столбчатой диаграммы. По графику 1 видно, что в датасете присутствуют названия, состоящие из большого количества слов. Такие случаи обычно характерны либо для метаданных, случайно попавших в заголовок, либо для чрезмерно подробных описаний. Поскольку задача — обучить модель генерировать короткие и понятные художе-

ственные названия, слишком длинные заголовки могут вносить шум и снижать качество обучения.

```
data["title_len"] = data.title.apply(lambda x: len(str(x).split()))
length_counts = data.groupby('title_len').size()

plt.figure(figsize=(12,6))
length_counts.plot(kind='bar')
plt.xlabel("Длина названия (слов)")
plt.ylabel("Количество названий")
plt.title("Распределение длин названий в датасете без аугментации")
plt.show()
```

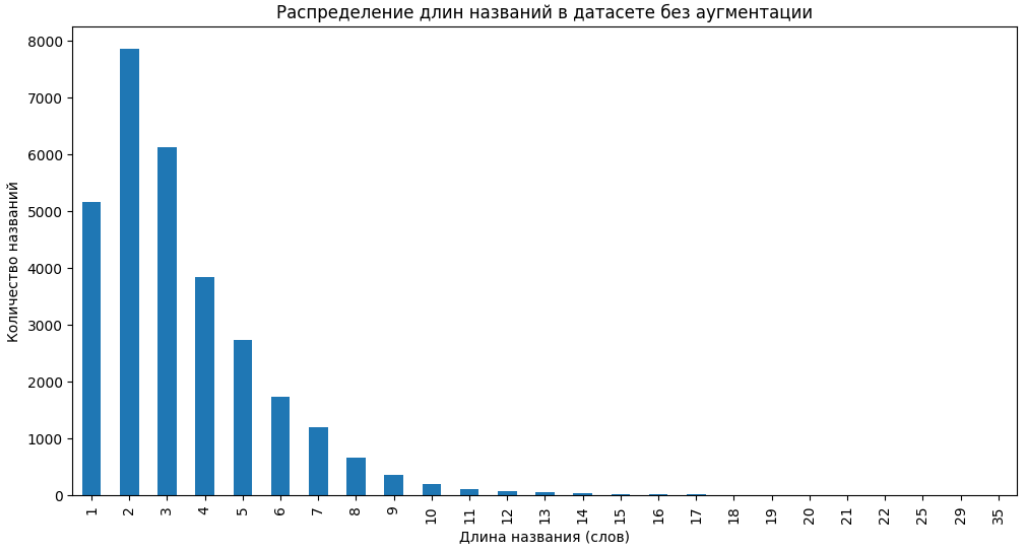

Рисунок 1 – Распределение длин названий в датасете без аугментации

После анализа было принято решение ограничить длину названий до 10 слов, поскольку более длинные примеры встречаются редко и не являются репрезентативными. Соответствующие строки были удалены, а датасет пересчитан.

Далее были рассчитаны дополнительные статистики: количество примеров, число уникальных заголовков, средняя длина текста и средняя длина заголовков. Эти значения позволяют оценить «средний масштаб» данных и убедиться, что после фильтрации структура выборки остаётся адекватной для обучения модели.

После этого вспомогательные столбцы title_len и text_len были удалены, а финальный датасет сохранён в CSV-файл для дальнейшего использования.

Чтобы дополнительно проанализировать частотность слов в названиях произведений, было построено облако слов, объединяющее все заголовки в единый корпус. Полученная визуализация, представленная на рисунке 2 демонстрирует, какие слова встречаются чаще всего.

```
all_titles = " ".join(data.title.astype(str))
wordcloud = WordCloud(
    width=1200, height=600,
    background_color="white",
    max_words=400,
    colormap="viridis",
    collocations=False,
).generate(all_titles)
```

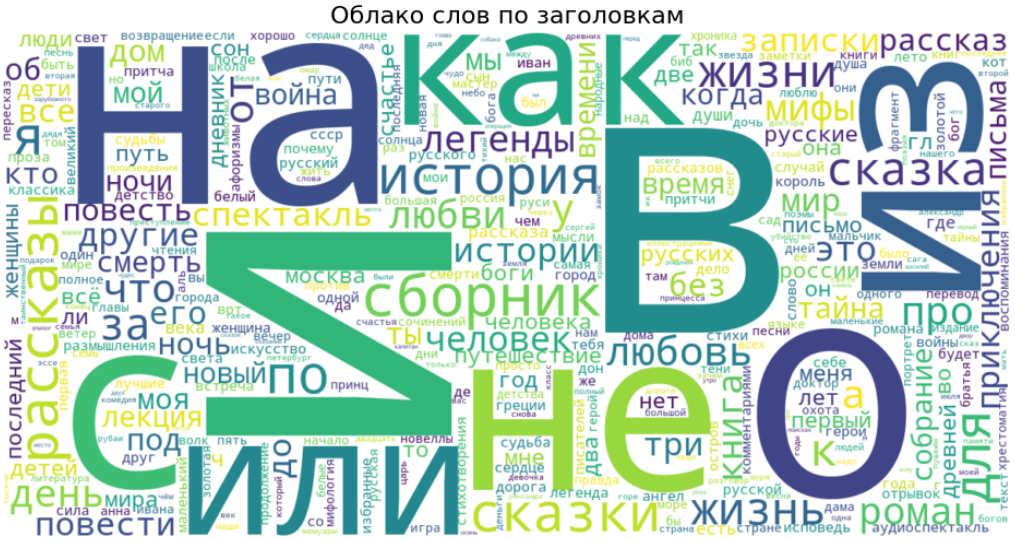

Рисунок 2 – Облако слов по заголовкам

На облаке слов заметно, что значительную роль играют предлоги, союзы и другие служебные слова. Это логично, поскольку они естественным образом чаще всего встречаются в русских названиях. Хотя такие слова можно было бы исключить для получения более информативного облака, в данном случае было решено оставить их, чтобы модель могла учиться более реалистичной структуре заголовков, приближённой к художественным текстам.

2.3 Аугментация данных

Следующим этапом работы с датасетом стала аугментация данных, направленная на увеличение объёма тренировочной выборки без непосредствен-

ного добавления новых источников. Это особенно важно при обучении моделей обработки естественного языка, так как наличие более широкого набора текстов способствует лучшей генерализации и снижению переобучения.

Перед аугментацией исходные данные были корректно разделены на тренировочную и валидационную выборки с учётом ограничения на повторяющиеся названия. Это позволило избежать потенциальной утечки информации: валидационная выборка после повторного разбиения не содержит текстов с одинаковыми заголовками, что улучшает объективность последующей оценки модели.

train_df, val_df = train_test_split(data, test_size=0.2, random_state=42)

После предварительного анализа стало видно, что в первоначальную валидационную выборку попали несколько текстов с одинаковыми названиями. Это создаёт риск искажения оценки качества модели, если одно и то же произведение (или его вариации) встречается и в обучении, и в валидации, модель может фактически «узнать» его структуру, а не обобщать.

Чтобы избежать подобной утечки данных, было выполнено контролируемое разделение датасета.

Для этого реализована функция split_with_controlled_test_size, которая:

1. Группирует записи по названию произведения.
2. Перемешивает список уникальных заголовков.
3. Для каждого заголовка случайным образом выбирает ровно один пример в валидационную выборку (пока не будет набран заданный размер).
4. Остальные примеры заголовка попадают в обучение.

Таким образом мы гарантируем, что один и тот же заголовок не встречается в обеих выборках одновременно, что обеспечивает более честную оценку генеративной модели.

def split_with_controlled_test_size(data, target_test_size=0.2, random_state=42):

```
    np.random.seed(random_state)

    title_groups = data.groupby('title').apply(lambda x: x.index.tolist()).to_dict()

    unique_titles = list(title_groups.keys())
    np.random.shuffle(unique_titles)

    train_indices = []
    test_indices = []

    target_test_count = int(len(data) * target_test_size)

    for title in unique_titles:
        indices = title_groups[title]

        if len(test_indices) < target_test_count:
            test_idx = np.random.choice(indices, 1)[0]
            test_indices.append(test_idx)
            train_indices.extend([idx for idx in indices if idx != test_idx])
        else:
            train_indices.extend(indices)

    return data.iloc[train_indices], data.iloc[test_indices]
```

После выполнения функции было выведено распределение размеров выборок, что позволило убедиться в корректности деления:

```
Общий размер данных: 29815
Тренировочная выборка: 23852 записей (80.0%)
Валидационная выборка: 5963 записей (20.0%)
```

После формирования тренировочной выборки был выполнен этап аугментации. Важно подчеркнуть, что аугментировать нужно только train-набор, чтобы избежать утечки информации в валидацию, которая служит для объективной оценки качества модели.

Поэтому перед началом процедуры индекс тренировочного датасета был сброшен.

Для увеличения количества обучающих примеров текст каждой записи был разбит на небольшие смысловые фрагменты. Разбиение происходит по предложениям с фиксированной длиной блока: в данном случае — по три предложения на каждый фрагмент. Это позволяет:

1. увеличить объём обучающих данных;

2. дать модели больше разнообразия контекста;
3. улучшить способность генерации коротких связных отрывков.

Для этого была реализована функция split_text_into_chunks, использующая токенизацию предложений:

```
def split_text_into_chunks(text, sentences_per_chunk=3):
    sentences = sent_tokenize(text, language="russian")
    chunks = []
    for i in range(0, len(sentences), sentences_per_chunk):
        chunk = " ".join(sentences[i:i + sentences_per_chunk])
        chunks.append(chunk)
    return chunks
```

Каждый chunk получает тот же заголовок, что и оригинальный текст, так как принадлежит одному произведению.

В итоге формируется новый датасет augmented_dataset, размер которого значительно превышает исходный train-набор.

После генерации chunk'ов был выполнен знакомый по предыдущим этапам этап очистки

1. Удаление строк, не содержащих значимых символов (букв или цифр);
2. Удаление лишней пунктуации в начале строки;
3. Замена нежелательных символов на пробел;
4. Нормализация пробелов и удаление хвостовых отступов.

После разбиения некоторые chunk'и оказались слишком короткими или малоинформативными. Такие строки не несут полезной контекстной нагрузки и могут ухудшить качество обучения, особенно при генерации связного текста.

Поэтому было добавлено дополнительное условие фильтрации:

```
augmented_dataset = augmented_dataset[augmented_dataset.text.str.strip().str.len()
↪   > 10]
```

Это позволяет:

1. Отсеять «обрывки»;
2. Исключить заглушки;
3. Сохранить только содержательные обучающие примеры.

После аугментации тренировочных данных (разбиения текстов на фрагменты) образовался существенно расширенный набор записей. Однако дальнейший анализ выявил ряд потенциальных проблем, которые необходимо устранить, чтобы избежать ухудшения качества обучения модели.

Из-за разбиения длинных текстов на множество фрагментов отдельные заголовки (title) начали появляться слишком часто. Это создает дисбаланс и может привести модель к переобучению на популярные заголовки, снижая её способность обобщать.

Чтобы равномерно распределить данные, было принято решение оставить не более 500 фрагментов на каждый заголовок:

```
max_per_title = 500
augmented_dataset = augmented_dataset.groupby("title").head(max_per_title).rese
↪   t_index(drop=True)
```

Таким образом, наиболее длинные произведения перестают доминировать в обучающем корпусе, а модель получает более стабильный распределённый по заголовкам набор данных.

После обрезки по количеству записей появился небольшой процент точных дубликатов, где совпадали и text, и title. Такие записи никак не обогащают датасет и фактически дублируют сигнал для модели, поэтому были удалены. Это позволяет снизить помощь модели «запоминать» конкретные фрагменты.

Следующий обнаруженный источник шума — случаи, когда один и тот же текстовый фрагмент встречается под разными названиями. Это может привести к обучению модели ложным связям: одинаковый контент начинает ассоциироваться с разными заголовками, что затрудняет обучение генерации.

Такие записи были удалены с помощью фильтрации по дубликатам на уровне столбца text:

```
augmented_dataset = augmented_dataset[~augmented_dataset.text.duplicated(keep
↪   =False)].reset_index(drop=True)
```

В завершение был выполнен дополнительный анализ:

```
dup_titles = (
   augmented_dataset.groupby("text")["title"]
```

```
    .nunique()
    .reset_index()
    .query("title > 1")
)
```

Он показывает, сколько фрагментов всё ещё имеют более одного уникального заголовка. После проведённой фильтрации количество таких случаев больше не было, что подтверждает корректность предпринятых действий.

После выполнения аугментации тренировочного набора данных было важно оценить, как изменилось распределение длин заголовков (в словах). Для этого были рассчитаны длины названий в трёх выборках:

— Исходный датасет без аугментации;

— Тренировочная выборка после аугментации;

— Валидационная выборка.

Измерение длины выполнялось по количеству слов в заголовке.

Далее была построена сравнительная гистограмма, представленная на рисунке 3, отображающая распределение полученных значений.

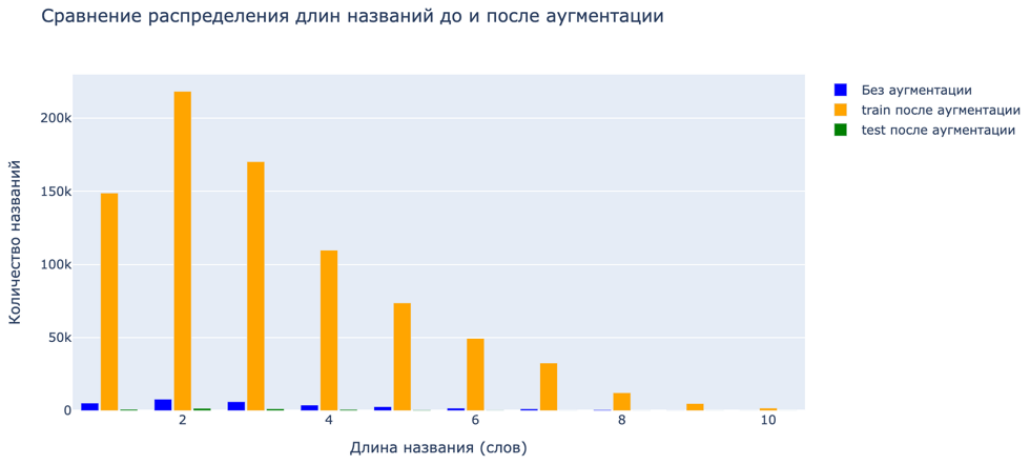

Рисунок 3 – Сравнение распределения длин названий до и после аугментации

По графику можно заметить следующие особенности:

1. Исходное распределение
   — Распределение достаточно равномерное в области коротких названий.

— Количество примеров сравнительно небольшое, так как это необработанный датасет без разбиения текста на части.

2. train после аугментации
   — Сильно увеличилось общее количество примеров.
   — Распределение визуально сохраняет схожую форму: короткие названия встречаются значительно чаще, чем длинные.
   — Это закономерно, так как аугментация увеличивает количество текстовых фрагментов, но название у каждого chunk'а остаётся прежним.
3. test/val после аугментации
   — Количество названий здесь существенно меньше, так как валидация purposely не аугментируется.
   — Распределение по длинам заголовков остаётся очень близким к исходному, что важно для корректной оценки модели.

3 Написание и обучение моделей

3.1 Модель на основе n-gram

Для подготовки текста к обучению модели используется функция tokenize, реализующая базовый этап предобработки входных данных. Её задача — преобразовать сырой текст в последовательность токенов, пригодных для последующего анализа и построения n-граммной модели.

На первом шаге весь текст приводится к нижнему регистру. Это устраняет различия между одинаковыми словами, записанными с разным использованием заглавных букв, и уменьшает размер словаря.

Затем выполняется очистка строки с помощью регулярного выражения. Из текста удаляются все символы, не являющиеся буквами латиницы или кириллицы, цифрами или пробелами.

После очистки строка разбивается на отдельные токены по пробелам. Результатом работы функции является список слов, упорядоченных в соответствии с исходным текстом. Эти токены далее используются для построения обучающих последовательностей и формирования n-грамм.

```
def tokenize(text: str) -> List[str]:
    """Функция самой простой предобработки текста, основанной на разбиении на
    ↪  токены по пробелам."""
    text = text.lower()
    text = re.sub(r"[^a-za-яё0-9\s]", "", text)
    return text.split()
```

Для решения задачи генерации названий по входному тексту используется класс TitleNgramModel, реализующий статистическую n-граммную модель. Она обучается на парах (текст + название) и оценивает вероятность появления следующего слова на основе нескольких предыдущих.

В методе __init__ задаются основные настройки и внутренние структуры данных, необходимые для работы модели:

1. n_gram — порядок модели, определяющий, сколько предыдущих токенов используется как контекст при прогнозировании следующего слова.

Например, при n_gram = 3 модель опирается на биграммный контекст;

2. ngrams — словарь, в котором каждому контексту сопоставляется счётчик слов, появляющихся после него. Это ядро модели: оно хранит частоты всех наблюдённых n-грамм;
3. context_counts — количество встреч каждого контекста; Используется при вычислении вероятностей;
4. vocab — словарь уникальных токенов, включающий наиболее частотные слова корпуса и специальные служебные символы;
5. Специальные токены:
   - <s> — начало последовательности;
   - <title> — разделитель, отделяющий исходный текст от целевого названия;
   - </s> — конец последовательности;
   - <unk> — токен для слов, отсутствующих в словаре.
6. smoothing — стратегия сглаживания вероятностей (по умолчанию используется Лапласовское сглаживание). Сглаживание предотвращает нулевые вероятности для слов, отсутствующих в обучающих n-граммах;
7. alpha — коэффициент, применяемый в формулах сглаживания;
8. lower_order_models — модели меньшего порядка (например, для 4-граммной модели автоматически создаются 1-, 2- и 3-граммные версии). Они используются для методов типа linear interpolation или backoff, позволяющих улучшать генерацию при редких или отсутствующих контекстах.

```
class TitleNgramModel:
    """Модель n-грамм, обучающаяся по парам (текст → название)."""

    def __init__(self, n_gram: int = 4, smoothing: str | None = 'laplace', alpha: float
    ↪  = 1.):
        self.n_gram = n_gram
        self.ngrams = defaultdict(Counter)
        self.context_counts = defaultdict(int)
        self.vocab = set()
        self.start_token = "<s>"
        self.title_token = "<title>"
        self.end_token = "</s>"
```

```
self.unknown_token = "<unk>"
self.smoothing = smoothing
self.alpha = alpha
self.lower_order_models = {}
```

Метод create_ngrams отвечает за построение последовательностей n-грамм, которые позднее используются при обучении модели. Он принимает на вход список токенов и преобразует его в набор перекрывающихся отрезков длины n, каждый из которых описывает локальный контекст в тексте.

```
def create_ngrams(self, tokens: List[str]) -> list[tuple[str, ...]]:
    """Создание n-грамм с учётом начала и конца."""
    tokens = [self.start_token] + tokens + [self.end_token]
    return [tuple(tokens[i:i + self.n_gram]) for i in range(len(tokens) - self.n_gram + 1)]
```

Дальше идет функция полного обучения моделей.

Метод train выполняет полное построение n-граммной модели на основе набора пар (текст $\rightarrow$)..

1. Токенизация данных и формирование общего корпуса:

   Первым шагом выполняется преобразование исходных пар в единый поток токенов. Для каждого текста и соответствующего ему названия выполняется разбиение на слова, после чего элементы объединяются в одну последовательность, разделённую специальным маркером <title>.

   ```
   all_tokens = []
   for text, title in pairs:
       text_tokens = tokenize(text)
       title_tokens = tokenize(title)
       combined = text_tokens + [self.title_token] + title_tokens
       all_tokens.extend(combined)
   ```

2. Формирование словаря токенов (vocabulary):

   После сбора всех токенов подсчитываются частоты слов, и формируется словарь модели. В него включаются только те слова, которые встретились более одного раза, что помогает уменьшить число редких элементов.

   ```
   token_counts = Counter(all_tokens)
   self.vocab = set(token for token, count in token_counts.items() if count > 1)
   self.vocab.update([self.start_token, self.title_token, self.end_token,
    ↪  self.unknown_token])
   ```

Слова, отсутствующие в словаре, будут заменяться на специальный токен <unk>.

3. Построение моделей меньшего порядка:

   Если основная модель использует n-граммы порядка n > 1, то автоматически создаются модели всех меньших порядков (1-граммная, 2-граммная и т.д.).

   Они будут применяться для линейной интерполяции или backoff-алгоритмов.

```
if self.n_gram > 1:
for n in range(1, self.n_gram):
    self.lower_order_models[n] = TitleNgramModel(n, self.smoothing, self.alpha)
    self.lower_order_models[n].train(pairs)
```

4. Подготовка данных и построение n-грамм:

   На следующем этапе текст и заголовок повторно токенизируются, но теперь с учетом словаря: редкие слова заменяются <unk>.

```
text_tokens = [token if token in self.vocab else self.unknown_token for token in
↪   text_tokens]
title_tokens = [token if token in self.vocab else self.unknown_token for token in
↪   title_tokens]
combined = text_tokens + [self.title_token] + title_tokens
ngrams_list = self.create_ngrams(combined)
```

5. Подсчёт частот контекстов и слов:

   Для каждой n-граммы фиксируется её контекст — первые n1 слов — и следующее слово. Модель накапливает статистику, необходимую для расчёта вероятностей.

```
for ngram in ngrams_list:
    context = ngram[:-1]
    next_word = ngram[-1]
    self.ngrams[context][next_word] += 1
    self.context_counts[context] += 1
```

   В результате модель получает полный набор частот, на которых далее строятся вероятности предсказания следующего слова.

Для удобства работы с датасетом реализован вспомогательный метод train_from_csv, который выполняет загрузку данных, их фильтрацию и передачу в основной тренировочный цикл модели. Метод обеспечивает удобную

интеграцию n-граммной модели с табличными корпусами.

На вход подаётся путь к CSV-файлу, содержащему пары (текст, название). Файл читается в формате pandas.DataFrame:

```
df = pd.read_csv(csv_path)
if limit:
    df = df.head(limit)
```

Параметр limit позволяет ограничить размер обучающей выборки, что полезно на этапе отладки.

Далее осуществляется проход по строкам таблицы.

Используется индикатор выполнения tqdm, который позволяет отслеживать сколько осталось времени до полного прохода.

```
pairs = []
for _, row in tqdm(df.iterrows(), total=len(df), desc="Обучение модели"):
    text = getattr(row, text_col, None)
    title = getattr(row, title_col, None)
    if text and title:
        pairs.append((text, title))
```

В каждой строке извлекаются значения из столбцов text и title (их имена можно менять через параметры функции).

Если в строке отсутствует одно из полей, пара пропускается.

В результате формируется список, полностью соответствующий тому формату данных, который использует основной метод обучения train.

После подготовки пар выполняется вызов базового алгоритма обучения.

В n-граммной модели ключевым этапом является оценка условной вероятности появления следующего слова при заданном контексте. В реализации предусмотрено несколько вариантов вычисления вероятностей: без сглаживания, со сглаживанием Лапласа и с линейной интерполяцией. Каждый из них решает разные проблемы статистической модели.

1. Простая вероятностная оценка (без сглаживания):

```
def get_simple_probability(self, context: tuple, word: str) -> float:
    context_counts = self.ngrams.get(context, {})
    total = sum(context_counts.values())
    return context_counts[word] / total if total else 0.0
```

Вероятность оценивается по формуле: $P(w_t \mid context) = \frac{C(context, w_t)}{C(context)}$ где

— $C(context, w_t)$ — число раз, когда после данного контекста появилось слово w_t,

— $C(context)$ — сколько раз этот контекст встречался в данных.

Особенности:

— Это самый точный вариант для часто встречающихся n-грамм.

— Главный недостаток — нулевые вероятности для комбинаций, которые не встречались в обучении. При генерации текста это приводит к «обрыву» предсказаний.

2. Сглаживание Лапласа (Add-one):

Для борьбы с нулевыми вероятностями используется метод Лапласовского сглаживания:

```
def get_laplace_probability(self, context: tuple, word: str) -> float:
    context_counts = self.ngrams.get(context, {})
    total = sum(context_counts.values())
    vocab_size = len(self.vocab)

    count_word = context_counts.get(word, 0)
    return (count_word + self.alpha) / (total + self.alpha * vocab_size)
```

Принцип метода:

Сглаживание Лапласа добавляет единицу к частоте каждого слова: $P(w_t \mid context) = \frac{C(context, w_t)+\alpha}{C(context)+\alpha \cdot |V|}$ где

— $\alpha = 1()$,

— |V| — размер словаря.

3. Линейная интерполяция (с моделями меньшего порядка):

Метод get_linear_interpolation_probability улучшает качество модели за счёт комбинирования вероятностей разных порядков:

```
def get_linear_interpolation_probability(self, context: tuple, word: str) -> float:
    if self.n_gram == 1 or not self.lower_order_models:
        return self.get_laplace_probability(context, word)

    lambda_current = 0.6
    lambda_backoff = 0.4

    current_prob = self.get_laplace_probability(context, word)
```

```
backoff_context = context[1:] if len(context) > 1 else tuple()
backoff_model = self.lower_order_models[self.n_gram - 1]
backoff_prob = backoff_model.get_probability(backoff_context, word)

return lambda_current * current_prob + lambda_backoff * backoff_prob
```

Математическая формула: $P = \lambda P_n + (1 - \lambda) P_{n-1}$, где

— P_n — вероятность в модели порядка n,

— P_{n-1} — вероятность в модели меньшего порядка (n1),

— λ — вес старшей модели.

Так как редкие контексты приводят к недостоверным оценкам. Интерполяция позволяет «страховаться» моделями меньшего порядка, которые более устойчивы статистически.

4. Универсальный метод выбора вероятности:

   Общий интерфейс get_probability выбирает метод в зависимости от параметров модели:

```
def get_probability(self, context: tuple, word: str) -> float:
    if word not in self.vocab:
        word = self.unknown_token

    if self.smoothing == 'laplace':
        return self.get_laplace_probability(context, word)
    elif self.smoothing == 'linear':
        return self.get_linear_interpolation_probability(context, word)
    elif self.smoothing is None:
        return self.get_simple_probability(context, word)
    else:
        return self.get_laplace_probability(context, word)
```

Далее идут методы для генерации названия.

Метод generate_with_backoff вычисляет распределение вероятностей слов для заданного контекста, используя стратегию постепенного уменьшения длины контекста.

Если полная n-грамма отсутствует в тренировочных данных, метод постепенно сокращает контекст, пока не найдёт подходящий.

Если подходящего контекста нет вовсе, используется равномерное распределение по словарю.

1. Инициализация контекста и предельной глубины отката:

   Сначала фиксируется исходный контекст и максимальная допустимая глубина backoff-уменьшения.

   Если глубина не указана вручную, она устанавливается равной n - 1, что позволяет модели последовательно пройти через все возможные контексты меньшего порядка.

```
if max_depth is None:
    max_depth = self.n_gram - 1
current_context = context
depth = 0
```

2. Проверка наличия контекста в модели:

   На каждом шаге выполняется проверка: встречался ли текущий контекст в обучающих данных. Если да — можно вычислять вероятности слов, следующих за этим контекстом.

```
if current_context in self.ngrams:
    probabilities = dict()
```

3. Вычисление вероятностей слов для текущего контекста:

   Для каждого слова из словаря, за исключением служебных токенов, вычисляется вероятность по выбранной стратегии сглаживания:

```
for word in self.vocab:
    if word not in [self.title_token, self.title_token, self.end_token]:
        probabilities[word] = self.get_probability(current_context, word)
```

4. Нормировка распределения:

   После расчёта суммируются все вероятности. Если сумма положительна, выполняется нормировка, чтобы получить корректное вероятностное распределение, сумма которого равна 1.

```
total_prob = sum(probabilities.values())
if total_prob > 0:
    return {word: prob / total_prob for word, prob in probabilities.items()}
```

5. При отсутствии данных — уменьшение контекста (backoff):

   Если ни один след за данным контекстом не зафиксирован, модель постепенно сокращает контекст, удаляя первый токен.

   Процесс продолжается, пока:

а) не будет найдено подходящее состояние;

б) или не будет достигнута максимальная глубина отката;

в) или контекст не станет однословным.

```
if len(current_context) > 1:
    current_context = current_context[1:]
else:
    break
depth += 1
```

6. Фоллбек: равномерное распределение по словарю:

   Если ни один контекст (ни меньшего, ни полного порядка) не встречался в корпусе, применяется равномерное распределение по всем доступным словам словаря, за исключением служебных токенов.

```
words = [w for w in self.vocab if w not in [self.start_token, self.title_token,
↪   self.end_token]]
return {word: 1.0 / len(words) for word in words}
```

Метод generate_title выполняет автоматическое порождение заголовка на основе входного текста.

Он использует вероятностную модель n-грамм, backoff-алгоритм и выбранный метод сглаживания.

Процесс генерации включает несколько последовательных этапов.

1. Токенизация входного текста и нормализация токенов:

   На первом шаге входная строка разбивается на токены. Каждый токен проверяется на принадлежность словарю модели: если токен отсутствует в словаре, он заменяется специальным маркером <unk>.

   После текста добавляется маркер <title>, обозначающий начало заголовка.

```
tokens = tokenize(input_text)
tokens = [token if token in self.vocab else self.unknown_token for token in
↪   tokens]
tokens += [self.title_token]
```

2. Формирование исходного контекста:

   Для старта генерации используется последние n-1 токенов исходной последовательности, куда входит маркер <title>.

   Это обеспечивает корректную условность предсказываемого слова: $P(w_t|\text{context})$.

3. Итеративное предсказание следующего слова:

   Генерация выполняется максимум в течение max_words шагов.

   На каждом шаге рассчитывается вероятностное распределение следующих слов с помощью backoff-механизма.

```
word_probs = self.generate_with_backoff(context)
if not word_probs:
    break
```

4. Нормировка распределения:

   Backoff возвращает ненормированное распределение.

   Сначала вычисляется сумма вероятностей, затем каждое значение нормируется так, чтобы сумма равнялась 1.

```
word_probs = {word: prob for word, prob in word_probs.items()}
total = sum(word_probs.values())
word_probs = {word: prob / total for word, prob in word_probs.items()}
```

5. Стохастический выбор следующего слова:

   Следующее слово выбирается случайно, пропорционально вероятностям.

   Используется стандартная выборка с весами.

```
words = list(word_probs.keys())
probs = list(word_probs.values())
next_word = random.choices(words, weights=probs)[0]
```

   Если модель сгенерировала маркеры </s> или <title>, процесс завершатся — это признаки конца последовательности.

```
if next_word in {self.end_token, self.title_token}:
    break
```

6. Обновление контекста для следующего шага:

   Добавленное слово включается в контекст, который смещается на один шаг вправо.

   Таким образом, в каждый момент времени используется актуальная n-1-грамма.

```
result.append(next_word)
context = (*context[1:], next_word) if len(context) > 0 else (next_word,)
```

7. Формирование итогового заголовка:

Полученные слова объединяются в строку. Если модель не смогла сгенерировать ни одного слова, возвращается fallback-значение «Без названия».

```
title = " ".join(result)
return title if title else "Без названия"
```

Для оценки качества работы модели, основанной на n-граммах, использовалась только одна метрика — METEOR, поскольку она наиболее подходит для оценки качества коротких текстов и учитывает как точные совпадения слов, так и семантическую близость.

Метод evaluate_meteor вычисляет среднее значение METEOR-метрики для заданного тестового набора.

Эта метрика позволяет количественно оценить качество генерации заголовков, сопоставляя с эталонными (reference) заголовками.

```
def evaluate_meteor(self, test_pairs: List[Tuple[str, str]]) -> float:
    """Вычисление средней METEOR-оценки для тестового набора (text,
    ↪  reference_title)."""
    scores = []
    for text, true_title in tqdm(test_pairs, desc="Оценка METEOR"):
        generated = self.generate_title(text)
        score = meteor_score([tokenize(true_title)], tokenize(generated))
        scores.append(score)
    return sum(scores) / len(scores) if scores else 0.0
```

Реализованный класс TitleNgramModel содержит методы save и load, которые позволяют сериализовать модель на диск и восстановить её из файла.

Для этого используется стандартный модуль Python — pickle.

```
def save(self, path: str):
    with open(path, "wb") as f:
    pickle.dump(self, f)

@staticmethod
def load(path: str):
    with open(path, "rb") as f:
        return pickle.load(f)
```

Для обучения n-граммной модели использовался заранее подготовленный набор данных, содержащий пары (текст + заголовок). Обучающая выборка была загружена из CSV-файла, после чего модель была обучена и сохранена для последующего использования.

Модель была создана с использованием триграмм (n = 3). Такой размер контекста позволяет учитывать два предыдущих слова при предсказании следующего.

После запуска процесса обучения был сформирован полный набор n-грамм и произведена оценка качества генерации на валидационном наборе.

В ходе обучения были получены следующие результаты:

1. Было обработано 821612 обучающих примеров.
2. По итогам обучения модель сформировала 11126709 уникальных контекстов.

После завершения обучения была проведена оценка качества генерации заголовков с использованием метрики METEOR на валидационном наборе объёмом 5963 примера. В результате средняя METEOR оценка составила 0.

Это показывает, что сгенерированные моделью заголовки практически не совпадают с референсными. Это ожидаемо для простой статистической модели.

Для демонстрации работы модели были выполнены запросы на генерацию названий для нескольких произвольных текстов. Как видно из примеров, модель формирует последовательности слов, которые не связаны по смыслу с содержанием текста. Это подтверждает ограниченность n-граммного подхода в задачах семантической генерации.

Пример 1

Исходный текст (фрагмент): «У меня большая семья из шести человек: я, мама, папа, старшая сестра, бабушка и дедушка. . . »

Сгенерированное название: «свиста ниппур апокалиптические каллиграфическими кастрированный благословление гвардейскую»

Пример 2

Исходный текст (фрагмент): «Я с детства хотел завести собаку, но родители мне не разрешали. . . »

Сгенерированное название: «местечковая постройнела просыпали унижениями расчерченные направляюттолкают многотысячными»

Пример 3

Исходный текст (фрагмент): «Когда я окончил университет, то начал ходить по собеседованиям в разные компании. . . »

Сгенерированное название: «запястий эмбриологии залишилась крючьев сочиняющие айви зацветающих»

Сгенерированные заголовки почти полностью состоят из редких или случайных слов, часто морфологически несовместимых. У них отсутствуют: смысловые связи, тематическая релевантность, грамматическая структура.

Поведение модели объясняется тем, что n-граммная модель:

1. Работает только на поверхностных статистических закономерностях;
2. Не понимает смысла текста;
3. Формирует заголовки по принципу вероятностного продолжения последовательностей, встретившихся в корпусе.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Techjury. How Much Data Is Created Every Day in 2024? [Электронный ресурс онлайн]. — [Б. м. : б. и.], 2024. — Режим доступа: https://techjury.net/blog/how-much-data-is-created-every-day/.

2 Шамигов, Ф.Ф. Автоматическая генерация новостных заголовков при помощи нейронной сети RuGPT-3 (влияние обучающего датасета на результативность модели) [Текст] / Шамигов, Ф.Ф. и Резанова, З.И. // Искусственный интеллект и цифровые коммуникации. — 2025. — Т. 4, № 1(13). — С. 62–70. — Поступила в редакцию: 06.11.2024; Принята к печати: 21.01.2025. Режим доступа: https://elibrary.ru/hfslfk.

3 Boczkowski, Pablo J. News comes across when I'm in a moment of leisure: Understanding the practices of incidental news consumption on social media [Текст] / Boczkowski, Pablo J., Mitchelstein, Eugenia и Matassi, María // New Media & Society. — 2018. — Т. 20, № 10. — С. 3523–3539.

4 Боковиков, С. А. Извлечение данных о погоде с помощью парсинга вебстраниц в Python [Текст] // Современные научные исследования и инновации. — 2024. — № 1 (153). — Студент 2 курса, факультет экономико-математический.

5 Jurafsky, Daniel. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition [Текст] / Jurafsky, Daniel и Martin, James H. — 2nd изд. — Upper Saddle River : Prentice Hall, 2024.

6 Вашкевич, Е.К. ТОКЕНЕЗАЦИЯ В NLP [Электронный ресурс онлайн]. — [Б. м. : б. и.], 2020. — Режим доступа: https://libeldoc.bsuir.by/bitstream/123456789/40257/1/Vashkevich_Tokenizatsiya.pdf.

7 Астапов, Р.Л. Автоматизированныя предобработка текста для определения эмоциональной окраски текста [Текст] / Астапов, Р.Л. и Мухамадеева, Р.М. //

Актуальные научные исследования в современном мире. — 2021. — № 5-2 (73). — С. 19–23.

8 BLEU: a Method for Automatic Evaluation of Machine Translation [Текст] / Papineni, Kishore, Roukos, Salim, Ward, Todd и Zhu, Wei-Jing. — [Б. м.] : Association for Computational Linguistics, 2002. — С. 311–318.

9 Jurafsky, Daniel. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models [Текст] / Jurafsky, Daniel и Martin, James H. — 3rd изд. — [Б. м. : б. и.], 2025. — Режим доступа: https://web.stanford.edu/~jurafsky/slp3/.

10 Lin, Chin-Yew. ROUGE: A Package for Automatic Evaluation of Summaries [Text] // Text Summarization Branches Out. — 2004. — P. 74–81.

11 Биджиева, С.Х. Анализ метрик оценки качества генеративных языковых моделей [Текст] / Биджиева, С.Х., Байрамукова, М.И. и Гриева, К.М. // Тенденции развития науки и образования. — 2024. — № 116-19. — С. 15–17.

12 Banerjee, Satanjeev. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments [Текст] / Banerjee, Satanjeev и Lavie, Alon. — 2005. — Июнь. — С. 65–72. — Режим доступа: https://www.aclweb.org/anthology/W05-0909.

13 Weizenbaum, Joseph. ELIZA – A Computer Program for the Study of Natural Language Communication between Man and Machine [Текст] // Communications of the ACM. — 1966. — Т. 9, № 1. — С. 36–45.

14 Colby, Kenneth. Simulation of Belief Systems [Текст] // Computer Models of Thought and Language. — 1972. — С. 251–286.

15 Winograd, Terry. Understanding Natural Language [Текст]. — New York : Academic Press, 1972.

16 Jurafsky, Daniel. Speech and Language Processing [Текст] / Jurafsky, Daniel и Martin, James H. — 1st изд. — [Б. м.] : Prentice Hall, 2000.

17 Katz, Slava M. Estimation of probabilities from sparse data for the language model component of a speech recognizer [Текст]. — 1987. — № TR-13-87.

18 Franz, Andreas. N-gram-based language models [Текст] / Franz, Andreas и Brants, Thorsten. — 2006. — С. 1234–1237.

19 Aiden, Erez. The Digital Humanities and the Future of the Book [Текст] / Aiden, Erez и Michel, Jean-Baptiste. — [Б. м. : б. и.], 2011. — С. 45–52.

20 Infini-gram: Scaling Unbounded n-gram Language Models to a Trillion Tokens [Текст] / Liu, Jiacheng, Min, Sewon, Zettlemoyer, Luke, Choi, Yejin и Hajishirzi, Hannaneh // arXiv preprint arXiv:2401.17377. — 2024. — jan. — License: CC BY 4.0. Режим доступа: https://arxiv.org/abs/2401.17377.

21 A Neural Probabilistic Language Model [Text] / Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Jauvin, Christian // Journal of Machine Learning Research. — 2003. — Vol. 3. — P. 1137–1155.

22 Москаленко, А.А. Разработка приложения веб-скрапинга с возможностями обхода блокировок [Текст] / Москаленко, А.А., Лапонина, О.Р. и Сухомлин, В.А. // Современные информационные технологии и ИТ-образование. — 2019. — Т. 15, № 2. — С. 413–420.

23 Shannon, Claude E. A Mathematical Theory of Communication [Текст] // Bell System Technical Journal. — 1948. — Т. 27, № 3. — С. 379–423.

24 Manning, Christopher D. Foundations of Statistical Natural Language Processing [Текст] / Manning, Christopher D. и Schütze, Hinrich. — Cambridge, MA : MIT Press, 1999.

25 Jurafsky, Daniel. Speech and Language Processing [Текст] / Jurafsky, Daniel и Martin, James H. — Upper Saddle River, NJ : Prentice Hall, 2000.

26 Towards Explainable Evaluation Metrics for Natural Language Generation [Текст] / Leiter, Christoph, Lertvittayakumjorn, Piyawat, Fomicheva, Marina, Zhao, Wei, Gao, Yang и Eger, Steffen. — 2022. — 03.

27 Zhang, Zhen. Language Model Perplexity Predicts Scientific Surprise and Transformative Impact [Text] / Zhang, Zhen and Evans, James // arXiv preprint. — 2025. — arXiv:2509.05591.

28 Li, Ming. What is Wrong with Perplexity for Long-context Language Modeling? [Text] / Li, Ming and Zhang, Wei // arXiv preprint. — 2024. — arXiv:2410.23771.

ПРИЛОЖЕНИЕ А

Парсинг

```
import time
import json
import re
from typing import Optional, Dict
from datetime import datetime, timedelta
from urllib.parse import urljoin
from tqdm import tqdm

import requests
from bs4 import BeautifulSoup
from fake_useragent import UserAgent


class ProzaRuParser:
    """Парсер для сайта proza.ru"""

    def __init__(self, base_url: str = "https://proza.ru/texts/list.html", delay: float = 1.5):
        self.base_url = base_url
        self.headers = {"User-Agent": UserAgent().random}
        self.timeout = 10
        self.delay = delay
        self.output_file = "../data/temp_data/json/data_proza_ru.json"

        with open(self.output_file, 'w', encoding='utf-8') as f:
            json.dump({}, f, ensure_ascii=False, indent=4)

    def _get_page(self, url: str) -> Optional[BeautifulSoup]:
        """Загружает страницу и возвращает BeautifulSoup-объект."""
        try:
            print(f"Загружается: {url}")
            time.sleep(self.delay)
            response = requests.get(url, headers=self.headers, timeout=self.timeout)
            response.raise_for_status()
            return BeautifulSoup(response.content, "html.parser")
        except requests.exceptions.RequestException as e:
            print(f"Ошибка при загрузке {url}: {e}")
            return None

    def get_all_forms(self) -> Dict[str, str]:
        """Получает названия и ссылки на разделы с малыми формами."""
        soup = self._get_page(self.base_url)
        if not soup:
            return {}

        works_block = soup.find('ul', attrs={'type': 'square', 'style': 'color:#404040'})
        all_forms = works_block.find_all('ul', attrs={'type': 'square'})
        data_all_forms = {}
```

```
48          for form in all_forms:
49              category = form.find_all('a')
50              for link in category:
51                  title = link.text.strip()
52                  full_link = "https://www.proza.ru" + link['href']
53                  data_all_forms[title] = full_link
54
55          return data_all_forms
56
57      def get_text(self, url: str) -> str:
58          soup = self._get_page(url)
59          if not soup:
60              return ""
61
62          text = soup.find('div', attrs={'class': 'text'})
63          if not text:
64              return ""
65
66          return self.clean_text(str(text))
67
68      def clean_text(self, html: str) -> str:
69          soup = BeautifulSoup(html, 'html.parser')
70          for element in soup(['iframe', 'img', 'script', 'style',
71                               'div.video-blk', 'div.video-block',
72                               'div.ads', 'div.advertisement']):
73              element.decompose()
74
75          for div in soup.find_all('div', class_=lambda x: x and 'hidden' in x):
76              div.decompose()
77
78          clean_text = soup.get_text(separator='\n', strip=True)
79          lines = [line.strip() for line in clean_text.split('\n') if line.strip()]
80          return '\n'.join(lines)
81
82      def get_works(self, url: str, category_title: str) -> Dict[str, Dict[str, str]]:
83          """
84
85          :param url:
86          :param category_title:
87          :return:
88          """
89          soup = self._get_page(url)
90          if not soup:
91              return {}
92
93          works_block = soup.find_all('ul', attrs={'type': 'square', 'style':
            ↪  'color:#404040'})
94          data_small_works = {}
95
96          for works_list in works_block:
97              for work in works_list.find_all('li'):
98                  work_data = work.find('a')
99                  if not work_data:
```

```
100                     continue
101                 try:
102                     author = work.find('a', attrs={'class': 'poemlink'}).text.strip()
103                     title = work_data.text.strip()
104                     link = "https://www.proza.ru" + work_data['href']
105                     text = self.get_text(link)
106 
107                     data_small_works[author] = {
108                         'link': link,
109                         'title': title,
110                         'text': text
111                     }
112                     self._update_output_file(category_title, title, data_small_works[title])
113                     time.sleep(self.delay)
114                 except Exception as e:
115                     print(f"Ошибка при обработке произведения: {e}")
116                     continue
117 
118         return data_small_works
119 
120     def parse_by_dates(self, start_date: str, end_date: str, category_title: str, topic:
        ↪  str) -> Dict[str, dict]:
121         """
122         Парсит материалы за указанный период в обратном порядке
123         :param category_title:
124         :param start_date: Дата начала в формате 'YYYY-MM-DD'
125         :param end_date: Дата окончания в формате 'YYYY-MM-DD'
126         :param topic: ID темы
127         :return: Словарь с данными
128         """
129         current_date = datetime.strptime(start_date, "%Y-%m-%d")
130         end_date = datetime.strptime(end_date, "%Y-%m-%d")
131         result = {}
132 
133         while current_date >= end_date:
134             date_str = current_date.strftime("%Y-%m-%d")
135             print(f"\nОбработка даты: {date_str}")
136 
137             day = current_date.strftime("%d")
138             month = current_date.strftime("%m")
139             year = current_date.strftime("%Y")
140 
141             url = f"{self.base_url}?day={day}&month={month}&year={year}&topic={
                ↪  topic}"
142             data_day = self.get_works(url, category_title)
143             result.update(data_day)
144 
145             current_date -= timedelta(days=1)
146 
147         return result
148 
149     def _update_output_file(self, category: str, title: str, work_data: dict):
150         """Обновляет JSON-файл, добавляя новое произведение"""
```

```
151         try:
152             with open(self.output_file, 'r', encoding='utf-8') as f:
153                 existing_data = json.load(f)
154 
155             if category not in existing_data:
156                 existing_data[category] = {}
157             existing_data[category][title] = work_data
158 
159             with open(self.output_file, 'w', encoding='utf-8') as f:
160                 json.dump(existing_data, f, ensure_ascii=False, indent=4)
161 
162         except Exception as e:
163             print(f" Ошибка при обновлении файла: {e}")
164 
165     def get_all_work(self) -> Dict[str, Dict[str, Dict[str, str]]]:
166         """Получает все малые формы и произведения внутри них."""
167         all_data = {}
168         all_forms = self.get_all_forms()
169 
170         for all_form_title, all_form_link in all_forms.items():
171             print(f"\nОбработка категории: {all_form_title}")
172             works = self.get_works(all_form_link, all_form_title)
173             topic = re.search(r'topic=(\d+)', all_form_link).group(1)
174             works_by_data = self.parse_by_dates("2025-07-18", "2025-07-11",
                ↪  all_form_title, topic)
175             merged_works = {**works, **works_by_data}
176             all_data[all_form_title] = merged_works
177             time.sleep(self.delay)
178 
179         return all_data
180 
181     def save_to_json(self, data: dict, filename: str = "data/data_proza_ru.json"):
182         """Сохраняет данные в JSON-файл."""
183         try:
184             with open(filename, 'w', encoding='utf-8') as f:
185                 json.dump(data, f, ensure_ascii=False, indent=4)
186             print(f"\n Данные сохранены в {filename}")
187         except Exception as e:
188             print(f" Ошибка при сохранении JSON: {e}")
189 
190 
191 parser = ProzaRuParser()
192 parser.get_all_work()
```

```
1 import random
2 import time
3 import json
4 from typing import Optional, Dict
5 
6 from tqdm import tqdm
7 
8 import requests
9 from bs4 import BeautifulSoup
```

```
10  from fake_useragent import UserAgent
11
12  link = "https://briefly.ru/cultures/"
13
14
15  class ParsingBriefly():
16      """Парсинг сайта Брифли с краткими пересказами произведений"""
17
18      def __init__(self, url: str):
19          self.briefly_url = "https://briefly.ru"
20          self.base_url = url
21          self.headers = {"User-Agent": UserAgent().random}
22          self.timeout = 5
23          self.filename = "../data/temp_data/json/briefly.json"
24
25      def _get_page(self, url: str) -> Optional[BeautifulSoup]:
26          """Загружает страницу и возвращает BeautifulSoup-объект."""
27          try:
28              self.human_delay()
29              response = requests.get(url, headers=self.headers, timeout=self.timeout)
30              response.raise_for_status()
31              return BeautifulSoup(response.content, "html.parser")
32          except requests.exceptions.RequestException as e:
33              print(f"Ошибка при загрузке {url}: {e}")
34              return None
35
36      def get_authors(self, url: str, name_culture: str = None):
37          """Загружает всех авторов и их произведения"""
38          soup = self._get_page(url)
39          all_data = {}
40          if soup is None:
41              return None
42
43          alphabet_index = soup.find("div", class_="index alphabet")
44          if alphabet_index is None:
45              return None
46
47          authors = alphabet_index.find_all("a", class_="author")
48          for author in tqdm(authors, postfix=name_culture):
49              link = self.briefly_url + author.get("href")
50              author_data = self.get_works(link)
51              if author_data:  # защита
52                  all_data.update(author_data)
53
54          return all_data
55
56      def get_works(self, url: str):
57          """Загружает произведения автора"""
58          soup = self._get_page(url)
59          if soup is None:
60              return {}
61
62          works_data = {}
```

```
        try:
            section = soup.find("section", class_="works_index")

            if section is None:
                section = soup.find("section", class_="author_works")
                if section is None:
                    return works_data

                works = section.find_all("div", class_="w-featured")
                for work in works:
                    title_el = work.find("div", class_="w-title")
                    link_el = work.find("a")
                    if not title_el or not link_el:
                        continue

                    title = title_el.text.strip()
                    link = self.briefly_url + link_el.get("href")
                    if any(word in title.lower() for word in ["глава", "том", "действие"]):
                        continue

                    text = self.get_text(link, category="published")
                    works_data[title] = text
                    self.human_delay(base=1.5, var=1.0)
            else:
                full_retelling = section.find_all("li", class_="published")
                small_retelling = section.find_all("li", class_="pending")

                for work in small_retelling:
                    work = work.find("a", class_="title")
                    if not work:
                        continue
                    title = work.text.strip()
                    link = work.get("href")
                    if any(word in title.lower() for word in ["глава", "том", "действие"]):
                        continue
                    text = self.get_text(link, category="pending")
                    works_data[title] = text
                    self.human_delay(base=1.5, var=1.0)

                for work in full_retelling:
                    work = work.find("a", class_="title")
                    if not work:
                        continue
                    title = work.text.strip()
                    link = self.briefly_url + work.get("href")
                    if any(word in title.lower() for word in ["глава", "том", "действие"]):
                        continue
                    text = self.get_text(link, category="published")
                    works_data[title] = text
                    self.human_delay(base=1.5, var=1.0)

            return works_data
        except Exception as e:
```

```
116             print(f"Ошибка при парсинге {url}: {e}")
117             return {}
118
119     def get_text(self, url: str, category: str):
120         """Получает текст произведения"""
121         soup = self._get_page(url)
122         if soup is None:
123             return "Описание отсутствует"
124
125         try:
126             if category == "pending":
127                 element = soup.find("div", class_="microsummary__content")
128             else:
129                 element = soup.find("p", class_="microsummary__content")
130
131             if element:
132                 text = element.get_text(strip=True)
133             else:
134                 main_div = soup.find("div", id="text")
135                 if main_div:
136                     for ad in main_div.find_all("div", class_="honey"):
137                         ad.decompose()
138                     paragraphs = [p.get_text(" ", strip=True) for p in
                        ↪  main_div.find_all("p")]
139                     text = " ".join(paragraphs)
140                 else:
141                     text = ""
142             return text if text else "Описание отсутствует"
143         except Exception as e:
144             print(f"Ошибка в get_text {url}: {e}")
145             return "Описание отсутствует"
146
147     def get_all_data(self, url: str):
148         all_data = self._load_existing_data()
149         soup = self._get_page(url)
150         try:
151             cultures_cards = soup.find_all("a", class_="visited-hidden")
152             for culture in cultures_cards[7:]:
153                 name = culture.get_text(strip=True)
154                 link = self.briefly_url + culture.get("href")
155                 data_culture = self.get_authors(link, name)
156                 if data_culture:
157                     all_data.update(data_culture)
158
159                     self.save_to_json(all_data)
160                     self.human_delay(base=2, var=1.5)
161         except Exception as e:
162             print(e)
163
164     def save_to_json(self, data: Dict, filename: str = None) -> None:
165         """Сохраняет данные в JSON-файл."""
166         filename = filename or self.filename
167         with open(filename, "w", encoding="utf-8") as f:
```

```
            json.dump(data, f, ensure_ascii=False, indent=4)

    def _load_existing_data(self) -> Dict:
        """Загружает уже сохраненные данные, чтобы дописывать новые."""
        try:
            with open(self.filename, "r", encoding="utf-8") as f:
                return json.load(f)
        except (FileNotFoundError, json.JSONDecodeError):
            return {}

    @staticmethod
    def human_delay(base: float = 1.5, var: float = 1.0, long_pause_prob: float = 0.05):
        """Делает более человеческие задержки"""
        delay = random.uniform(base, base + var)
        time.sleep(delay)

        if random.random() < long_pause_prob:
            long_delay = random.uniform(5, 15)
            time.sleep(long_delay)


parser = ParsingBriefly(link)
parser.get_all_data(link)
```

```
import time
import json
from typing import Optional, Dict
from urllib.parse import urljoin
from tqdm import tqdm

import requests
from bs4 import BeautifulSoup
from fake_useragent import UserAgent


class LitPrichalParser:
    """Парсер сайта ЛитПричал"""

    def __init__(self, base_url: str = "https://www.litprichal.ru"):
        self.base_url = base_url
        self.headers = {"User-Agent": UserAgent().random}
        self.timeout = 10

    def _get_page(self, url: str) -> Optional[BeautifulSoup]:
        """Загружает страницу и возвращает BeautifulSoup-объект."""
        try:
            time.sleep(1)
            response = requests.get(url, headers=self.headers, timeout=self.timeout)
            response.raise_for_status()
            return BeautifulSoup(response.content, "html.parser")
        except requests.exceptions.RequestException as e:
            print(f"Ошибка при загрузке {url}: {e}")
            return None
```

```
30
31      def get_genres(self) -> Dict[str, Dict[str, str]]:
32          """Парсит список жанров с главной страницы."""
33          url = f"{self.base_url}/prose.php"
34          soup = self._get_page(url)
35          if not soup:
36              return {}
37
38          genres = {}
39          genre_blocks = soup.find_all("div", class_="col-sm-6 col-md-4")
40
41          print("Парсинг жанров:")
42          for block in tqdm(genre_blocks, desc="Жанры"):
43              for genre_link in block.find_all("a"):
44                  name = genre_link.text.strip()
45                  link = urljoin(self.base_url, genre_link.get("href"))
46                  genres[name] = {"link": link}
47
48          return genres
49
50      def _get_page_count(self, genre_url: str) -> int:
51          """Определяет количество страниц в жанре."""
52          soup = self._get_page(genre_url)
53          if not soup:
54              return 1
55
56          pagination = soup.find("ul", class_="pagination")
57          if not pagination:
58              return 1
59
60          pages = pagination.find_all("li")
61          if not pages:
62              return 1
63
64          try:
65              last_page = int(pages[-1].find("a").get("href")
66                              .strip("/").split("/")[-1]
67                              .replace("p", ""))
68              return last_page
69          except (ValueError, IndexError):
70              return 1
71
72      def get_books(self, genre_url: str) -> Dict[str, Dict[str, str]]:
73          """Парсит книги из указанного жанра с учетом пагинации.
74
75          Args:
76              genre_url: URL страницы жанра
77          """
78          total_pages = self._get_page_count(genre_url)
79
80          books_data = {}
81
82          print(f"\nПарсинг книг в жанре {genre_url} (всего страниц: {total_pages}):")
```

```
83
84          for page in range(1, total_pages + 1):
85              page_url = f"{genre_url}/{f'p{str(page)}'}" if page > 1 else genre_url
86              soup = self._get_page(page_url)
87              if not soup:
88                  continue
89
90              books = soup.find_all("div", class_="col-md-6 x2")
91
92              for book in tqdm(books, desc=f"Страница {page}/{total_pages}"):
93                  try:
94                      title = book.find("a", class_="bigList").text.strip()
95                      link = self.base_url + book.find("a", class_="bigList").get("href")
96                      author = book.find("a", class_="forum").text.strip()
97
98                      if author not in books_data:
99                          text = self.get_text(link)
100                         books_data[author] = {
101                             "link": link,
102                             "title": title,
103                             "text": text,
104                             "genre_url": genre_url
105                         }
106
107                 except Exception as e:
108                     print(f"\nОшибка при парсинге книги: {e}")
109                     continue
110
111             time.sleep(1.5)
112
113         return books_data
114
115     def get_text(self, url: str) -> str:
116         """Возвращает текст"""
117         time.sleep(0.5)
118         soup = self._get_page(url)
119         if not soup:
120             return ""
121
122         text_blocks = soup.find_all("div", class_="col-md-12 x2")
123         return self.clean_text(str(text_blocks[1]))
124
125     def clean_text(self, html: str) -> str:
126         """Очищает HTML от ненужных элементов и возвращает чистый текст"""
127         soup = BeautifulSoup(html, 'html.parser')
128
129         for element in soup(['iframe', 'img', 'script', 'style',
130                              'div.video-blk', 'div.video-block',
131                              'div.ads', 'div.advertisement']):
132             element.decompose()
133
134         for div in soup.find_all('div', class_=lambda x: x and 'hidden' in x):
135             div.decompose()
```

```
136         clean_text = soup.get_text(separator='\n', strip=True)
137         lines = []
138         for line in clean_text.split('\n'):
139             line = line.strip()
140             if line:
141                 lines.append(line)
142 
143         final_text = '\n'.join(lines)
144 
145         return final_text
146 
147     def save_to_json(self, data: Dict, filename: str =
        ↪  "data/парсинг/data_litprichal.json") -> None:
148         """Сохраняет данные в JSON-файл."""
149         with open(filename, "w", encoding="utf-8") as f:
150             json.dump(data, f, ensure_ascii=False, indent=4)
151         print(f'\nДанные сохранены в {filename}')
152 
153     def parse_all_in_genre(self) -> Dict[str, Dict]:
154         """Парсит все жанры и книги в них."""
155         genres = self.get_genres()
156         result = {}
157         print("\nПарсинг книг по всем жанрам:")
158         for genre_name, genre_data in tqdm(genres.items(), desc="Общий прогресс"):
159             books = self.get_books(genre_data["link"])
160             result[genre_name] = books
161             time.sleep(10)
162         return genres
163 
164 
165 parser = LitPrichalParser()
166 books_data = parser.parse_all_in_genre()
167 parser.save_to_json(books_data)
```

```
1  import time
2  import json
3  from typing import Optional, Dict
4  from tqdm import tqdm
5  
6  import requests
7  from bs4 import BeautifulSoup
8  from fake_useragent import UserAgent
9  
10 link = "https://www.litres.ru/genre/klassicheskaya-literatura-5028/"
11 
12 
13 class LitresParser:
14     """Парсер сайта ЛитРес"""
15 
16     def __init__(self, url: str):
17         self.litres_url = "https://www.litres.ru"
18         self.base_url = url
19         self.headers = {"User-Agent": UserAgent().random}
```

```
20          self.timeout = 10
21          self.filename = "/data/temp_data/litres.json"
22
23      def _get_page(self, url: str) -> Optional[BeautifulSoup]:
24          """Загружает страницу и возвращает BeautifulSoup-объект."""
25          try:
26              time.sleep(1.5)
27              response = requests.get(url, headers=self.headers, timeout=self.timeout)
28              response.raise_for_status()
29              return BeautifulSoup(response.content, "html.parser")
30          except requests.exceptions.RequestException as e:
31              print(f"Ошибка при загрузке {url}: {e}")
32              return None
33
34      def get_books(self, url: str, pages: int = 800):
35          """Парсит книги с учетом пагинации и сохраняет после каждой страницы."""
36          books_data = self._load_existing_data()
37
38          for page in tqdm(range(1, pages + 1)):
39              page_url = f"{url}?page={page}"
40              soup = self._get_page(page_url)
41              if not soup:
42                  continue
43
44              books = soup.find_all(
45                  "div", class_="Art-module__3wrtfG__content
                    ↪  Art-module__3wrtfG__content_full"
46              )
47              time.sleep(1)
48
49              for book in books:
50                  try:
51                      info = book.find("a", class_="ArtInfo-module__Y-DtKG__title")
52                      title = info.text.strip()
53                      link = self.litres_url + info.get("href")
54
55                      if title in books_data:
56                          continue
57
58                      text = self.get_text(link)
59                      books_data[title] = text
60
61                      time.sleep(0.5)
62
63                  except Exception as e:
64                      print(f"\nОшибка при парсинге книги: {e}")
65                      continue
66
67              self.save_to_json(books_data)
68              time.sleep(2)
69
70          return books_data
71
```

```
72      def get_text(self, url: str):
73          """Получает текст - аннотацию"""
74          soup = self._get_page(url)
75          if not soup:
76              return None
77
78          block = soup.find("div",
            ↪  class_="BookDetailsAbout-module__p8ABVW__truncate")
79          if not block:
80              return None
81
82          truncated = block.find("div",
            ↪  class_="Truncate-module__FwxwPG__truncated")
83          return truncated.text if truncated else None
84
85      def save_to_json(self, data: Dict, filename: str = None) -> None:
86          """Сохраняет данные в JSON-файл."""
87          filename = filename or self.filename
88          with open(filename, "w", encoding="utf-8") as f:
89              json.dump(data, f, ensure_ascii=False, indent=4)
90
91      def _load_existing_data(self) -> Dict:
92          """Загружает уже сохраненные данные, чтобы дописывать новые."""
93          try:
94              with open(self.filename, "r", encoding="utf-8") as f:
95                  return json.load(f)
96          except (FileNotFoundError, json.JSONDecodeError):
97              return {}
98
99
100 parser = LitresParser(link)
101 books = parser.get_books(link)
```

ПРИЛОЖЕНИЕ Б

Работа с данными

```
#%% md
# # Импорт библиотек
#%%
import json
import re

import pandas as pd

import matplotlib.pyplot as plt
from pandas import DataFrame

from wordcloud import WordCloud

#%% md
# # Объединим все данные в один датасет
#%%
def open_json(input_file: str) -> dict:
    with open(input_file) as json_file:
        data = json.load(json_file)
    return data
#%%
def json_to_csv_lp(json_data: dict) -> DataFrame:
    rows = []
    for category, works in json_data.items():
        for author, work_data in works.items():
            rows.append({
                "title": work_data.get("title", ""),
                "text": work_data.get("text", "")
            })

    df = pd.DataFrame(rows)
    return df
#%%
def json_to_csv(json_data: dict) -> DataFrame:
    rows = []
    for title, text in json_data.items():
        rows.append({
            "title": title,
            "text": text
        })

    df = pd.DataFrame(rows)
    return df
#%% md
# clean_data
#%%
briefly = open_json("../data/temp_data/json/briefly.json")
litprichal = open_json("../data/temp_data/json/data_litprichal.json")
proza_ru = open_json("../data/temp_data/json/data_proza_ru.json")
```

```
50  litres = open_json("../data/temp_data/json/litres.json")
51  #%%
52  briefly = json_to_csv(briefly)
53  litprichal = json_to_csv_lp(litprichal)
54  proza_ru = json_to_csv_lp(proza_ru)
55  litres = json_to_csv(litres)
56  #%%
57  data = pd.concat([briefly, litprichal, proza_ru, litres], ignore_index=True)
58  #%%
59  data
60  #%%
61  len(briefly) + len(litprichal) + len(proza_ru) + len(litres) - len(data)
62  #%% md
63  # Никакие данные не потерялись
64  #%% md
65  # # Работа с данными
66  #%%
67  data = data.apply(lambda col: col.str.lower().str.strip())
68  #%% md
69  # ## Почистим данные от всего лишнего
70  #%% md
71  # ### Удалим лишние символы
72  #%% md
73  # Удалим пропущенные значения
74  #%%
75  data.isna().sum()
76  #%%
77  data = data.dropna().reset_index(drop=True)
78  #%%
79  def clean_text(text):
80      """Более аккуратная очистка текста для генерации заголовков"""
81      if pd.isna(text):
82          return ""
83      text = str(text).lower()
84      text = re.sub(r"http\S+", "", text)
85      text = re.sub(r"<[^>]+>", "", text)
86      text = re.sub(r"[^\w\s,.!?-]", " ", text)
87      text = re.sub(r"\s+", " ", text).strip()
88      return text
89  #%%
90  data.text = data.text.apply(clean_text)
91  #%% md
92  # ## Уберем из названий нумерацию (том, эпизод и т.п.)
93  #%%
94  def clean_title_completely(title):
95      """
96      Полная очистка названия от всех видов нумерации
97      """
98      if pd.isna(title):
99          return title
100
101     cleaned = str(title).lower().strip()
102
```

```
103     complex_patterns = [
104         r'\(?\s*\d+\s+(?:эпизод|серия|глава|часть)\s+\d+\s+(?:том|книга|т\.)\s*\d
            ↪  *\s*\.?\)?',
105         r'\(?\s*\d+\s+(?:том|книга|т\.)\s+\d+\s+(?:эпизод|серия|глава|часть)\s*\d
            ↪  *\s*\.?\)?',
106         r'\b\d+\s+\d+\s+(?:том|часть|книга|эпизод|серия)\b',
107         r'\b(?:том|часть|книга|эпизод|серия)\s+\d+\s+\d+\b',
108         r'\b\d+[-\.,]\s*\d+\s+(?:том|часть|книга)',
109         r'\b(?:том|часть|книга)\s+\d+[-\.,]\s*\d+\b',
110     ]
111
112     for pattern in complex_patterns:
113         cleaned = re.sub(pattern, '', cleaned, flags=re.IGNORECASE)
114
115     basic_patterns = [
116       r'\s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)\s*[ivxlcdm0-9]+',
117       r'\s*[ivxlcdm0-9]+\s*(?:том|часть|книга|т\.|vol\.?|эпизод|серия|глава|выпуск)',
118         r'\s*\d+[-\.,]?\s*(?:том|часть|книга|глава)',
119         r'\s*(?:том|часть|книга|глава)[-\.,]?\s*\d+',
120     ]
121     for pattern in basic_patterns:
122         cleaned = re.sub(pattern, '', cleaned, flags=re.IGNORECASE)
123
124     number_words = [
125         "первая", "вторая", "третья", "четвертая", "пятая", "шестая", "седьмая",
126         "восьмая", "девятая", "десятая", "одиннадцатая", "двенадцатая",
            ↪  "тринадцатая",
127         "четырнадцатая", "пятнадцатая", "шестнадцатая", "семнадцатая",
            ↪  "восемнадцатая",
128         "девятнадцатая", "двадцатая"
129     ]
130     text_num_pattern = r'\b(?:глава|часть|том|эпизод|серия|книга|выпуск)\s+(?:'
        ↪  + "|".join(number_words) + r')\b'
131     cleaned = re.sub(text_num_pattern, '', cleaned, flags=re.IGNORECASE)
132
133     cleaned = re.sub(
134         r'\b\d+[-–]?(?:я|й|е|ой|ая|ое|ые|ых)?\s+(?:глава|часть|том|книга|серия|эпизо
            ↪  д|выпуск)\b',
135         '', cleaned, flags=re.IGNORECASE
136     )
137     cleaned = re.sub(
138         r'\b(?:глава|часть|том|книга|серия|эпизод|выпуск)\s+\d+[-–]?(?:я|й|е|ая|ое|ы
            ↪  е|ых)?\b',
139         '', cleaned, flags=re.IGNORECASE
140     )
141
142     cleaned = re.sub(r'№', '', cleaned)
143
144     cleaned = re.sub(r'\([^)]*\d+[^)]*\)', '', cleaned)
145     cleaned = re.sub(r'\s+\d+\s*\.?$', '', cleaned)
146     cleaned = re.sub(r'^\d+\s+', '', cleaned)
147
148     cleaned = re.sub(r'[^\w\s.,!?-]', '', cleaned)
```

```
149     cleaned = re.sub(r '\s+ ', ' ', cleaned)
150     cleaned = cleaned.strip( ' ,.- ')
151 
152     return cleaned
153 #%%
154 data["cleaned_title"] = data.title.apply(clean_title_completely)
155 #%%
156 changed_count = (data.title != data.cleaned_title).sum()
157 print(f"Изменено названий: {changed_count} из {len(data)}")
158 #%%
159 data.describe()
160 #%% md
161 # почему то есть полностью пустые тексты
162 #%%
163 data.isna().sum()
164 #%%
165 data = data[data.text != ""].reset_index(drop=True)
166 data = data[data.title != ""].reset_index(drop=True)
167 #%%
168 data.describe()
169 #%% md
170 # есть пустые тексты с заглушкой «описание отсутствует», удалим их
171 #%%
172 data = data[data.text != "описание отсутствует"].reset_index(drop=True)
173 #%%
174 data.describe()
175 #%%
176 data.cleaned_title.value_counts().head(20)
177 #%% md
178 # Почистим строки, в которых есть не русские буквы
179 #%%
180 def keep_only_russian(text):
181     if pd.isna(text):
182         return ""
183     text = re.sub(r"[^А-Яа-яЁё0-9\s.,!?-]", "", text)
184     text = re.sub(r"\s+", " ", text).strip()
185     return text
186 
187 data.text = data.text.apply(keep_only_russian)
188 data.cleaned_title = data.cleaned_title.apply(keep_only_russian)
189 
190 data = data[(data.text != "") & (data.cleaned_title != "")].reset_index(drop=True)
191 
192 #%% md
193 # Заметим, что в названиях часто встречаются пустые названия или состоящие
    ↪  только из символов.
194 #%%
195 data = data[data.cleaned_title.str.strip() != ""]
196 noise_titles = ["***"]
197 data = data[~data.cleaned_title.isin(noise_titles)].reset_index(drop=True)
198 
199 print(data.cleaned_title.value_counts().head(20))
200 print(f"Всего примеров после очистки: {len(data)}")
```

```
#%% md
# ## Почистим строки, в которых нет текста и удалим лидирующую пунктуацию
#%%
def is_meaningful(text):
    return bool(re.search(r"[А-Яа-яA-Za-z0-9]", text))

def clean_leading_punct(text):
    return re.sub(r"^[^\wА-Яа-я0-9]+", "", text).strip()
#%%
data = data[data.text.apply(is_meaningful)].reset_index(drop=True)
data.text = data.text.str.replace(r"[^\w\s,.!?-]", " ", regex=True)
data.text = data.text.str.replace(r"\s+", " ", regex=True).str.strip()
data.describe()
#%% md
# ## Посмотрим на дубликаты
#%%
data.duplicated().sum()
#%%
data.text.duplicated().sum(), data.title.duplicated().sum()
#%% md
# Заметим, что есть дубликаты в названиях, но это нормально, так как парсились разные сайты, на которых могли быть одни и те же произведения. Такое можно оставить.
#
# Но есть дубликаты в тексте, причем большинство из них такие, что текст одинаковый, а названия разные. Такое нужно удалять, это создаст лишний шум для модели.
#%%
data = data[~data.text.duplicated(keep=False)].reset_index(drop=True)
#%%
data.duplicated().sum()
#%%
data.describe()
#%%
data.drop(columns=["title"], inplace=True)
#%%
data.rename(columns={"cleaned_title": "title"}, inplace=True)
#%%
data.to_csv("../data/temp_data/all_data_cleaned.csv", index=False)
#%% md
# # Анализ длины слов в названиях
#%%
title_counts = data.title.value_counts()
title_counts.head(10)
#%%
data["title_len"] = data.title.apply(lambda x: len(str(x).split()))

length_counts = data.groupby('title_len').size()

plt.figure(figsize=(12,6))
length_counts.plot(kind='bar')
plt.xlabel("Длина названия (слов)")
plt.ylabel("Количество названий")
```

```
250 plt.title("Распределение длин названий в датасете без аугментации")
251 plt.show()
252 #%% md
253 # Заметно, что есть названия с большим количеством слов. Это внесет в модель
    ↪ шум, поэтому стоит удалить их.
254 #%%
255 data.describe(include="O")
256 #%%
257 data = data[data.title_len <= 10].reset_index(drop=True)
258 #%%
259 data.describe(include="O")
260 #%%
261 data['text_len'] = data.text.str.split().apply(len)
262 #%%
263 print("Всего примеров:", len(data))
264 print("Уникальные заголовки:", data.title.nunique())
265 print("Средняя длина текста:", data.text_len.mean())
266 print("Средняя длина заголовка:", data.title_len.mean())
267 #%%
268 data.head()
269 #%%
270 data.drop(columns=["title_len", "text_len"], inplace=True)
271 #%%
272 data.to_csv("../data/all_data.csv", index=False)
273 #%% md
274 # # Посмотрим на облако слов
275 #%%
276 all_titles = " ".join(data.title.astype(str))
277
278 wordcloud = WordCloud(
279     width=1200, height=600,
280     background_color="white",
281     max_words=400,
282     colormap="viridis",
283     collocations=False,
284 ).generate(all_titles)
285 #%%
286 plt.figure(figsize=(12, 6))
287 plt.imshow(wordcloud, interpolation='bilinear')
288 plt.axis("off")
289 plt.title("Облако слов по заголовкам", fontsize=16)
290 plt.show()
291 #%% md
292 # Заметно, что самые популярные слова — предлоги, союзы и т.п., что логично. Их
    ↪ можно убрать, но оставим, чтобы модель училась генерировать названия,
    ↪ приближенные к реальности.
```

```
1 # %% md
2 # # Импорт библиотек
3 # %%
4 import re
5
6 import pandas as pd
```

```
import nltk
import plotly.graph_objects as go
from sklearn.model_selection import train_test_split
import numpy as np

from tqdm import tqdm

nltk.download("punkt")

from nltk.tokenize import sent_tokenize

# %%
data = pd.read_csv("../data/all_data.csv")
# %%
data.describe()
# %% md
# # Разделение данных
# %%
train_df, val_df = train_test_split(data, test_size=0.2, random_state=42)
# %%
len(train_df), len(val_df)
# %%
val_df.describe()


# %% md
# Заметим, что в валидацию попали несколько текстов с одинаковыми названиями. Это не очень хорошо, так как может исказить результаты оценки модели.
#
# Переделаем разделение.
# %%
def split_with_controlled_test_size(data, target_test_size=0.2, random_state=42):
    np.random.seed(random_state)

    title_groups = data.groupby('title').apply(lambda x: x.index.tolist()).to_dict()

    unique_titles = list(title_groups.keys())
    np.random.shuffle(unique_titles)

    train_indices = []
    test_indices = []

    target_test_count = int(len(data) * target_test_size)

    for title in unique_titles:
        indices = title_groups[title]

        if len(test_indices) < target_test_count:
            test_idx = np.random.choice(indices, 1)[0]
            test_indices.append(test_idx)
            train_indices.extend([idx for idx in indices if idx != test_idx])
        else:
            train_indices.extend(indices)
```

```
59
60      return data.iloc[train_indices], data.iloc[test_indices]
61
62
63  train_df, val_df = split_with_controlled_test_size(data)
64  # %%
65  print(f"Общий размер данных: {len(data)}")
66  print(f"Тренировочная выборка: {len(train_df)} записей ({len(train_df) / len(data)
    ↪  * 100:.1f}%)")
67  print(f"Валидационная выборка: {len(val_df)} записей ({len(val_df) / len(data) *
    ↪  100:.1f}%)")
68  # %%
69  val_df.describe()
70  # %% md
71  # Теперь в валидации нет повторяющихся заголовков.
72  # %%
73  val_df.to_csv("../data/training_data/val_df.csv", index=False)
74  # %% md
75  # val_df.to_csv("../data/val_df.csv", index=False)# Аугментация данных
76  # %% md
77  # Аугментировать будем только train набор, чтобы не произошла утечка.
78  # %%
79  train_df.reset_index(drop=True, inplace=True)
80
81
82  # %%
83  def split_text_into_chunks(text, sentences_per_chunk=3):
84      sentences = sent_tokenize(text, language="russian")
85      chunks = []
86      for i in range(0, len(sentences), sentences_per_chunk):
87          chunk = " ".join(sentences[i:i + sentences_per_chunk])
88          chunks.append(chunk)
89      return chunks
90
91
92  # %%
93  augmented_rows = []
94  for _, row in tqdm(train_df.iterrows(), total=len(train_df)):
95      chunks = split_text_into_chunks(row.text, sentences_per_chunk=3)
96      for chunk in chunks:
97          augmented_rows.append({"text": chunk, "title": row.title})
98  # %%
99  augmented_dataset = pd.DataFrame(augmented_rows)
100 # %%
101 augmented_dataset.describe()
102
103
104 # %% md
105 # ## Почистим строки, в которых нет текста и удалим лидирующую пунктуацию
106 # %%
107 def is_meaningful(text):
108     return bool(re.search(r"[А-Яа-яA-Za-z0-9]", text))
109
```

```
110
111 def clean_leading_punct(text):
112     return re.sub(r"^[^\wA-Яа-я0-9]+", "", text).strip()
113
114
115 # %%
116 augmented_dataset = augmented_dataset[augmented_dataset.text.apply(is_meaning
    ↪ ful)].reset_index(drop=True)
117 augmented_dataset.text = augmented_dataset.text.str.replace(r"[^\w\s,.!?-]", " ",
    ↪ regex=True)
118 augmented_dataset.text = augmented_dataset.text.str.replace(r"\s+", " ",
    ↪ regex=True).str.strip()
119 augmented_dataset.describe()
120 # %% md
121 # Удалим строки, с небольшим количеством данных
122 # %%
123 augmented_dataset = augmented_dataset[augmented_dataset.text.str.strip().str.len()
    ↪ > 10]
124 # %%
125 augmented_dataset.describe()
126 # %% md
127 # Заметим, что у нас очень много текстов с одинаковыми названиями. Это может
    ↪ плохо повлиять на модель, если она будет видеть одни и те же названия.
    ↪ Оставим только по 500 каждого
128 # %%
129 max_per_title = 500
130 augmented_dataset = augmented_dataset.groupby("title").head(max_per_title).rese
    ↪ t_index(drop=True)
131
132 print(augmented_dataset.title.value_counts().head(10))
133 # %%
134 augmented_dataset.describe()
135 # %% md
136 # Появились одинаковые тексты
137 # %%
138 augmented_dataset.duplicated().sum()
139 # %% md
140 # Есть полные дубликаты текст + название. Такое удалим.
141 # %%
142 augmented_dataset.drop_duplicates(inplace=True)
143 # %%
144 augmented_dataset.reset_index(drop=True, inplace=True)
145 # %%
146 augmented_dataset.describe()
147 # %% md
148 # Все еще остались одинаковые тексты, но теперь у них разные названия. Удалим
    ↪ и их.
149 # %%
150 augmented_dataset = augmented_dataset[~augmented_dataset.text.duplicated(keep
    ↪ =False)].reset_index(drop=True)
151 # %%
152 augmented_dataset.describe()
```

```
153 # %%
154 augmented_dataset
155 # %%
156 augmented_dataset.to_csv("../data/training_data/train_df.csv", index=False)
157 # %% md
158 # ## Посмотрим на распределение названий по длине после разделения
159 # %%
160 train_df = augmented_dataset.copy()
161 # %%
162 dup_titles = (
163     augmented_dataset.groupby("text")["title"]
164     .nunique()
165     .reset_index()
166     .query("title > 1")
167 )
168
169 print(f"Текстов с одинаковыми содержаниями, но разными названиями:
    ↪  {len(dup_titles)}")
170 # %%
171 data["title_len"] = data.title.apply(lambda x: len(str(x).split()))
172 length_counts = data.groupby('title_len').size()
173 train_df["title_len"] = train_df.title.apply(lambda x: len(str(x).split()))
174 length_counts_aug = train_df.groupby('title_len').size()
175 val_df["title_len"] = val_df.title.apply(lambda x: len(str(x).split()))
176 length_counts_aug_val = val_df.groupby('title_len').size()
177 # %%
178 fig = go.Figure()
179
180 fig.add_trace(go.Bar(
181     x=length_counts.index,
182     y=length_counts.values,
183     name="Без аугментации",
184     marker_color="blue"
185 ))
186
187 fig.add_trace(go.Bar(
188     x=length_counts_aug.index,
189     y=length_counts_aug.values,
190     name="train после аугментации",
191     marker_color="orange"
192 ))
193
194 fig.add_trace(go.Bar(
195     x=length_counts_aug_val.index,
196     y=length_counts_aug_val.values,
197     name="test после аугментации",
198     marker_color="green"
199 ))
200
201 fig.update_layout(
202     title="Сравнение распределения длин названий до и после аугментации",
203     xaxis_title="Длина названия (слов)",
204     yaxis_title="Количество названий",
```

```
205         barmode="group",
206         bargap=0.2,
207         bargroupgap=0.1,
208         width=1000,
209         height=500
210     )
211
212     fig.show()
```

ПРИЛОЖЕНИЕ В

Модель на основе n-gram

```
import random
import re
from collections import defaultdict, Counter
from typing import List, Tuple
from nltk.translate.meteor_score import meteor_score
import pickle

import pandas as pd
from tqdm import tqdm


def tokenize(text: str) -> List[str]:
    """Функция самой простой предобработки текста, основанной на разбиении на токены по пробелам."""
    text = text.lower()
    text = re.sub(r"[^a-za-яё0-9\s]", "", text)
    return text.split()


class TitleNgramModel:
    """Модель n-грамм, обучающаяся по парам (текст → название)."""

    def __init__(self, n_gram: int = 4, smoothing: str | None = 'laplace', alpha: float = 1.):
        self.n_gram = n_gram
        self.ngrams = defaultdict(Counter)
        self.context_counts = defaultdict(int)
        self.vocab = set()
        self.start_token = "<s>"
        self.title_token = "<title>"
        self.end_token = "</s>"
        self.unknown_token = "<unk>"
        self.smoothing = smoothing
        self.alpha = alpha
        self.lower_order_models = {}

    def create_ngrams(self, tokens: List[str]) -> list[tuple[str, ...]]:
        """Создание n-грамм с учётом начала и конца."""
        tokens = [self.start_token] + tokens + [self.end_token]
        return [tuple(tokens[i:i + self.n_gram]) for i in range(len(tokens) - self.n_gram + 1)]

    def train(self, pairs: List[Tuple[str, str]]) -> None:
        """Обучение на списке пар (text, title)"""
        all_tokens = []
        for text, title in pairs:
            text_tokens = tokenize(text)
            title_tokens = tokenize(title)
            combined = text_tokens + [self.title_token] + title_tokens
```

```
47              all_tokens.extend(combined)
48
49          token_counts = Counter(all_tokens)
50          self.vocab = set(token for token, count in token_counts.items() if count > 1)
51          self.vocab.update([self.start_token, self.title_token, self.end_token,
            ↪  self.unknown_token])
52
53          if self.n_gram > 1:
54              for n in range(1, self.n_gram):
55                self.lower_order_models[n] = TitleNgramModel(n, self.smoothing, self.alpha)
56                 self.lower_order_models[n].train(pairs)
57
58          for text, title in pairs:
59              text_tokens = tokenize(text)
60              title_tokens = tokenize(title)
61
62              text_tokens = [token if token in self.vocab else self.unknown_token for token
                ↪  in text_tokens]
63              title_tokens = [token if token in self.vocab else self.unknown_token for token
                ↪  in title_tokens]
64
65              combined = text_tokens + [self.title_token] + title_tokens
66              ngrams_list = self.create_ngrams(combined)
67
68              for ngram in ngrams_list:
69                  context = ngram[:-1]
70                  next_word = ngram[-1]
71                  self.ngrams[context][next_word] += 1
72                  self.context_counts[context] += 1
73
74      def train_from_csv(self, csv_path: str, text_col: str = "text", title_col: str =
            ↪  "title",
75                         limit: int | None = None):
76          """Обучение модели по CSV-файлу"""
77          df = pd.read_csv(csv_path)
78          if limit:
79              df = df.head(limit)
80
81          pairs = []
82          for _, row in tqdm(df.iterrows(), total=len(df), desc="Обучение модели"):
83              text = getattr(row, text_col, None)
84              title = getattr(row, title_col, None)
85              if text and title:
86                  pairs.append((text, title))
87
88          self.train(pairs)
89
90          print(f"Обучение завершено. Количество уникальных контекстов:
            ↪  {len(self.ngrams)}")
91
92      def get_simple_probability(self, context: tuple, word: str) -> float:
93          """Вычисляет вероятность по формуле P(w_t | context) = C(context + word)
            ↪  / C(context)"""
```

```
94          context_counts = self.ngrams.get(context, {})
95          total = sum(context_counts.values())
96          return context_counts[word] / total if total else 0.0
97
98      def get_laplace_probability(self, context: tuple, word: str) -> float:
99          """Сглаживание Лапласа (Add-one)."""
100         context_counts = self.ngrams.get(context, {})
101         total = sum(context_counts.values())
102         vocab_size = len(self.vocab)
103
104         count_word = context_counts.get(word, 0)
105         return (count_word + self.alpha) / (total + self.alpha * vocab_size)
106
107     def get_linear_interpolation_probability(self, context: tuple, word: str) -> float:
108         """Линейная интерполяция с моделями меньшего порядка."""
109         if self.n_gram == 1 or not self.lower_order_models:
110             return self.get_laplace_probability(context, word)
111
112         lambda_current = 0.6
113         lambda_backoff = 0.4
114
115         current_prob = self.get_laplace_probability(context, word)
116
117         backoff_context = context[1:] if len(context) > 1 else tuple()
118         backoff_model = self.lower_order_models[self.n_gram - 1]
119         backoff_prob = backoff_model.get_probability(backoff_context, word)
120
121         return lambda_current * current_prob + lambda_backoff * backoff_prob
122
123     def get_probability(self, context: tuple, word: str) -> float:
124         """Вычисляет вероятность с выбранным методом сглаживания."""
125
126         if word not in self.vocab:
127             word = self.unknown_token
128
129         if self.smoothing == 'laplace':
130             return self.get_laplace_probability(context, word)
131         elif self.smoothing == 'linear':
132             return self.get_linear_interpolation_probability(context, word)
133         elif self.smoothing is None:
134             return self.get_simple_probability(context, word)
135         else:
136             return self.get_laplace_probability(context, word)
137
138     def generate_with_backoff(self, context: tuple, max_depth: None | int = None):
139         """Генерация с backoff: если контекст не найден, используем контекст
            ↪ короче."""
140
141         if max_depth is None:
142             max_depth = self.n_gram - 1
143
144         current_context = context
145         depth = 0
```

```
146
147         while depth <= max_depth:
148             if current_context in self.ngrams:
149                 probabilities = dict()
150
151                 for word in self.vocab:
152                     if word not in [self.title_token, self.title_token, self.end_token]:
153                         probabilities[word] = self.get_probability(current_context, word)
154
155                 total_prob = sum(probabilities.values())
156                 if total_prob > 0:
157                     return {word: prob / total_prob for word, prob in probabilities.items()}
158
159             if len(current_context) > 1:
160                 current_context = current_context[1:]
161             else:
162                 break
163             depth += 1
164
165         words = [w for w in self.vocab if w not in [self.start_token, self.title_token,
            ↪  self.end_token]]
166         return {word: 1.0 / len(words) for word in words}
167
168     def generate_title(self, input_text: str, max_words: int = 7) -> str:
169         """Генерация названия на основе входного текста с использованием backoff и
            ↪  сглаживания."""
170         tokens = tokenize(input_text)
171         tokens = [token if token in self.vocab else self.unknown_token for token in tokens]
172         tokens += [self.title_token]
173         context = tuple(tokens[-(self.n_gram - 1):])
174         result = []
175
176         for _ in range(max_words):
177             word_probs = self.generate_with_backoff(context)
178
179             if not word_probs:
180                 break
181
182             word_probs = {word: prob for word, prob in word_probs.items()}
183             total = sum(word_probs.values())
184             word_probs = {word: prob / total for word, prob in word_probs.items()}
185             words = list(word_probs.keys())
186             probs = list(word_probs.values())
187
188             next_word = random.choices(words, weights=probs)[0]
189
190             if next_word in {self.end_token, self.title_token}:
191                 break
192
193             result.append(next_word)
194             context = (*context[1:], next_word) if len(context) > 0 else (next_word,)
195
196         title = " ".join(result)
```

```
197         return title if title else "Без названия"
198
199     def evaluate_meteor(self, test_pairs: List[Tuple[str, str]]) -> float:
200         """Вычисление средней METEOR-оценки для тестового набора (text,
            ↪ reference_title)."""
201         scores = []
202         for text, true_title in tqdm(test_pairs, desc="Оценка METEOR"):
203             generated = self.generate_title(text)
204             score = meteor_score([tokenize(true_title)], tokenize(generated))
205             scores.append(score)
206         return sum(scores) / len(scores) if scores else 0.0
207
208     def save(self, path: str):
209         with open(path, "wb") as f:
210             pickle.dump(self, f)
211
212     @staticmethod
213     def load(path: str):
214         with open(path, "rb") as f:
215             return pickle.load(f)
```

```
1  import pandas as pd
2  from n_gram_model import TitleNgramModel
3
4  train_df = "../../data/training_data/train_df.csv"
5  val_df = pd.read_csv("../../data/training_data/val_df.csv")
6
7  model = TitleNgramModel(n_gram=3)
8  model.train_from_csv(train_df)
9  model.save(path="../../models/history/title_ngram_model.pkl")
10
11 val_pairs = list(zip(val_df.text, val_df.title))
12 meteor = model.evaluate_meteor(val_pairs)
13 print(f"\nСредняя METEOR-оценка на валидационном наборе: {meteor:.4f}")
```

```
1  from n_gram_model import TitleNgramModel
2
3  model = TitleNgramModel.load("../../models/history/title_ngram_model.pkl")
4
5  while True:
6      text = input("> Введите текст или exit для выхода")
7      if text.lower() == "exit":
8          break
9      print(f"Сгенерированное название: {model.generate_title(text)}")
```