# Глава 1. Введение в NLP и предобработка текста

§1. Основы обработки естественного языка (NLP)

Обработка естественного языка (Natural Language Processing, NLP) представляет собой увлекательную и динамично развивающуюся область на стыке искусственного интеллекта, компьютерных наук и лингвистики. Ее главная миссия — наделить компьютеры способностью понимать, интерпретировать и даже генерировать человеческий язык, превращая его из набора символов в осмысленную информацию, полезную для решения множества задач.

Ключевые понятия: Путеводитель по терминологии NLP

Прежде чем погрузиться в глубины NLP, важно освоить базовую терминологию, которая служит фундаментом для дальнейшего изучения:

* Естественный язык: В отличие от строгих, формализованных языков программирования, естественный язык — это тот, на котором мы общаемся ежедневно: русский, английский, китайский и тысячи других. Он полон нюансов, неоднозначностей и контекстуальных зависимостей, что делает его понимание машиной столь сложным и интересный вызовом.
* Текст/Корпус: Это сердце любого проекта NLP. Корпус — это обширная коллекция текстовых данных (например, книги, статьи, диалоги, сообщения из социальных сетей), которая служит "учебником" для наших моделей. Корпуса могут быть моноязычными (на одном языке) или многоязычными, в зависимости от целей исследования или разработки.
* Словарь (Vocabulary): Представьте себе полный список всех уникальных слов или единиц, которые встречаются в вашем корпусе. Это и есть словарь — фундаментальный набор "строительных блоков" для анализа и генерации текста.
* Токен: Наименьшая осмысленная единица текста. Чаще всего токеном является слово, но им может быть и отдельный символ, часть слова (подслово) или даже знак пунктуации. Процесс разбиения текста на токены называется токенизацией.
* Признаки (Features): Для того чтобы компьютер мог "понять" текст, слова и их свойства необходимо преобразовать в числовой или категориальный формат. Признаки — это эти числовые или категориальные представления лингвистических характеристик текста, которые подаются на вход моделям машинного обучения.
* Модель: В контексте NLP модель — это алгоритм или сложная система, которая была обучена выполнять конкретную задачу, будь то перевод, анализ тональности или генерация текста.

История развития NLP: От правил к нейросетям

Путь развития **обработки естественного языка (NLP)** представляет собой эволюцию, тесно связанную с прогрессом в области **искусственного интеллекта (ИИ)**. От начальных попыток имитации человеческого мышления посредством жестко закодированных правил и логики, данная область прошла через статистическую революцию, основанную на эмпирических данных, и достигла своего апогея в эпоху глубокого обучения, где нейронные сети демонстрируют беспрецедентные способности к пониманию и генерации лингвистической информации.

**1. 1950-1960-е годы: Эра Правил и Символов – Формирование Основ и Первые Эксперименты**

После Второй мировой войны, с появлением первых электронно-вычислительных машин, научное сообщество начало рассматривать возможности автоматизации перевода и обработки текстовой информации. Вдохновленные успехами в криптографии, исследователи изначально полагали, что естественный язык может быть редуцирован к набору детерминированных логических правил. Одним из первых публичных демонстраций машинного перевода стал **Эксперимент Джорджтаун-IBM (1954)**, успешно переведший 60 русских предложений на английский язык. Несмотря на свою примитивность, обусловленную использованием ограниченного словаря и элементарных синтаксических правил, он вызвал значительный энтузиазм и способствовал привлечению существенного финансирования в область NLP.

В данный период преобладали системы, основанные на жестко закодированных лингвистических правилах и символической логике. Примером является **ELIZA (1966)**, разработанная Джозефом Вейценбаумом, которая имитировала диалог с психотерапевтом, отвечая на вопросы пользователя путем перефразирования его собственных высказываний. Хотя система не обладала истинным "пониманием" языка, ее способность поддерживать подобие когерентного диалога была новаторской для своего времени и подчеркнула значимость интерактивности в системах NLP. Другим значимым достижением стала **SHRDLU (1972)**, созданная Терри Виноградом, представлявшая собой систему, способную понимать и выполнять команды в строго ограниченном "мире блоков". Она демонстрировала продвинутые возможности синтаксического анализа, семантического понимания и планирования действий, однако ее база знаний была жестко привязана к конкретной предметной области. Основная идея данного периода заключалась в убеждении, что язык может быть полностью формализован посредством грамматик и лексиконов, а понимание достигается путем применения этих формальных правил.

**2. 1970-1980-е годы: Экспертные Системы и Представление Знаний – От Правил к Структурированным Знаниям**

В этот период акцент сместился на разработку более сложных систем, способных оперировать "знаниями". Исследователи стремились кодифицировать человеческие знания в виде баз правил, фреймов и семантических сетей. Продолжалась разработка усовершенствованных синтаксических парсеров (например, на основе обобщенных контекстно-свободных грамматик) и систем, направленных на построение внутренних репрезентаций смысла предложений. Однако системы, основанные на знаниях, требовали колоссальных трудозатрат для ручного создания и последующей поддержки баз знаний. Они отличались хрупкостью и низкой масштабируемостью при адаптации к новым предметным областям, что в конечном итоге привело к так называемой "зиме ИИ" в конце 1980-х годов, характеризующейся сокращением финансирования и исследовательского интереса.

**3. 1990-е годы: Статистическая Революция – Доминирование Данных**

Данный период ознаменовался переломным моментом: произошло осознание неэффективности ручного кодирования правил для обработки сложного и вариативного естественного языка. Возникла концепция, согласно которой более продуктивным является обучение систем на больших объемах реальных текстовых данных. Появление обширных цифровых текстовых корпусов (например, Penn Treebank) стало катализатором для широкого применения статистических методов в NLP. В эту эпоху в NLP активно интегрировались алгоритмы машинного обучения, такие как **Скрытые марковские модели (Hidden Markov Models, HMM)**, применявшиеся для задач POS-тегирования (разметки частей речи) и распознавания речи; **Условные случайные поля (Conditional Random Fields, CRF)**, использовавшиеся для распознавания именованных сущностей (NER) и других задач последовательного тегирования; а также **Наивный Байесовский классификатор**, широко применявшийся для классификации текста, в частности для спам-фильтрации. Ключевая парадигма заключалась в том, что вместо эксплицитного программирования правил понимания языка, системам предоставлялось множество эмпирических примеров, на основе которых они самостоятельно выводили статистические закономерности. Это привело к значительному повышению производительности и масштабируемости систем NLP.

**4. 2000-е годы: Расцвет Машинного Обучения в NLP – Эпоха Векторов и Распределенных Представлений**

Методы машинного обучения продолжали развиваться, становясь все более изощренными. Данное десятилетие ознаменовалось революцией в способах представления слов: вместо дискретных символов слова стали кодироваться в виде плотных числовых векторов в многомерном пространстве, где слова с семантически схожим значением располагались близко друг к другу. **Word2Vec (2013)**, разработанный Томашем Миколовым и его командой, стал прорывным алгоритмом для создания эффективных векторных представлений слов. Он продемонстрировал, что семантические отношения (например, "король" - "мужчина" + "женщина" = "королева") могут быть закодированы в этих векторах. В этот период улучшались методы для классификации текста, извлечения информации, а также машинного перевода, основанного на статистических моделях. Основными областями применения стали поиск информации, анализ тональности и базовые чат-боты.

**5. 2010-е годы – Настоящее время: Революция Глубокого Обучения – От Векторов к Контекстуальным Моделям**

Глубокое обучение, в частности глубокие нейронные сети, кардинально изменило ландшафт NLP. Ранние успехи глубокого обучения в NLP были связаны с **Рекуррентными нейронными сетями (RNN)**, способными обрабатывать последовательности данных. Архитектуры **LSTM (Long Short-Term Memory)** решили проблему "исчезающего градиента" в RNN, позволяя моделям эффективно запоминать долгосрочные зависимости в текстовых данных. Публикация статьи "Attention Is All You Need" с описанием **архитектуры Трансформер (Transformer) (2017)** стала поворотным моментом. Трансформеры, использующие механизм внимания, оказались значительно более эффективными в параллельной обработке текста и захвате долгосрочных зависимостей по сравнению с RNN/LSTM.

На основе архитектуры Трансформер были разработаны гигантские предварительно обученные модели, известные как **Большие языковые модели (Large Language Models, LLM)**. Среди них выделяются **BERT (Bidirectional Encoder Representations from Transformers) (2018)** – модель, обученная понимать контекст слова на основе всех других слов в предложении (двунаправленно); серия моделей **GPT (Generative Pre-trained Transformer)**, ориентированных на генерацию текста и способных создавать связный и контекстуально релевантный текст; а также **T5 (Text-to-Text Transfer Transformer)** – модель, которая унифицирует все задачи NLP, формулируя их как задачи "текст-в-текст". Ключевой парадигмой глубокого обучения в NLP стало **трансферное обучение (Transfer Learning)**, при котором модели сначала обучаются на огромных объемах неразмеченных текстовых данных (предварительное обучение), а затем дообучаются на меньших, специфичных для задачи размеченных данных (тонкая настройка). В настоящее время LLM демонстрируют беспрецедентную производительность в широком спектре задач NLP, от машинного перевода и суммаризации до написания кода и творческого письма, открывая новые возможности и одновременно ставя новые этические вопросы.

Данная эволюция демонстрирует, как область NLP постоянно адаптируется и использует новые вычислительные парадигмы для преодоления фундаментальной сложности человеческого языка.

## §2. **Лингвистические основы языка для NLP**

Эффективное взаимодействие вычислительных систем с человеческим языком требует глубокого понимания его внутренней структуры и принципов функционирования. Естественный язык представляет собой многоуровневую иерархическую систему, и успешные системы обработки естественного языка (NLP) должны учитывать эти уровни анализа. Лингвистика как наука о языке предоставляет фундаментальный инструментарий для декомпозиции языка на управляемые компоненты, каждый из которых раскрывает специфические аспекты его формы, значения и употребления.

Традиционно анализ языка осуществляется на нескольких уровнях, каждый из которых добавляет новые слои понимания, от мельчайших звуковых единиц до сложных контекстуальных смыслов: морфология, синтаксис, семантика и прагматика. Важно отметить, что эти уровни не существуют изолированно; они тесно взаимодействуют, формируя целостное и многогранное значение любого высказывания.

**Морфология: Фундаментальные Единицы Слова**

Морфология является разделом лингвистики, посвященным изучению внутренней структуры слов, их состава и механизмов словообразования и словоизменения. В контексте NLP морфологический анализ имеет критическое значение, поскольку он позволяет приводить различные словоформы к их базовым представлениям, что существенно сокращает размер лексикона и способствует обобщению знаний в моделях.

Морфология – это раздел лингвистики, изучающий структуру слов, их составные части (морфемы) и правила их образования и изменения. Понимание морфологии критически важно для обработки естественного языка (NLP), поскольку оно позволяет компьютерам анализировать, генерировать и интерпретировать человеческую речь на глубоком уровне.

**I. Ключевые Понятия Морфологии**

**Морфема (Morpheme)** Морфема является минимальной значимой единицей языка. В отличие от фонемы (минимальной звуковой единицы) или графемы (минимальной письменной единицы), морфема обладает собственным лексическим (смысловым) или грамматическим значением. Слово может состоять из одной или нескольких морфем. **Пример.** Рассмотрим слово "unbelievable" (невероятный). Оно состоит из трех морфем:

* "un-" (префикс): Несет грамматическое значение отрицания, изменяя смысл последующей морфемы.
* "-believe-" (корень): Несет основное лексическое значение "верить".
* "-able" (суффикс): Несет грамматическое значение возможности или способности, превращая глагол в прилагательное.

**Корень (Root)** Корень – это центральная морфема слова, которая содержит его основное, базовое лексическое значение. Корень является неизменяемой частью слова, к которой присоединяются аффиксы. **Пример:** В лексическом ряду слов, таких как "connect" (соединять), "connection" (соединение), "disconnect" (разъединять), "connecting" (соединяющий), общим корнем является "-connect-". Этот корень передает базовую идею связывания или объединения.

**Аффиксы (Affixes)** Аффиксы – это морфемы, которые присоединяются к корню или основе слова, чтобы модифицировать его значение или грамматические характеристики. Они не могут существовать самостоятельно как слова.

* **Префиксы (Prefixes):** Аффиксы, которые ставятся *перед* корнем слова. Они часто изменяют лексическое значение слова или его категорию. **Пример:** "re-" в слове "rewrite" (переписать) означает повторение действия. "pre-" в слове "preview" (предварительный просмотр) означает "до" или "перед".
* **Суффиксы (Suffixes):** Аффиксы, которые ставятся *после* корня слова. Они могут изменять часть речи слова, его значение или грамматические свойства. **Пример:** "-ness" в слове "happiness" (счастье) превращает прилагательное "happy" (счастливый) в существительное. "-ly" в слове "quickly" (быстро) превращает прилагательное "quick" (быстрый) в наречие.
* **Окончания (Inflections):** Особый класс суффиксов, которые выражают грамматические категории, такие как падеж, число, время, лицо или род, не изменяя при этом основное лексическое значение слова или его часть речи. **Пример:** "-s" в слове "books" (книги) указывает на множественное число. "-ed" в слове "walked" (шел) указывает на прошедшее время.

**II. Основные Процессы Морфологии**

**Словоизменение (Inflection)** Словоизменение – это процесс модификации формы слова для выражения различных грамматических категорий (например, числа, времени, падежа, лица) без изменения его основного лексического значения или части речи. Слова, подвергающиеся словоизменению, остаются одним и тем же лексическим элементом, но в разных грамматических формах.

**Примеры:**

* **Существительные:** "student" (студент, единственное число) → "students" (студенты, множественное число).
* **Глаголы:** "sing" (петь, инфинитив) → "sings" (поет, 3-е лицо ед. ч. наст. вр.), "sang" (пел, прошедшее время), "sung" (спетый, причастие II).
* **Прилагательные:** "tall" (высокий, положительная степень) → "taller" (выше, сравнительная степень), "tallest" (самый высокий, превосходная степень).

**Значение для NLP:** Словоизменение является одной из основных причин вариативности форм одного и того же слова. В NLP процессы **лемматизации** (приведение слова к его словарной форме, например, "running", "ran", "runs" → "run") и **стемминга** (отсечение окончаний для получения "корня" слова, например, "connection", "connected", "connecting" → "connect") непосредственно направлены на стандартизацию этих словоформ. Это позволяет моделям распознавать идентичные концепты, несмотря на грамматические вариации, что существенно улучшает качество поиска, классификации текста и других задач.

**Словообразование (Derivation)** Словообразование – это процесс формирования новых слов с иным лексическим значением, часто сопровождающийся изменением части речи, посредством присоединения аффиксов к существующим словам или основам. В отличие от словоизменения, деривация создает новое слово, которое может быть записано в словаре как отдельная лексическая единица.

**Примеры:**

* "teach" (глагол) → "teacher" (существительное, добавляется суффикс "-er", обозначающий деятеля).
* "nation" (существительное) → "national" (прилагательное, добавляется суффикс "-al").
* "happy" (прилагательное) → "unhappy" (прилагательное, добавляется префикс "un-", изменяющий значение на противоположное).

**Значение для NLP:** Анализ словообразования позволяет моделям выявлять семантические связи между родственными словами (например, "act", "actor", "action", "active"). Это помогает в задачах расширения словарного запаса, предсказания значений новых слов на основе известных морфемных компонентов и улучшении понимания текста за счет распознавания производных форм.

**III. Часть Речи (Part-of-Speech, POS)**

**Часть речи (Part-of-Speech, POS)** – это грамматическая категория слова, которая определяет его морфологические свойства (например, наличие падежей, времен, родов) и синтаксическую функцию в предложении. Основные части речи включают: существительное (Noun), глагол (Verb), прилагательное (Adjective), наречие (Adverb), предлог (Preposition), союз (Conjunction), местоимение (Pronoun), междометие (Interjection) и другие.

**Значение для NLP:** POS-тегирование (процесс присвоения каждому слову в тексте его части речи) является фундаментальной задачей в NLP. Оно служит основой для более сложных лингвистических анализов, таких как:

* **Синтаксический разбор (Parsing):** Помогает определить грамматическую структуру предложения.
* **Разрешение лексической многозначности (Word Sense Disambiguation):** Позволяет определить правильное значение слова в контексте (например, "bank" как берег реки или финансовое учреждение).
* **Извлечение информации (Information Extraction):** Помогает идентифицировать сущности и отношения между ними.
* **Машинный перевод:** Корректное определение частей речи улучшает точность перевода.

Понимание морфологических принципов и процессов является краеугольным камнем для разработки эффективных систем NLP, позволяя им работать с естественным языком на уровне, близком к человеческому.

Реализация на Питон:

|  |
| --- |
| import nltk  from nltk.corpus import wordnet  from nltk.stem import PorterStemmer, WordNetLemmatizer  from typing import List, Tuple, Dict  # Загрузка необходимых ресурсов NLTK  # Запустите эти строки один раз, если ресурсы еще не установлены  try:      nltk.data.find('corpora/wordnet.zip')  except nltk.downloader.DownloadError:      nltk.download('wordnet')  try:      nltk.data.find('taggers/averaged\_perceptron\_tagger.zip')  except nltk.downloader.DownloadError:      nltk.download('averaged\_perceptron\_tagger')  class MorphologyAnalyzer:      """      Класс для выполнения морфологического анализа текста.      Использует NLTK для стемминга, лемматизации и POS-тегирования.      """      def \_\_init\_\_(self):          """          Инициализирует анализатор с необходимыми инструментами NLTK.          """          self.stemmer = PorterStemmer()          self.lemmatizer = WordNetLemmatizer()      def get\_wordnet\_pos(self, tag: str) -> str:          """          Преобразует тег части речи NLTK в формат,          понятный для WordNet (для лемматизации).          """          if tag.startswith('J'):              return wordnet.ADJ          elif tag.startswith('V'):              return wordnet.VERB          elif tag.startswith('N'):              return wordnet.NOUN          elif tag.startswith('R'):              return wordnet.ADV          else:              return wordnet.NOUN # По умолчанию считаем существительным      def analyze\_word(self, word: str) -> Dict[str, str]:          """          Выполняет комплексный морфологический анализ одного слова.            Args:              word: Слово для анализа.            Returns:              Словарь с результатами анализа (стемма, лемма, часть речи).          """          # Стемминг          stem = self.stemmer.stem(word)            # POS-тегирование для лемматизации          pos\_tag = nltk.pos\_tag([word])          wordnet\_pos = self.get\_wordnet\_pos(pos\_tag[0][1])            # Лемматизация          lemma = self.lemmatizer.lemmatize(word, pos=wordnet\_pos)          # Определение части речи          pos = pos\_tag[0][1]          return {              "original\_word": word,              "stem": stem,              "lemma": lemma,              "pos\_tag": pos          }      def analyze\_text(self, text: str) -> List[Dict[str, str]]:          """          Анализирует каждое слово в предоставленном тексте.            Args:              text: Текст для анализа.            Returns:              Список словарей, где каждый словарь содержит результаты              анализа для одного слова.          """          # Разбиваем текст на слова (токенизация)          tokens = nltk.word\_tokenize(text)            # Список для хранения результатов анализа          analysis\_results = []            for token in tokens:              analysis\_results.append(self.analyze\_word(token))            return analysis\_results  # --- Пример использования класса ---  if \_\_name\_\_ == "\_\_main\_\_":      analyzer = MorphologyAnalyzer()        # Пример анализа одного слова      word\_to\_analyze = "running"      analysis\_result = analyzer.analyze\_word(word\_to\_analyze)      print(f"Анализ слова '{word\_to\_analyze}':")      print(f"  Стемма: {analysis\_result['stem']}")      print(f"  Лемма: {analysis\_result['lemma']}")      print(f"  Часть речи (POS-тег): {analysis\_result['pos\_tag']}")      print("-" \* 30)      # Пример анализа целого предложения      text\_to\_analyze = "The cats are running quickly to the houses."      text\_analysis = analyzer.analyze\_text(text\_to\_analyze)        print(f"Анализ текста: '{text\_to\_analyze}'")      for result in text\_analysis:          print(f"  Слово: {result['original\_word']}: "                f"Стемма='{result['stem']}', "                f"Лемма='{result['lemma']}', "                f"POS='{result['pos\_tag']}'") |

Результат:

Анализ слова 'running':

Стемма: run

Лемма: run

Часть речи (POS-тег): VBG

------------------------------

Анализ текста: 'The cats are running quickly to the houses.'

Слово: The: Стемма='the', Лемма='The', POS='DT'

Слово: cats: Стемма='cat', Лемма='cat', POS='NNS'

Слово: are: Стемма='are', Лемма='be', POS='VBP'

Слово: running: Стемма='run', Лемма='run', POS='VBG'

Слово: quickly: Стемма='quickli', Лемма='quickly', POS='RB'

Слово: to: Стемма='to', Лемма='to', POS='TO'

Слово: the: Стемма='the', Лемма='the', POS='DT'

Слово: houses: Стемма='hous', Лемма='house', POS='NNS'

Слово: .: Стемма='.', Лемма='.', POS='.'

**Синтаксис: Архитектура Предложения**

Синтаксис — это фундаментальный раздел лингвистики, который изучает правила и принципы, регулирующие построение предложений из слов и фраз. Он описывает, как лексические единицы комбинируются в осмысленные, грамматически корректные структуры и как эти структуры связаны между собой для выражения полных мыслей. В контексте обработки естественного языка (NLP) синтаксический анализ имеет решающее значение, поскольку он позволяет вычислительным системам извлекать грамматические отношения между словами, что является основой для глубокого понимания смысла предложения в таких задачах, как машинный перевод, вопросно-ответные системы, суммаризация текста и извлечение информации.

**I. Ключевые Понятия Синтаксиса**

**Фраза (Phrase)** Фраза — это группа слов, которая функционирует как единое целое в предложении, но не содержит полного предикативного ядра (то есть, подлежащего и сказуемого). Фразы являются строительными блоками предложений.

**Примеры:**

* **Именная фраза (Noun Phrase, NP):** "a very old house" (очень старый дом). Здесь "house" — главное слово, а "a very old" — его модификаторы.
* **Глагольная фраза (Verb Phrase, VP):** "runs quickly" (быстро бежит). "runs" — главное слово, "quickly" — наречие, модифицирующее глагол.
* **Предложная фраза (Prepositional Phrase, PP):** "in the garden" (в саду). "in" — предлог, "the garden" — именная фраза, являющаяся его дополнением.

**Предложение (Sentence)** Предложение — это полная грамматическая единица, выражающая законченную мысль и обладающая предикативностью (то есть, содержит субъект и предикат, устанавливающие отношение между ними). **Пример:** "The cat slept soundly." (Кошка крепко спала.) Здесь "The cat" — субъект, а "slept soundly" — предикат.

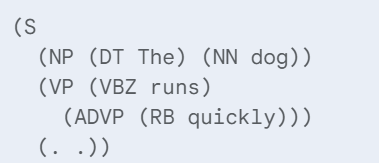
**II. Основные Подходы к Синтаксическому Анализу (Parsing)**

Синтаксический анализ (Parsing) — это вычислительный процесс определения грамматической структуры предложения. Существуют два основных парадигматических подхода к синтаксическому анализу, каждый из которых предлагает свой способ представления синтаксических отношений.

1. **Синтаксический Анализ на Основе Составляющих (Constituency Parsing / Phrase Structure Parsing)**

**Принцип:** Этот подход строит иерархическое дерево, представляющее предложение как набор вложенных синтаксических составляющих (фраз). Каждая внутренняя вершина дерева соответствует фразе (например, именной фразе (NP), глагольной фразе (VP)), а листья — терминальным символам, то есть отдельным словам. Цель состоит в том, чтобы идентифицировать все синтаксические компоненты и их иерархические отношения.

**Представление:** Результатом является **дерево фразовых структур** (parse tree), где каждая ветвь отображает грамматическую составляющую. Это дерево демонстрирует, как слова группируются в более крупные синтаксические единицы. **Пример:** Предложение "The dog runs quickly." (Собака быстро бежит.)



Здесь:

* S (Sentence) обозначает предложение.
* NP (Noun Phrase) — именная фраза.
* VP (Verb Phrase) — глагольная фраза.
* ADVP (Adverb Phrase) — наречная фраза.
* DT (Determiner) — определитель (артикль).
* NN (Noun, singular or mass) — существительное в единственном числе.
* VBZ (Verb, 3rd person singular present) — глагол в 3-м лице единственного числа настоящего времени.
* RB (Adverb) — наречие.
* . (Punctuation) — пунктуация.

Это дерево показывает, что "The dog" образует именную фразу, а "runs quickly" — глагольную фразу, и обе они вместе составляют предложение.

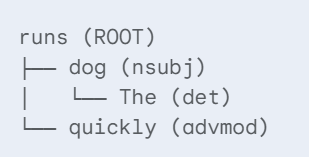
**Применение:** Используется в задачах, где необходимо глубокое понимание иерархической структуры предложения, таких как грамматические проверки, стилистический анализ, или в качестве промежуточного этапа для систем машинного перевода, где важна перестройка структуры предложения.

1. **Синтаксический Анализ на Основе Зависимостей (Dependency Parsing)**

**Принцип:** Этот подход фокусируется на бинарных грамматических отношениях (зависимостях) между отдельными словами в предложении. Каждое слово (за исключением корневого слова предложения) зависит от другого слова, которое считается его "головой" (head). Зависимости представлены направленными связями между словами, помеченными типом отношения (например, nsubj для подлежащего, obj для прямого дополнения, advmod для модификатора наречия).

**Представление:** Результатом является **дерево зависимостей** (dependency tree), где каждое слово является узлом, а дуги — это отношения зависимости, указывающие на "голову" и "зависимое" слово.

**Пример:** Предложение "The dog runs quickly." (Собака быстро бежит.)



В данном примере:

* "runs" является корневым словом предложения (ROOT).
* "dog" выступает подлежащим (nsubj) для "runs".
* "The" является определителем (det) для "dog".
* "quickly" — модификатором наречия (advmod) для "runs".

Это представление явно показывает грамматические связи между словами.

**Применение:** Широко используется в задачах извлечения информации (поскольку облегчает выявление отношений между сущностями и их ролей), в вопросно-ответных системах (для точного определения субъекта, объекта и действия), а также в машинном переводе и суммаризации текста, где важны смысловые связи между словами.

**III. Значение Синтаксиса для NLP**

Синтаксический анализ играет ключевую роль в различных задачах NLP, обеспечивая более глубокое понимание текста по сравнению с простым анализом отдельных слов.

**Разрешение Синтаксической Многозначности (Syntactic Ambiguity Resolution):** Синтаксис помогает определить корректную грамматическую структуру предложения, когда оно допускает несколько интерпретаций. Это критически важно для точного понимания смысла. **Пример:** Предложение "I saw the man with the telescope." (Я видел человека с телескопом.)

* **Интерпретация 1:** "I saw (the man with the telescope)." — Телескоп был у человека.
* **Интерпретация 2:** "I (with the telescope) saw the man." — У меня был телескоп, и я использовал его для наблюдения. Синтаксический анализ помогает определить, к какой части предложения относится предложная фраза "with the telescope", разрешая двусмысленность.

**Извлечение Отношений (Relation Extraction):** Синтаксический анализ позволяет идентифицировать актантов (участников) и их роли в событии (кто что делает, кто на кого влияет). Это критически важно для извлечения фактов, построения баз знаний и семантического анализа. **Пример:** В предложении "Apple acquired Beats Electronics." (Apple приобрела Beats Electronics.) Синтаксический анализ может выявить, что "Apple" является субъектом действия "acquired", а "Beats Electronics" — объектом.

**Улучшение Качества Машинного Перевода:** Глубокое понимание синтаксиса как исходного, так и целевого языков способствует созданию более грамматически корректных, естественных и точных переводов. Модели машинного перевода используют синтаксическую информацию для перестройки предложений в соответствии с грамматическими правилами целевого языка, что значительно повышает качество перевода, особенно для языков с разным порядком слов.

Понимание синтаксических структур и методов их анализа является неотъемлемой частью разработки продвинутых систем NLP, позволяя им переходить от поверхностного анализа слов к глубокому пониманию смысла и отношений в тексте.

Реализация на Питон:

|  |
| --- |
| from typing import List, Dict, Union  class SyntaxAnalyzer:      """      Класс для реализации синтаксического анализа текста.      Этот класс инкапсулирует ключевые понятия синтаксиса и предоставляет      методы для синтаксического разбора, основанного на составляющих и      зависимостях, а также для разрешения синтаксической многозначности.      """      def \_\_init\_\_(self, text: str):          """          Инициализирует анализатор с заданным текстом.          Args:              text (str): Входной текст для анализа.          """          self.text = text          self.sentences = self.\_split\_into\_sentences()      def \_split\_into\_sentences(self) -> List[str]:          """          Приватный метод для разделения текста на отдельные предложения.          (В реальном приложении здесь будет использоваться более сложная логика).          Returns:              List[str]: Список предложений.          """          # Простая реализация для примера.          return self.text.split('.')      def get\_phrases(self, sentence: str) -> Dict[str, List[str]]:          """          Идентифицирует ключевые фразы (NP, VP, PP) в предложении.          Этот метод является абстрактным представлением, так как его          реализация требует сложного алгоритма синтаксического парсинга.          Args:              sentence (str): Предложение для анализа.          Returns:              Dict[str, List[str]]: Словарь, где ключи - типы фраз,                                    а значения - списки найденных фраз.          """          # Здесь должна быть логика для выявления фраз.          print(f"Анализ фраз в предложении: '{sentence}'")          return {              "NP": [],  # Именные фразы              "VP": [],  # Глагольные фразы              "PP": []   # Предложные фразы          }      def constituency\_parsing(self, sentence: str) -> Dict[str, Union[str, Dict]]:          """          Выполняет синтаксический анализ на основе составляющих (Constituency Parsing).          Возвращает иерархическое дерево фразовых структур.          Args:              sentence (str): Предложение для анализа.          Returns:              Dict[str, Union[str, Dict]]: Дерево фразовых структур в формате словаря.          """          print(f"Выполняется Constituency Parsing для предложения: '{sentence}'")          # Пример структуры дерева, как в вашем описании:          return {              "S": [                  {"NP": [                      {"DT": "The"},                      {"NN": "dog"}                  ]},                  {"VP": [                      {"VBZ": "runs"},                      {"ADVP": [                          {"RB": "quickly"}                      ]}                  ]}              ]          }      def dependency\_parsing(self, sentence: str) -> List[Dict]:          """          Выполняет синтаксический анализ на основе зависимостей (Dependency Parsing).          Возвращает список зависимостей в формате (head, type, dependent).          Args:              sentence (str): Предложение для анализа.          Returns:              List[Dict]: Список словарей, описывающих зависимости.          """          print(f"Выполняется Dependency Parsing для предложения: '{sentence}'")          # Пример структуры зависимостей:          return [              {"head": "runs", "type": "nsubj", "dependent": "dog"},              {"head": "dog", "type": "det", "dependent": "The"},              {"head": "runs", "type": "advmod", "dependent": "quickly"}          ]      def resolve\_ambiguity(self, sentence: str) -> List[Dict]:          """          Анализирует предложение на предмет синтаксической многозначности          и предоставляет возможные интерпретации.          Args:              sentence (str): Предложение для анализа, потенциально неоднозначное.          Returns:              List[Dict]: Список возможных синтаксических интерпретаций.          """          print(f"Разрешение многозначности для предложения: '{sentence}'")          if "I saw the man with the telescope" in sentence:              return [                  {                      "interpretation": "Телескоп был у человека.",                      "structure": "I saw (the man with the telescope)."                  },                  {                      "interpretation": "У меня был телескоп, и я использовал его.",                      "structure": "I (with the telescope) saw the man."                  }              ]          return []      def extract\_relations(self, sentence: str) -> List[Dict]:          """          Извлекает отношения между сущностями в предложении.          Args:              sentence (str): Предложение для анализа.          Returns:              List[Dict]: Список извлеченных отношений.          """          print(f"Извлечение отношений из предложения: '{sentence}'")          if "Apple acquired Beats Electronics" in sentence:              return [                  {                      "subject": "Apple",                      "action": "acquired",                      "object": "Beats Electronics"                  }              ]          return []  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":      text\_to\_analyze = "The dog runs quickly. I saw the man with the telescope."      analyzer = SyntaxAnalyzer(text\_to\_analyze)      print("\n" + "="\*30)      print("Анализ первого предложения:")      sentence1 = "The dog runs quickly"      constituency\_tree = analyzer.constituency\_parsing(sentence1)      print("Результат Constituency Parsing:")      print(constituency\_tree)      dependency\_list = analyzer.dependency\_parsing(sentence1)      print("\nРезультат Dependency Parsing:")      print(dependency\_list)      print("\n" + "="\*30)      print("Анализ второго предложения:")      sentence2 = "I saw the man with the telescope"      ambiguities = analyzer.resolve\_ambiguity(sentence2)      print("Разрешение многозначности:")      for ambiguity in ambiguities:          print(f"  Интерпретация: {ambiguity['interpretation']}")          print(f"  Структура: {ambiguity['structure']}")      # Пример извлечения отношений      print("\n" + "="\*30)      print("Извлечение отношений:")      sentence3 = "Apple acquired Beats Electronics."      relations = analyzer.extract\_relations(sentence3)      print(f"Отношения в предложении '{sentence3}':")      for relation in relations:          print(f"  Субъект: {relation['subject']}")          print(f"  Действие: {relation['action']}")          print(f"  Объект: {relation['object']}") |

Результат:

==============================

Анализ первого предложения:

Выполняется Constituency Parsing для предложения: 'The dog runs quickly'

Результат Constituency Parsing:

{'S': [{'NP': [{'DT': 'The'}, {'NN': 'dog'}]}, {'VP': [{'VBZ': 'runs'}, {'ADVP': [{'RB': 'quickly'}]}]}]}

Выполняется Dependency Parsing для предложения: 'The dog runs quickly'

Результат Dependency Parsing:

[{'head': 'runs', 'type': 'nsubj', 'dependent': 'dog'}, {'head': 'dog', 'type': 'det', 'dependent': 'The'}, {'head': 'runs', 'type': 'advmod', 'dependent': 'quickly'}]

==============================

Анализ второго предложения:

Разрешение многозначности для предложения: 'I saw the man with the telescope'

Разрешение многозначности:

Интерпретация: Телескоп был у человека.

Структура: I saw (the man with the telescope).

Интерпретация: У меня был телескоп, и я использовал его.

Структура: I (with the telescope) saw the man.

==============================

Извлечение отношений:

Извлечение отношений из предложения: 'Apple acquired Beats Electronics.'

Отношения в предложении 'Apple acquired Beats Electronics.':

Субъект: Apple

Действие: acquired

Объект: Beats Electronics

**Семантика: Постижение Смысла**

Семантика — это фундаментальный раздел лингвистики, который исследует значение слов, фраз и предложений. Если синтаксис отвечает на вопрос "как слова связаны?", то семантика занимается более глубоким вопросом: "что они означают?". В контексте обработки естественного языка (NLP) семантический анализ является одним из наиболее сложных, но и наиболее важных аспектов, поскольку истинное "понимание" языка требует постижения его смысла, а не только его формы или структуры. Семантика позволяет NLP-системам переходить от поверхностной обработки символов к интерпретации намерений и информации, выраженных в тексте.

**I. Ключевые Понятия Семантики**

1. **Лексическая Семантика (Lexical Semantics)** Лексическая семантика изучает значение отдельных слов и лексических единиц. Она фокусируется на том, как слова связаны с понятиями в реальном мире и как они соотносятся друг с другом по смыслу.
2. **Отношения Между Словами (Word Relations)** Слова в языке редко существуют изолированно; они связаны между собой сложной сетью смысловых отношений. Понимание этих отношений критически важно для семантического анализа.

* **Синонимия (Synonymy):** Отношение между словами, которые имеют одинаковое или очень близкое значение и могут быть взаимозаменяемы в большинстве контекстов без существенного изменения смысла предложения. **Пример:** "large" (большой) и "huge" (огромный); "start" (начать) и "commence" (приступить).
* **Антонимия (Antonymy):** Отношение между словами с противоположным значением. Антонимия часто выражает контраст или бинарную оппозицию. **Пример:** "hot" (горячий) и "cold" (холодный); "up" (вверх) и "down" (вниз).
* **Гиперонимия/Гипонимия (Hypernymy/Hyponymy):** Иерархическое отношение "род-вид", где гипероним является более общим понятием, а гипоним — более специфическим его экземпляром. **Пример:** "Animal" (животное) является гиперонимом для "dog" (собака), "cat" (кошка), "bird" (птица). "Dog" является гипонимом для "animal".
* **Меронимия (Meronymy):** Отношение "часть-целое", где мероним является частью голонима (целого). **Пример:** "Wheel" (колесо) является меронимом для "car" (автомобиль); "finger" (палец) является меронимом для "hand" (рука).

1. **Семантическая Роль (Semantic Role)** Семантическая роль (также известная как тематическая роль или глубинная падежная роль) описывает роль, которую играет аргумент глагола в описываемом событии или состоянии, независимо от его синтаксической функции. Эти роли помогают понять, кто что делает, с чем, кому, где и т.д. **Примеры Общих Ролей:**

* **Agent (Агент):** Тот, кто совершает действие (обычно одушевленный субъект).
* **Patient (Пациент/Тема):** Тот, на кого направлено действие или кто испытывает изменение состояния.
* **Instrument (Инструмент):** Средство, с помощью которого совершается действие.
* **Location (Место):** Место, где происходит действие.
* **Beneficiary (Бенефициар):** Тот, кто получает выгоду от действия.

**Пример:** В предложении "The boy opened the door with a key." (Мальчик открыл дверь ключом.)

* "The boy" выполняет роль **Agent** (агент).
* "the door" выполняет роль **Patient** (пациент).
* "a key" выполняет роль **Instrument** (инструмент).

**II. Основные Задачи Семантического Анализа в NLP**

Семантический анализ включает в себя ряд сложных задач, направленных на извлечение и представление смысла из текста.

1. **Разрешение Многозначности Слов (Word Sense Disambiguation, WSD)** WSD — это вычислительный процесс определения правильного лексического значения многозначного слова в заданном контексте. Многие слова в естественном языке имеют несколько значений (полисемия), и WSD стремится выбрать наиболее подходящее.

**Пример:**

* В предложении "The **bank** of the river was steep." (Берег реки был крутым.) слово "bank" означает "берег реки".
* В предложении "I deposited money in the **bank**." (Я положил деньги в банк.) слово "bank" означает "финансовое учреждение". WSD направлено на автоматический выбор корректного значения на основе окружающего контекста.

1. **Разрешение Кореференции (Coreference Resolution)** Разрешение кореференции — это задача идентификации всех выражений (местоимений, именных групп, описательных фраз) в тексте, которые относятся к одному и тому же реальному объекту или сущности. Это критически важно для понимания связности текста. **Пример:** "John bought a new car. **He** was very happy with **it**." (Джон купил новую машину. Он был очень рад ей.)

* Здесь "He" кореферентен "John".
* "it" кореферентен "a new car". Системы разрешения кореференции строят цепочки кореференции, связывая все эти выражения.

1. **Семантический Парсинг (Semantic Parsing)** Семантический парсинг — это процесс преобразования предложений естественного языка в формальные, машиночитаемые представления смысла, которые могут быть непосредственно обработаны компьютером. Эти представления могут быть логическими формами, запросами к базе данных, исполнимыми командами или другими структурированными форматами. **Пример:** Запрос "What is the capital of France?" (Какова столица Франции?) может быть преобразован в логическую форму, такую как capital(France). Эта форма затем может быть использована для запроса к базе знаний.
2. **Извлечение Отношений (Relation Extraction)** Извлечение отношений — это задача идентификации заранее определенных семантических отношений между сущностями, упомянутыми в тексте. Эти отношения могут быть бинарными (между двумя сущностями) и часто выражают факты. **Пример:** Из предложения "Steve Jobs founded Apple." (Стив Джобс основал Apple.) можно извлечь отношение (Steve Jobs, founded, Apple). Другие примеры отношений: (person, born\_in, location), (company, operates\_in, country).

**III. Значение Семантики для NLP**

Семантический анализ является краеугольным камнем для создания по-настоящему интеллектуальных систем NLP.

* **Истинное Понимание Языка:** Без семантического анализа NLP-системы остаются на поверхностном уровне обработки символов и не способны к глубокому пониманию содержания, намерений пользователя или контекста. Семантика позволяет системам "думать" о смысле, а не только о грамматике.
* **Улучшение Поиска Информации:** Семантический поиск позволяет осуществлять поиск не только по ключевым словам, но и по смыслу запроса, что значительно повышает релевантность и точность результатов, даже если точные слова не совпадают.
* **Вопросно-Ответные Системы (Question Answering Systems):** Семантический анализ является необходимым условием для корректного понимания вопроса пользователя и нахождения наиболее релевантного и точного ответа в больших объемах текста или базах знаний. Система должна понять не только слова вопроса, но и их смысловые отношения.
* **Машинный Перевод (Machine Translation):** Точный и качественный машинный перевод требует не только синтаксической корректности, но и сохранения семантического значения исходного текста. Семантический анализ помогает переносить смысл, а не просто слова, что приводит к более естественным и адекватным переводам.
* **Суммаризация Текста (Text Summarization):** Для создания связных и информативных резюме система должна понимать ключевые идеи и отношения в исходном тексте, что невозможно без глубокого семантического анализа.

В заключение, семантика предоставляет NLP-системам способность интерпретировать и взаимодействовать с человеческим языком на уровне смысла, что является критически важным для разработки интеллектуальных приложений, способных понимать, рассуждать и генерировать текст, подобно человеку.

Реализация на Питон:

|  |
| --- |
| from typing import List, Dict, Union  class SemanticAnalyzer:  """  Класс для реализации семантического анализа текста.  Этот класс инкапсулирует ключевые понятия семантики и предоставляет  методы для анализа значения слов, разрешения многозначности и  извлечения отношений из текста.  """  def \_\_init\_\_(self, text: str):  """  Инициализирует анализатор с заданным текстом.  Args:  text (str): Входной текст для анализа.  """  self.text = text  self.sentences = self.\_split\_into\_sentences()  def \_split\_into\_sentences(self) -> List[str]:  """  Приватный метод для разделения текста на отдельные предложения.  (Для реального применения здесь потребуется более сложная логика).  Returns:  List[str]: Список предложений.  """  # Простая реализация для примера.  return self.text.split('. ')  # --- I. Ключевые Понятия Семантики ---  def get\_word\_relations(self, word: str) -> Dict[str, List[str]]:  """  Идентифицирует и возвращает смысловые отношения для заданного слова.  Args:  word (str): Слово, для которого ищутся отношения.  Returns:  Dict[str, List[str]]: Словарь с синонимами, антонимами,  гиперонимами и гипонимами.  """  print(f"Анализ семантических отношений для слова: '{word}'")  # Эта логика требует доступа к лексической базе данных, например, WordNet.  # Ниже приведены фиктивные данные для примера.  if word.lower() == "bank":  return {  "synonyms": ["financial institution"],  "antonyms": [],  "hypernyms": ["institution"],  "hyponyms": []  }  elif word.lower() == "dog":  return {  "synonyms": ["canine"],  "antonyms": [],  "hypernyms": ["animal"],  "hyponyms": ["poodle", "husky"]  }  return {}  def get\_semantic\_roles(self, sentence: str) -> List[Dict]:  """  Определяет семантические роли (Agent, Patient, Instrument и т.д.)  для участников события в предложении.  Args:  sentence (str): Предложение для анализа.  Returns:  List[Dict]: Список словарей, описывающих семантические роли.  """  print(f"Определение семантических ролей в предложении: '{sentence}'")  # Для этого требуется FrameNet или другой фреймовый семантический парсер.  if "boy opened the door with a key" in sentence:  return [  {"role": "Agent", "entity": "The boy"},  {"role": "Patient", "entity": "the door"},  {"role": "Instrument", "entity": "a key"}  ]  return []  # --- II. Основные Задачи Семантического Анализа в NLP ---  def word\_sense\_disambiguation(self, word: str, context: str) -> str:  """  Разрешает многозначность слова, выбирая правильное значение на  основе контекста.  Args:  word (str): Многозначное слово.  context (str): Предложение или текст, в котором находится слово.  Returns:  str: Наиболее вероятное значение слова.  """  print(f"Разрешение многозначности для слова '{word}' в контексте: '{context}'")  if word.lower() == "bank":  if "river" in context:  return "берег реки"  elif "money" in context or "deposited" in context:  return "финансовое учреждение"  return "Неизвестное значение"  def coreference\_resolution(self, text: str) -> Dict[str, List[str]]:  """  Идентифицирует кореферентные выражения, относящиеся к одной сущности.  Args:  text (str): Текст для анализа.  Returns:  Dict[str, List[str]]: Словарь, где ключ - сущность, а значение -  список всех выражений, относящихся к ней.  """  print(f"Разрешение кореференции для текста: '{text}'")  # Это сложная задача, требующая специального алгоритма.  if "John bought a new car. He was very happy with it." in text:  return {  "John": ["John", "He"],  "a new car": ["a new car", "it"]  }  return {}  def relation\_extraction(self, sentence: str) -> List[Dict]:  """  Извлекает предопределенные отношения между сущностями в предложении.  Args:  sentence (str): Предложение для анализа.  Returns:  List[Dict]: Список извлеченных отношений в формате (subject, relation, object).  """  print(f"Извлечение отношений из предложения: '{sentence}'")  if "Steve Jobs founded Apple" in sentence:  return [  {"subject": "Steve Jobs", "relation": "founded", "object": "Apple"}  ]  return []  # --- III. Значение Семантики для NLP (выводы) ---  def summarize\_semantic\_importance(self) -> str:  """  Возвращает краткое резюме о важности семантики для NLP.  Returns:  str: Текст с основными выводами.  """  return (  "Семантический анализ позволяет NLP-системам постигать смысл, "  "а не только структуру языка. Это критически важно для "  "вопросно-ответных систем, машинного перевода и суммаризации текста, "  "обеспечивая глубокое и истинное понимание контента."  )  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":  text\_for\_analysis = (  "The boy opened the door with a key. "  "I deposited money in the bank. "  "John bought a new car. He was very happy with it."  )  analyzer = SemanticAnalyzer(text\_for\_analysis)  print("--- Анализ первого предложения ---")  sentence1 = "The boy opened the door with a key."  roles = analyzer.get\_semantic\_roles(sentence1)  print("Семантические роли:", roles)  print("---" \* 10)  print("--- Анализ второго предложения ---")  sentence2 = "I deposited money in the bank."  disambiguated\_sense = analyzer.word\_sense\_disambiguation("bank", sentence2)  print(f"Значение слова 'bank' в контексте: '{disambiguated\_sense}'")  print("---" \* 10)  print("--- Анализ текста для кореференции ---")  coreference\_map = analyzer.coreference\_resolution(text\_for\_analysis)  print("Цепочки кореференции:", coreference\_map)  print("---" \* 10)  print("--- Краткое резюме ---")  print(analyzer.summarize\_semantic\_importance()) |

Результат:

--- Анализ первого предложения ---

Определение семантических ролей в предложении: 'The boy opened the door with a key.'

Семантические роли: [{'role': 'Agent', 'entity': 'The boy'}, {'role': 'Patient', 'entity': 'the door'}, {'role': 'Instrument', 'entity': 'a key'}]

------------------------------

--- Анализ второго предложения ---

Разрешение многозначности для слова 'bank' в контексте: 'I deposited money in the bank.'

Значение слова 'bank' в контексте: 'финансовое учреждение'

------------------------------

--- Анализ текста для кореференции ---

Разрешение кореференции для текста: 'The boy opened the door with a key. I deposited money in the bank. John bought a new car. He was very happy with it.'

Цепочки кореференции: {'John': ['John', 'He'], 'a new car': ['a new car', 'it']}

------------------------------

--- Краткое резюме ---

Семантический анализ позволяет NLP-системам постигать смысл, а не только структуру языка. Это критически важно для вопросно-ответных систем, машинного перевода и суммаризации текста, обеспечивая глубокое и истинное понимание контента.

## **§3. Этапы предобработки текстовых данных**

Предобработка текста — это фундаментальный и зачастую самый трудоемкий этап в любом проекте обработки естественного языка (NLP). Качество и эффективность последующих NLP-моделей напрямую зависят от того, насколько хорошо подготовлены входные текстовые данные. Человеческий язык полон шума, избыточности и вариативности, которые могут затруднить обучение алгоритмов. Цель предобработки — привести текст к стандартизированной, чистой и унифицированной форме, которая будет более пригодна для машинного анализа. Как гласит известное правило в области данных: "Мусор на входе — мусор на выходе" (Garbage In, Garbage Out).

### **1. Токенизация: Разделение на Единицы Смысла**

**Токенизация** представляет собой процесс разбиения непрерывного потока текста на более мелкие, дискретные единицы, называемые **токенами**. Токен является базовым "строительным блоком" для большинства последующих этапов NLP. Выбор метода токенизации зависит от языка, задачи и используемой модели.

**Токенизация в обработке естественного языка (NLP)**

**Введение в токенизацию**

Токенизация — это фундаментальный процесс в обработке естественного языка (NLP), который включает в себя разбиение непрерывного текста на более мелкие единицы, называемые *токенами*. Эти токены могут быть словами, подсловами, символами или даже предложениями, в зависимости от контекста и цели анализа. Токенизация является первым шагом в большинстве задач NLP, поскольку она преобразует необработанный текстовый ввод в структурированный формат, который может быть обработан машиной.

**Почему токенизация важна?** Токенизация критически важна для NLP, поскольку она упрощает анализ, разбивая сложный текст на управляемые части. Токены служат основными признаками для моделей машинного обучения и помогают стандартизировать текстовые данные для дальнейшей обработки, такой как лемматизация или стемминг. Кроме того, токенизация способствует снижению размерности, обрабатывая повторяющиеся слова как отдельные сущности.

В этом разделе подробно рассмотрим различные методы токенизации, их принципы работы и особенности, проиллюстрировав каждый конкретными примерами.

**1. Токенизация по пробелам (Whitespace Tokenization)**

Токенизация по пробелам представляет собой наиболее простой и интуитивно понятный метод токенизации, основанный на базовом предположении, что слова в тексте разделены пробельными символами. Этот подход заключается в элементарном разбиении строки текста на подстроки всякий раз, когда встречается один или несколько пробельных символов, таких как пробелы, табуляции или символы новой строки. Каждый полученный сегмент затем рассматривается как отдельный токен.

Простота и скорость являются ключевыми характеристиками этого метода. Его реализация чрезвычайно легка, а вычислительная эффективность делает его привлекательным для задач, где производительность является приоритетом, а требования к точности токенизации невысоки. Он хорошо подходит для языков, в которых слова традиционно разделяются пробелами, например, для большинства европейских языков.

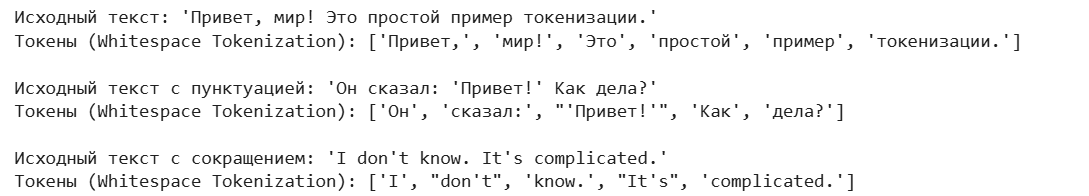
Например, строка "Привет, мир!" при токенизации по пробелам будет разбита на токены ["Привет,", "мир!"]. Как видно, пунктуация остается прикрепленной к словам, поскольку пробел является единственным разделителем.

Однако, несмотря на свою простоту, токенизация по пробелам обладает существенными ограничениями. Она не способна корректно обрабатывать пунктуацию; знаки препинания (точки, запятые, вопросительные знаки и т.д.) часто остаются прикрепленными к словам (например, "word." вместо отдельных "word" и "."), что приводит к формированию некорректных или неоднозначных токенов. Аналогично, сокращения (например, "don't", "I'm") не разбиваются на составляющие их слова, что может быть нежелательно для многих задач NLP, требующих более гранулярного анализа. Метод также не учитывает различия в регистре (например, "Word" и "word" будут рассматриваться как разные токены) и абсолютно неприменим для языков, которые не используют пробелы для разделения слов, таких как китайский, японский или тайский, где слова могут следовать друг за другом без явных разделителей.

Пример 1: Токенизация по пробелам

|  |
| --- |
| text = "Привет, мир! Это простой пример токенизации."  tokens = text.split()  print(f"Исходный текст: '{text}'")  print(f"Токены (Whitespace Tokenization): {tokens}")  text\_with\_punctuation = "Он сказал: 'Привет!' Как дела?"  tokens\_punct = text\_with\_punctuation.split()  print(f"\nИсходный текст с пунктуацией: '{text\_with\_punctuation}'")  print(f"Токены (Whitespace Tokenization): {tokens\_punct}")  text\_with\_contraction = "I don't know. It's complicated."  tokens\_contraction = text\_with\_contraction.split()  print(f"\nИсходный текст с сокращением: '{text\_with\_contraction}'")  print(f"Токены (Whitespace Tokenization): {tokens\_contraction}") |

Результат:



Как видно из примера, пунктуация остается прикрепленной к словам, а сокращения не разделяются.

### **2. Токенизация с использованием регулярных выражений (Regex Tokenization)**

Токенизация с использованием регулярных выражений представляет собой значительно более мощный и гибкий подход по сравнению с простым разбиением по пробелам. Этот метод позволяет разработчику определять сложные шаблоны (регулярные выражения) для точного определения того, что должно быть токеном, или, наоборот, что должно служить разделителем между токенами. Вместо автоматического разбиения по пробелам, можно создать шаблон, который соответствует желаемым единицам текста, таким как слова, числа, знаки пунктуации, и извлекать их, или же шаблон, который указывает на последовательности символов, по которым текст следует разбить.

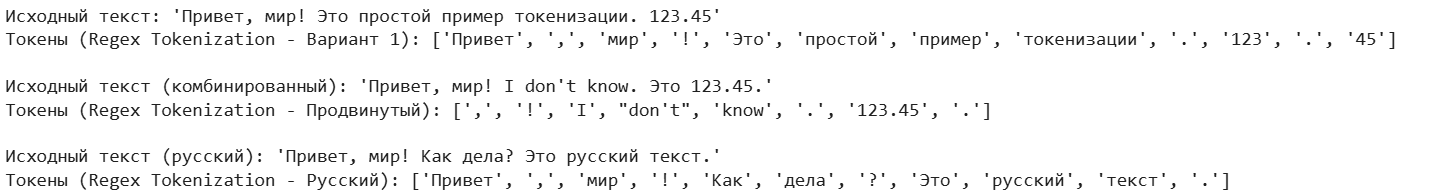
Основное преимущество токенизации с использованием регулярных выражений заключается в её исключительной гибкости и настраиваемости. Она предоставляет точный контроль над процессом, позволяя включать или исключать из токенов пунктуацию, числа, специальные символы и другие элементы в соответствии с требованиями конкретной задачи. Это делает её способной эффективно обрабатывать более сложные случаи, такие как сокращения, составные слова, числа с десятичными разделителями и другие нестандартные структуры, которые не поддаются простым методам токенизации.

Однако, несмотря на свою мощь, этот метод имеет свои сложности. Создание эффективных и безошибочных регулярных выражений может быть нетривиальной задачей, требующей глубокого понимания синтаксиса регулярных выражений. Более сложные шаблоны могут также привести к снижению производительности по сравнению с базовой токенизацией по пробелам, особенно при обработке очень больших объемов текста. Кроме того, регулярные выражения, разработанные для одного языка, могут оказаться неэффективными или некорректными для другого, поскольку они не учитывают лингвистический контекст. Это может привести к ошибкам в неоднозначных ситуациях, например, когда точка может быть как концом предложения, так и десятичным разделителем в числе.

Пример 2: Токенизация с использованием регулярных выражений

|  |
| --- |
| import re  # Регулярное выражение для извлечения слов и чисел, игнорируя пунктуацию  # \b - граница слова  # \w+ - один или более буквенно-цифровых символов (буквы, цифры, подчеркивание)  # \d+ - один или более цифр  # [^\w\s] - любой символ, который не является буквенно-цифровым и не является пробелом (т.е. пунктуация)  # | - ИЛИ  text = "Привет, мир! Это простой пример токенизации. 123.45"  # Вариант 1: Извлекаем слова и числа, пунктуацию отдельно  # pattern = r'\b\w+\b|\d+\.\d+|\S' # Более сложный шаблон для чисел и пунктуации  pattern = r"\b\w+\b|\d+\.\d+|[^\w\s]" # Извлекает слова, числа с десятичной точкой и отдельные знаки пунктуации  tokens = re.findall(pattern, text)  print(f"Исходный текст: '{text}'")  print(f"Токены (Regex Tokenization - Вариант 1): {tokens}")  # Более продвинутый Regex для слов, чисел и пунктуации,  # который также пытается сохранить сокращения.  # \w+ - слова, \d+ - числа, \s+ - пробелы, [.,!?;:] - пунктуация  # (?:[a-zA-Z]+(?:'[a-zA-Z]+)?|\d+(?:\.\d+)?|[.,!?;:])  # Это регулярное выражение пытается захватить:  # 1. Слова, возможно, с апострофом (например, don't, It's)  # 2. Числа, возможно, с десятичной точкой (например, 123, 123.45)  # 3. Отдельные знаки пунктуации  pattern\_advanced = r"[a-zA-Z]+(?:'[a-zA-Z]+)?|\d+(?:\.\d+)?|[.,!?;:]"  text\_combined = "Привет, мир! I don't know. Это 123.45."  tokens\_advanced = re.findall(pattern\_advanced, text\_combined, re.IGNORECASE | re.UNICODE)  # re.IGNORECASE для игнорирования регистра, re.UNICODE для работы с юникодом (русские буквы)  print(f"\nИсходный текст (комбинированный): '{text\_combined}'")  print(f"Токены (Regex Tokenization - Продвинутый): {tokens\_advanced}")  # Для русского текста лучше использовать более общий шаблон для слов  # или специализированные библиотеки.  # \b - граница слова, \w+ - буквенно-цифровые символы (включая русские буквы при re.UNICODE)  pattern\_russian = r'\b\w+\b|[.,!?;:]'  text\_russian = "Привет, мир! Как дела? Это русский текст."  tokens\_russian = re.findall(pattern\_russian, text\_russian, re.UNICODE)  print(f"\nИсходный текст (русский): '{text\_russian}'")  print(f"Токены (Regex Tokenization - Русский): {tokens\_russian}") |

Результат:



Как видно, регулярные выражения позволяют более тонко настраивать процесс токенизации, отделяя пунктуацию и сохраняя сокращения, если это необходимо.

### **3. Токенизация с использованием NLTK (Natural Language Toolkit)**

NLTK — это мощная библиотека Python, широко используемая для работы с человеческим языком. Она предоставляет обширный набор инструментов для токенизации, включая специализированные токенизаторы как для слов, так и для предложений. Для эффективного использования NLTK необходимо установить саму библиотеку и загрузить дополнительные данные, такие как модель punkt, которая является основой для токенизации предложений и слов.

|  |
| --- |
| # Установка NLTK (если еще не установлено)  # pip install nltk  # Загрузка необходимых данных NLTK  import nltk  try:      nltk.data.find('tokenizers/punkt')  except LookupError:      nltk.download('punkt')  try:      nltk.data.find('tokenizers/punkt\_tab')  except LookupError:      nltk.download('punkt\_tab') |

Где:

* **punkt**: Это предобученная модель для токенизации предложений и слов. Она использует статистический, неконтролируемый алгоритм для определения границ предложений, распознавания аббревиатур, коллокаций и слов, которые часто начинают предложения. Модель punkt обучается на большом корпусе текста и способна адаптироваться к особенностям различных языков. Она является основным компонентом для корректной работы sent\_tokenize и word\_tokenize.
* **punkt\_tab**: Этот ресурс является дополнительным компонентом, который может потребоваться для некоторых специфических реализаций токенизаторов NLTK, особенно для word\_tokenize. Он содержит таблицы или данные, используемые алгоритмом Punkt для более точной токенизации слов, включая обработку сложных случаев и исключений, которые могут быть специфичны для определенных языков или контекстов.

**3.1. nltk.word\_tokenize**

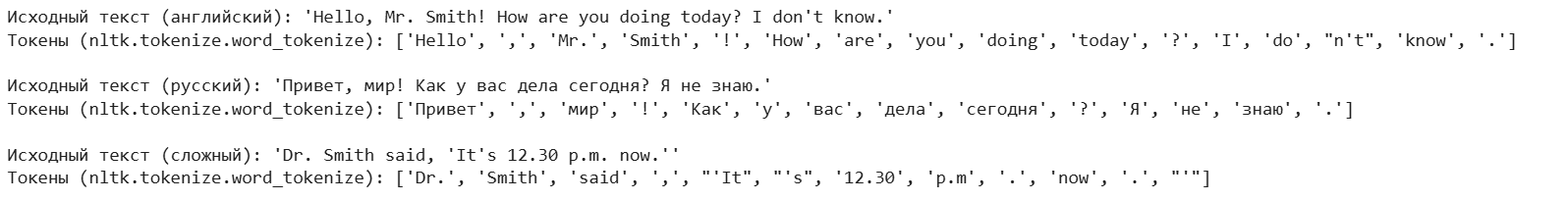
Метод nltk.word\_tokenize представляет собой токенизатор слов, который использует предобученный алгоритм Punkt. Этот алгоритм, обученный на обширном корпусе текста, предназначен для интеллектуального определения границ слов. В отличие от более простых подходов, word\_tokenize эффективно справляется с пунктуацией и некоторыми типами сокращений. Он способен отделять знаки препинания от слов (например, "Hello." будет разбито на "Hello" и "."), а также обрабатывать апострофы в сокращениях, разбивая их на более мелкие смысловые единицы (например, "don't" будет преобразовано в "do" и "n't").

Высокая точность является одним из ключевых преимуществ nltk.word\_tokenize, особенно по сравнению с токенизацией по пробелам или простыми регулярными выражениями, что делает его предпочтительным выбором для английского языка. Он хорошо справляется с отделением пунктуации и умеет разбивать распространенные сокращения. Кроме того, модель Punkt может быть адаптирована и обучена для работы с различными языками, что расширяет его применимость. Однако использование nltk.word\_tokenize требует предварительной загрузки предобученной модели punkt, что добавляет зависимость и может создавать небольшой оверхед. Иногда этот токенизатор может проявлять излишнюю агрессивность в разбиении токенов, которые должны оставаться единым целым (например, "U.S.A." может быть разбито на отдельные символы). Также его производительность может варьироваться для языков, чьи правила токенизации значительно отличаются от английского.

Пример 3: word\_tokenize

|  |
| --- |
| from nltk.tokenize import word\_tokenize  text\_en = "Hello, Mr. Smith! How are you doing today? I don't know."  tokens\_en = word\_tokenize(text\_en)  print(f"Исходный текст (английский): '{text\_en}'")  print(f"Токены (nltk.tokenize.word\_tokenize): {tokens\_en}")  text\_ru = "Привет, мир! Как у вас дела сегодня? Я не знаю."  tokens\_ru = word\_tokenize(text\_ru)  print(f"\nИсходный текст (русский): '{text\_ru}'")  print(f"Токены (nltk.tokenize.word\_tokenize): {tokens\_ru}")  text\_complex = "Dr. Smith said, 'It's 12.30 p.m. now.'"  tokens\_complex = word\_tokenize(text\_complex)  print(f"\nИсходный текст (сложный): '{text\_complex}'")  print(f"Токены (nltk.tokenize.word\_tokenize): {tokens\_complex}") |

**Результат:**



NLTK word\_tokenize демонстрирует хорошее разделение пунктуации и умение обрабатывать сокращения, хотя иногда может быть излишне агрессивен (например, "'It's" разбивается на "'It" и "'s").

**3.2. nltk.sent\_tokenize**

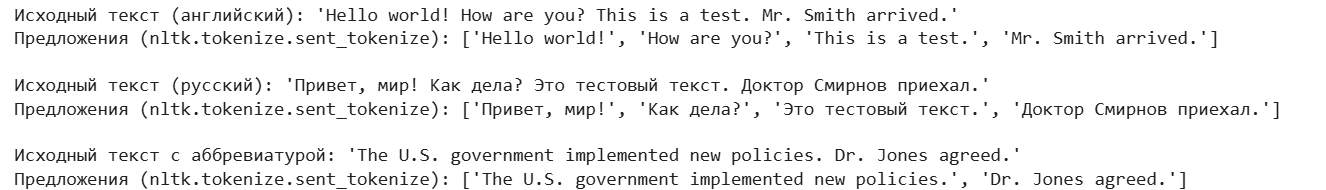
Метод nltk.sent\_tokenize предназначен для разбиения текста на отдельные предложения. Он также использует предобученную модель Punkt, но сфокусирован на идентификации концов предложений. Алгоритм ищет знаки пунктуации, такие как точки, вопросительные знаки и восклицательные знаки, и комбинирует их с различными эвристиками, например, анализом регистра следующего слова или наличия аббревиатур, чтобы точно определить, где заканчивается одно предложение и начинается другое.

Способность sent\_tokenize точно определять границы предложений является его ключевым преимуществом. Он эффективно справляется с неоднозначностью использования точек, например, после аббревиатур или в числах, что является частой проблемой для более простых методов. Токенизация предложений является критически важным первым шагом для многих последующих задач NLP, включая извлечение информации, машинный перевод и суммаризацию текста. Модель Punkt может быть обучена для различных языков, что позволяет адаптировать токенизатор к специфическим правилам каждого языка. Однако, как и в случае с word\_tokenize, для работы sent\_tokenize требуется загрузка модели punkt, что добавляет зависимость. Производительность токенизатора может варьироваться в зависимости от языка, а также он может испытывать трудности с очень сложными или нестандартными структурами предложений, особенно в неформальном тексте, таком как сообщения в социальных сетях.

Пример 3.2: sent\_tokenize

|  |
| --- |
| from nltk.tokenize import sent\_tokenize  text\_multi\_sentence\_en = "Hello world! How are you? This is a test. Mr. Smith arrived."  sentences\_en = sent\_tokenize(text\_multi\_sentence\_en)  print(f"Исходный текст (английский): '{text\_multi\_sentence\_en}'")  print(f"Предложения (nltk.tokenize.sent\_tokenize): {sentences\_en}")  text\_multi\_sentence\_ru = "Привет, мир! Как дела? Это тестовый текст. Доктор Смирнов приехал."  sentences\_ru = sent\_tokenize(text\_multi\_sentence\_ru)  print(f"\nИсходный текст (русский): '{text\_multi\_sentence\_ru}'")  print(f"Предложения (nltk.tokenize.sent\_tokenize): {sentences\_ru}")  text\_with\_abbreviation = "The U.S. government implemented new policies. Dr. Jones agreed."  sentences\_abbrev = sent\_tokenize(text\_with\_abbreviation)  print(f"\nИсходный текст с аббревиатурой: '{text\_with\_abbreviation}'")  print(f"Предложения (nltk.tokenize.sent\_tokenize): {sentences\_abbrev}") |

**Результат:**

****

nltk.sent\_tokenize успешно определяет границы предложений, даже когда точка используется в аббревиатурах, что является распространенной проблемой для более простых методов.

### 4. Токенизация с использованием spaCy

spaCy — это современная и высокопроизводительная библиотека NLP для Python, разработанная с учетом требований производственных систем. Её токенизатор является одним из самых продвинутых и быстрых, принципиально отличаясь от простых методов тем, что он работает на основе набора правил, специфичных для каждого языка, а не только на пробелах или регулярных выражениях. Токенизатор spaCy использует комбинацию правил для префиксов, суффиксов и инфиксов для отделения пунктуации, чисел и символов. Кроме того, он включает в себя набор исключений, специально разработанных для корректной обработки особых случаев, таких как сокращения (например, "don't", "U.S."), эмодзи, URL-адреса и другие нестандартные текстовые конструкции. Важно отметить, что, хотя сам токенизатор является правилосообразным, он глубоко интегрирован с языковыми моделями spaCy, которые могут быть использованы для выполнения более сложных задач после токенизации, таких как POS-тегирование или распознавание именованных сущностей.

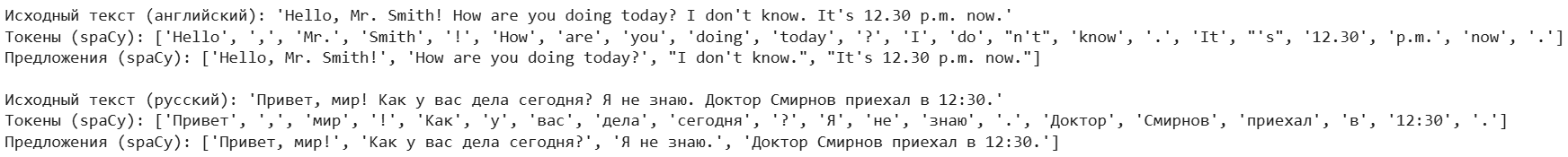
Для использования spaCy необходимо установить библиотеку и загрузить соответствующую языковую модель (например, `en\_core\_web\_sm` для английского языка или `ru\_core\_web\_sm` для русского).

spaCy предлагает один из самых точных токенизаторов, способный обрабатывать широкий спектр сложных сценариев, включая сокращения, URL-адреса, хэштеги и эмодзи, с высокой степенью надежности. Его исключительная скорость делает его идеальным выбором для обработки больших объемов текстовых данных в производственных условиях. Библиотека поставляется с предопределенными и настраиваемыми правилами для множества языков, что обеспечивает оптимальную производительность для каждого из них. Токенизация в spaCy является первым и неотъемлемым шагом в её комплексном конвейере обработки NLP, что значительно упрощает переход к последующим этапам, таким как лемматизация, морфологический анализ и синтаксический разбор. Более того, токенизатор spaCy способен выделять не только слова, но и числа, знаки пунктуации и другие символы, что делает его чрезвычайно универсальным инструментом. Однако, несмотря на все его преимущества, использование spaCy требует загрузки языковой модели, которая может быть довольно объемной. Хотя правила токенизации можно настраивать, создание полностью собственного токенизатора с нуля в рамках spaCy является более сложной задачей, чем простое написание регулярного выражения. Для очень простых задач, где достаточно базовой токенизации по пробелам, использование spaCy может быть избыточным из-за необходимости загрузки и инициализации модели.

Пример 4.1: Токенизация с использованием spaCy (английский)

|  |
| --- |
| !python -m spacy download ru\_core\_news\_sm  import spacy  # Загрузка английской модели (если еще не загружена)  # python -m spacy download en\_core\_web\_sm  nlp\_en = spacy.load("en\_core\_web\_sm")  # Пример 4.1: Токенизация с использованием spaCy (английский)  text\_en = "Hello, Mr. Smith! How are you doing today? I don't know. It's 12.30 p.m. now."  doc\_en = nlp\_en(text\_en)  tokens\_en = [token.text for token in doc\_en]  sentences\_en = [sent.text for sent in doc\_en.sents] # spaCy также умеет токенизировать предложения  print(f"Исходный текст (английский): '{text\_en}'")  print(f"Токены (spaCy): {tokens\_en}")  print(f"Предложения (spaCy): {sentences\_en}")  # Загрузка русской модели (если еще не загружена)  # python -m spacy download ru\_core\_news\_sm  nlp\_ru = spacy.load("ru\_core\_news\_sm")  # Пример 4.2: Токенизация с использованием spaCy (русский)  text\_ru = "Привет, мир! Как у вас дела сегодня? Я не знаю. Доктор Смирнов приехал в 12:30."  doc\_ru = nlp\_ru(text\_ru)  tokens\_ru = [token.text for token in doc\_ru]  sentences\_ru = [sent.text for sent in doc\_ru.sents]  print(f"\nИсходный текст (русский): '{text\_ru}'")  print(f"Токены (spaCy): {tokens\_ru}")  print(f"Предложения (spaCy): {sentences\_ru}") |

Результат



spaCy демонстрирует очень точную токенизацию, правильно разделяя пунктуацию, сокращения (например, "don't" на "do" и "n't", "It's" на "It" и "'s") и обрабатывая числа с десятичными точками и время. Он также эффективно токенизирует предложения.

**Токенизация по символам (Character Tokenization)**

На самом гранулярном уровне токенизации каждый символ в тексте рассматривается как отдельный токен. Например, слово "Привет" будет разбито на ["П", "р", "и", "в", "е", "т"].

Этот подход находит применение в следующих случаях:

* **Языки без явных границ слов**: Например, китайский или японский, где слова не разделяются пробелами.
* **Задачи, чувствительные к опечаткам**: Токенизация по символам позволяет моделям работать на уровне отдельных символов, что делает их более устойчивыми к ошибкам ввода.
* **Модели, работающие с морфологией**: Иногда используется для захвата тонких морфологических особенностей языка.

Основное ограничение токенизации по символам заключается в создании очень длинных последовательностей токенов, что увеличивает вычислительные затраты и может затруднить захват высокоуровневых семантических зависимостей.

## **§4. Subword-токенизация (Subword Tokenization)**

Subword-токенизация представляет собой компромисс между токенизацией по словам и по символам. Этот подход позволяет разбивать редкие или неизвестные слова на более мелкие, часто встречающиеся подслова (или морфемы), в то время как частые слова остаются неизменными. Это эффективно решает проблему слов вне словаря (Out-of-Vocabulary, OOV) и значительно сокращает размер словаря по сравнению с токенизацией по словам, при этом сохраняя часть семантической информации.

Ключевые преимущества subword-токенизации включают:

* **Обработка OOV-слов**: Любое слово может быть разбито на известные подслова, что позволяет моделям работать с ранее не встречавшимися словами.
* **Уменьшение размера словаря**: Размер словаря подслов значительно меньше, чем словарь символов, и более управляем, чем словарь полных слов.
* **Сохранение семантики**: Подслова часто несут часть значения исходного слова, что способствует лучшему пониманию текста.

Среди основных алгоритмов subword-токенизации выделяют:

1. **Алгоритм Byte-Pair Encoding (BPE)**

BPE, изначально разработанный как алгоритм сжатия данных, был адаптирован для задач токенизации текста в НЛП. Его принцип работы основан на итеративном объединении наиболее часто встречающихся пар символов или последовательностей символов в новые, более длинные субтокены. Этот процесс позволяет моделям НЛП эффективно обрабатывать слова с различными суффиксами и префиксами, разбивая их на общие корни и аффиксы. В результате значительно сокращается размер словаря, улучшается обработка OOV-слов и облегчается усвоение тонких семантических связей моделью.

Современные высокопроизводительные модели НЛП, такие как GPT, BERT, RoBERTa и другие, активно используют BPE (или его варианты, например, WordPiece) в процессе предварительной обработки данных.

**3. Поток алгоритма BPE**

Алгоритм BPE следует итеративному процессу для построения словаря субтокенов.

**3.1. Инициализация**

1. **Определение целевого размера словаря (**V**):** Устанавливается желаемый максимальный размер словаря субтокенов. Процесс объединения будет продолжаться до достижения этого размера или до исчерпания эффективных объединений.
2. **Начальное разбиение на символы:** Все слова в обучающем корпусе разбиваются на отдельные символы.
3. **Добавление символа конца слова (</w>):** К каждому слову добавляется специальный символ </w>. Этот символ помогает отличать полные слова от частичных субтокенов и предотвращает нежелательные объединения между концом одного слова и началом следующего. Важно отметить, что в некоторых вариантах субтокенизации (например, WordPiece) могут использоваться другие специальные символы, такие как ##, для обозначения продолжения субтокена внутри слова (например, "playing" rightarrow "play", "##ing"). Однако для классического BPE символ </w> является стандартным для маркировки границ слов.
4. **Формирование начального словаря:** Начальный словарь состоит из всех уникальных символов, присутствующих в корпусе. Частота каждого слова в корпусе подсчитывается.

Пример инициализации для текста: "Ученик учится в школе, а учитель учит ученика"

**Исходный текст:** "Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика"

**Разбиение на символы с </w>:**

|  |
| --- |
| У ч е н и к </w>  у ч и т с я </w>  в </w>  ш к о л е , </w>  а </w>  у ч и т е л ь </w>  у ч и т </w>  у ч е н и к а </w> |

**Начальный корпус (с частотами):**

|  |
| --- |
| {'У ч е н и к </w>': 1,  'у ч и т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'у ч и т е л ь </w>': 1,  'у ч и т </w>': 1,  'у ч е н и к а </w>': 1} |

**Начальный словарь:** ['У', 'ч', 'е', 'н', 'и', 'к', '</w>', 'т', 'с', 'я', 'в', 'ш', 'к', 'о', 'л', 'е', ',', 'а', 'л', 'ь'] (символы 'У' и 'у' рассматриваются как различные из-за регистра).

**3.2. Итеративное объединение**

Этот процесс повторяется до тех пор, пока не будет достигнут желаемый размер словаря (V) или пока следующая наиболее частая пара не будет иметь частоту встречаемости меньше 1 (т.е. не останется пар для выгодного объединения).

1. **Поиск наиболее частой пары:** Производится сканирование всего текущего корпуса для определения пары смежных символов (или уже объединенных субтокенов), которая встречается наиболее часто.
2. **Объединение пары:** Эта наиболее частая пара объединяется в новый субтокен.
3. **Обновление корпуса:** Все вхождения этой пары в корпусе заменяются на новый объединенный субтокен.
4. **Добавление в словарь:** Новый объединенный субтокен добавляется в словарь.

**Детальный пример итераций для текста: "Ученик учится в школе, а учитель учит ученика"**

**Итерация 1:**

**Поиск всех пар и их частот:**

У ч: 1

ч е: 2 (в У ч е н и к, у ч е н и к а)

е н: 2 (в У ч е н и к, у ч е н и к а)

н и: 2 (в У ч е н и к, у ч е н и к а)

и к: 2 (в У ч е н и к, у ч е н и к а)

к </w>: 1

у ч: 3 (в у ч и т с я, у ч и т е л ь, у ч и т) - *Примечание: 'У ч' и 'у ч' рассматриваются как разные пары из-за регистра.*

ч и: 3 (в у ч и т с я, у ч и т е л ь, у ч и т)

и т: 3 (в у ч и т с я, у ч и т е л ь, у ч и т)

т с: 1

с я: 1

я </w>: 1

в </w>: 1

ш к: 1

к о: 1

о л: 1

л е: 1

е ,: 1

, </w>: 1

а </w>: 1

т е: 1

е л: 1

л ь: 1

ь </w>: 1

т </w>: 2 (в у ч и т, у ч и т е л ь)

к а: 1

а </w>: 1

**Наиболее частая пара:** у ч (3 раза) и ч и (3 раза) и и т (3 раза). Выбираем у ч.

**Объединение:** у ч уч

**Обновление корпуса:**

|  |
| --- |
| {'У ч е н и к </w>': 1,  'уч и т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'уч и т е л ь </w>': 1,  'уч и т </w>': 1,  'уч е н и к а </w>': 1} |

**Добавление в словарь:** уч

**Итерация 2:**

**Наиболее частая пара:** уч и (встречается 3 раза: в уч и т с я, уч и т е л ь, уч и т).

**Объединение:** уч и учи

**Обновление корпуса:**

|  |
| --- |
| {'У ч е н и к </w>': 1,  'учи т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учи т е л ь </w>': 1,  'учи т </w>': 1,  'уч е н и к а </w>': 1} |

**Добавление в словарь:** учи

**Итерация 3:**

**Наиболее частая пара:** н и (встречается 2 раза: в У ч е н и к, у ч е н и к а). Пара т </w> также встречается 2 раза. Выбираем н и.

**Объединение:** н и ни

**Обновление корпуса:**

|  |
| --- |
| {'У ч е ни к </w>': 1,  'учи т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учи т е л ь </w>': 1,  'учи т </w>': 1,  'уч е ни к а </w>': 1} |

**Добавление в словарь:** ни

**Итерация 4:**

**Наиболее частая пара:** т </w> (встречается 2 раза: в учи т, учи т е л ь).

**Объединение:** т </w> т</w>

**Обновление корпуса:**

|  |
| --- |
| {'У ч е ни к </w>': 1,  'учи т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учи т е л ь </w>': 1,  'учи т</w>': 1,  'уч е ни к а </w>': 1} |

**Добавление в словарь:** т</w>

**Итерация 5:**

**Наиболее частая пара:** е н (встречается 2 раза: в У ч е н и к, у ч е н и к а).

**Объединение:** е н ен

**Обновление корпуса:**

|  |
| --- |
| {'У ч ен и к </w>': 1,  'учи т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учи т е л ь </w>': 1,  'учи т</w>': 1,  'уч ен и к а </w>': 1} |

**Добавление в словарь:** ен

**Итерация 6:**

**Наиболее частая пара:** ни к (встречается 2 раза: в У ч е ни к, уч е ни к а).

**Объединение:** ни к ник

**Обновление корпуса:**

|  |
| --- |
| {'У ч е ник </w>': 1,  'учи т с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учи т е л ь </w>': 1,  'учи т</w>': 1,  'уч е ник а </w>': 1} |

**Добавление в словарь:** ник

**Итерация 7:**

**Наиболее частая пара:** учи т (встречается 2 раза: в учи т с я, учи т е л ь).

**Объединение:** учи т учит

**Обновление корпуса:**

|  |
| --- |
| {'У ч е ник </w>': 1,  'учит с я </w>': 1,  'в </w>': 1,  'ш к о л е , </w>': 1,  'а </w>': 1,  'учит е л ь </w>': 1,  'учи т</w>': 1,  'уч е ник а </w>': 1} |

**Добавление в словарь:** учит

Процесс продолжается до достижения заданного размера словаря или до тех пор, пока не останется пар с частотой > 1.

**Пример конечного словаря (фрагмент):**

После завершения итераций словарь будет содержать как отдельные символы, так и часто встречающиеся комбинации символов (субтокены). Примеры субтокенов:

* Символы: У, ч, е, н, и, к, т, с, я, в, ш, к, о, л, е, ,, а, ь, </w>
* Объединенные субтокены: уч, учи, ни, т</w>, ен, ник, учит
* Возможные **целые слова (если часто встречаются):** ученик</w>, учитель</w>, школе,</w>

**Токенизация исходного предложения с использованием полученного словаря:**

Используя построенный словарь, исходное предложение "Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика" может быть токенизировано следующим образом, выбирая самые длинные из доступных субтокенов:

|  |
| --- |
| Ученик</w>  учится</w>  в</w>  школе,</w>  а</w>  учитель</w>  учит</w>  ученика</w> |

**4. Масштабирование Byte Pair Encoding на Многопредметный Корпус**

При работе с большим корпусом текстов, состоящим из множества предложений или документов, подход BPE немного адаптируется, чтобы сохранить лингвистическую целостность и избежать создания бессмысленных токенов.

**Подход:**

* **Раздельная Обработка (или с учетом границ):** Каждое предложение (или документ) обычно обрабатывается отдельно для подсчета пар. Это означает, что пары не подсчитываются и не объединяются через границы предложений.
* **Разделители Предложений:** Часто используются специальные токены для обозначения начала (<s>) и конца (</s>) предложений. Эти токены помогают модели понимать структуру текста и предотвращают слияния, которые могли бы нарушить границы предложений.
* **Без Межпредложенческих Слияний:** Это критически важно. Например, если конец одного предложения заканчивается на "ок", а начало следующего начинается на "но", BPE не будет объединять "ок" и "но" в "окно", если они находятся в разных предложениях. Это гарантирует, что создаваемые токены имеют лингвистический смысл в рамках одного предложения.

**Пример.** Давайте предположим, что следующие предложения представляют наш корпус для обучения BPE:

1. "Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика."
2. "Новый\_студент\_изучает\_грамматику."
3. "Учитель\_и\_студент\_читают\_книги."

**Начальная Настройка:**

Сначала мы добавим специальные токены для границ предложений: <s> для начала и </s> для конца.

**Токенизированный корпус (начальный символьный уровень):**

<s>\_У\_ч\_е\_н\_и\_к\_ \_у\_ч\_и\_т\_с\_я\_ \_в\_ \_ш\_к\_о\_л\_е\_,\_ \_а\_ \_у\_ч\_и\_т\_е\_л\_ь\_ \_у\_ч\_и\_т\_ \_у\_ч\_е\_н\_и\_к\_а\_.\_</s> <s>\_Н\_о\_в\_ы\_й\_ \_с\_т\_у\_д\_е\_н\_т\_ \_и\_з\_у\_ч\_а\_е\_т\_ \_г\_р\_а\_м\_м\_а\_т\_и\_к\_у\_.\_</s> <s>\_У\_ч\_и\_т\_е\_л\_ь\_ \_и\_ \_с\_т\_у\_д\_е\_н\_т\_ \_ч\_и\_т\_а\_ю\_т\_ \_к\_н\_и\_г\_и\_.\_</s>

**Итерации BPE на Корпусе:**

Теперь, когда мы подсчитываем пары, мы делаем это внутри каждого предложения отдельно, а затем суммируем частоты по всему корпусу.

**Пример подсчета пар (гипотетический):**

* Из Предложения 1: ('у', 'ч'): 5, ('и', 'к'): 2, ('\_', 'у'): 3, ('а', '\_'): 2, ...
* Из Предложения 2: ('н', 'о'): 1, ('с', 'т'): 1, ('т', 'у'): 1, ('г', 'р'): 1, ...
* Из Предложения **3:** ('у', 'ч'): 1, ('т', 'е'): 2, ('ч', 'и'): 1, ('и', '\_'): 1, ('к', 'н'): 1, ...

**Объединенные частоты (топ-пары по всему корпусу):**

|  |
| --- |
| ('у', 'ч'): 5 (из П1) + 1 (из П3) = 6  ('\_', 'у'): 3 (из П1) + 1 (из П3) = 4  ('и', 'к'): 2 (из П1) = 2  ('а', '\_'): 2 (из П1) = 2  ('т', 'е'): 1 (из П1) + 2 (из П3) = 3 |

...и так далее для всех пар.

**Решение об Объединении:** В этом случае, ('у', 'ч') с частотой 6 будет наиболее частой парой. Мы объединим ее в токен 'уч'. Затем мы обновим каждое предложение в корпусе, заменяя все вхождения ('у', 'ч') на 'уч'. Этот процесс повторяется.

**Почему Этот Подход Важен:**

* **Осмысленные токены:** Не объединяя пары через границы предложений, мы предотвращаем создание токенов, которые не представляют собой осмысленные лингвистические единицы. Это помогает сохранять семантику и синтаксис текста.
* **Целостность предложений:** Этот подход сохраняет целостность отдельных предложений, что критически важно для многих последующих задач НЛП, таких как машинный перевод, где структура предложения имеет первостепенное значение.
* **Гибкость:** Он позволяет обрабатывать потоковый текст или очень большие корпуса, где не все предложения доступны одновременно. Это делает BPE масштабируемым решением.
* **Последовательная токенизация:** Гарантирует, что одно и то же слово или субслово будет токенизировано одинаково, независимо от окружающего контекста, при условии, что оно находится в пределах одной границы предложения.

**5. Проблемы Стандартного BPE и Их Улучшения**

Хотя BPE является мощным и широко используемым алгоритмом, у него есть свои ограничения, которые привели к разработке различных вариантов и улучшений.

**Проблемы Стандартного BPE:**

* **Жадный и Детерминированный:** BPE всегда выбирает наиболее частую пару на каждой итерации. Этот "жадный" подход может не всегда быть оптимальным для конечной задачи НЛП, так как он не учитывает семантический или морфологический контекст. Он создает фиксированный набор слияний, которые не могут быть изменены после обучения.
* **Обработка Несловарных Слов (OOV):** Хотя BPE значительно лучше справляется с OOV-словами, чем токенизация на уровне слов, он все еще может испытывать трудности с очень редкими словами, уникальными написаниями или словами из совершенно новых доменов, которые могут разбиваться на слишком мелкие, менее осмысленные единицы.
* **Отсутствие Морфологической Осведомленности:** BPE не имеет встроенного понимания лингвистической структуры (морфологии, грамматики). Он просто ищет статистически частые последовательности символов. Это может привести к неоптимальным субсловесным разбиениям, которые не соответствуют естественным морфемам языка.
* **Фиксированный Словарь:** После обучения словарь BPE является статичным. Если язык эволюционирует, появляются новые слова или модель применяется к новому домену, словарь может стать менее эффективным, требуя переобучения.
* **Вычислительная Сложность:** Для очень больших корпусов итеративный процесс подсчета пар и слияния может быть ресурсоемким и занимать много времени.
* **Неоднозначность Сегментации:** Одно и то же слово может быть потенциально токенизировано по-разному в зависимости от того, какие слияния были выбраны на более ранних этапах обучения, что может привести к некоторой потере согласованности.

**Преимущества BPE:**

* BPE эффективно решает проблему OOV-слов, поскольку может разбивать редкие или неизвестные слова на известные компоненты, которые уже есть в словаре.
* Алгоритм позволяет управлять размером словаря, создавая его путем объединения частых пар символов или субслов.
* Он обеспечивает хороший баланс между символьным и словесным уровнем токенизации: частые слова могут быть представлены как единые токены, а редкие — как комбинации подслов.
* Особого внимания заслуживает байтовый BPE (Byte-level BPE), который может обрабатывать любой ввод, включая не-юникодовый текст, и гарантирует отсутствие неизвестных токенов, что делает его универсальным и надежным.

**Недостатки BPE:**

* BPE является жадным и детерминированным алгоритмом: он всегда выбирает наиболее частую пару для слияния, что не всегда приводит к оптимальной лингвистической сегментации.
* Алгоритм не обладает морфологическим пониманием и не учитывает лингвистические структуры (приставки, суффиксы, корни в лингвистическом смысле), что может приводить к неоптимальным разбиениям, не соответствующим естественной морфологии языка.
* Для очень больших корпусов итеративный процесс подсчета и слияния может быть вычислительно интенсивным и времязатратным.
* Стандартный BPE с предварительной токенизацией может приводить к искаженному распределению токенов, что ограничивает преимущества больших словарей.

BPE стал краеугольным камнем современной токенизации для больших языковых моделей таких как GPT и RoBERTa, но его эволюция (например, до Byte-level BPE) демонстрирует постоянный поиск баланса между лингвистической точностью, универсальностью и вычислительной эффективностью. Изначально BPE был разработан как метод сжатия данных, но его способность эффективно справляться с OOV-словами и управлять размером словаря сделала его идеальным для задач НЛП. Однако его жадный характер и отсутствие лингвистического понимания привели к появлению модификаций, таких как BoundlessBPE, которые пытаются улучшить лингвистическую когерентность токенов. Byte-level BPE — это еще одна важная адаптация, которая жертвует некоторой читаемостью токенов ради полной универсальности и отсутствия OOV, что критически важно для мультиязычных моделей и обработки "сырых" данных. Эта постоянная адаптация показывает, что нет единого "идеального" токенизатора, и выбор метода всегда зависит от конкретных требований к модели и характеристик данных.

Реализация на Питон.

|  |
| --- |
| import collections  class BPE:      """      Реализация алгоритма Byte-Pair Encoding (BPE) для токенизации текста.      """      def \_\_init\_\_(self, vocab\_size=None, num\_merges=None):          """          Инициализирует BPE токенизатор.          :param vocab\_size: Желаемый максимальный размер словаря субтокенов.                             Если указан, num\_merges будет игнорироваться.          :param num\_merges: Количество итераций объединения пар.                             Если vocab\_size не указан, будет использоваться это значение.          """          if vocab\_size is None and num\_merges is None:              raise ValueError("Необходимо указать либо vocab\_size, либо num\_merges.")          self.vocab\_size = vocab\_size          self.num\_merges = num\_merges          self.merges = {}  # Словарь для хранения выполненных объединений {pair: new\_token}          self.vocabulary = set() # Текущий словарь субтокенов          self.token\_to\_id = {} # Словарь для преобразования токенов в ID          self.id\_to\_token = [] # Список для преобразования ID в токены      def \_get\_initial\_tokens(self, text):          """          Разбивает текст на начальные символьные токены и добавляет символ конца слова.          :param text: Входная строка.          :return: Список списков символьных токенов для каждого слова.          """          # Заменяем подчеркивания на пробелы для разделения слов, затем разбиваем по пробелам          words = text.replace('\_', ' ').split()          initial\_tokens = []          for word in words:              # Добавляем символ конца слова к каждому слову              initial\_tokens.append(list(word) + ['</w>'])          return initial\_tokens      def \_get\_pair\_counts(self, tokenized\_corpus):          """          Подсчитывает частоту встречаемости всех смежных пар токенов в корпусе.          :param tokenized\_corpus: Список списков токенов (представляющих слова).          :return: Словарь с частотами пар {('token1', 'token2'): count}.          """          pair\_counts = collections.defaultdict(int)          for word\_tokens in tokenized\_corpus:              # Итерируем по парам токенов в каждом слове              for i in range(len(word\_tokens) - 1):                  pair\_counts[(word\_tokens[i], word\_tokens[i+1])] += 1          return pair\_counts      def \_merge\_pair(self, tokenized\_corpus, pair, new\_token):          """          Объединяет все вхождения заданной пары в новый токен в корпусе.          :param tokenized\_corpus: Текущий корпус токенов.          :param pair: Пара токенов для объединения (tuple).          :param new\_token: Новый токен, который заменит пару.          :return: Обновленный корпус токенов.          """          updated\_corpus = []          for word\_tokens in tokenized\_corpus:              new\_word\_tokens = []              i = 0              while i < len(word\_tokens):                  # Проверяем, является ли текущая позиция началом искомой пары                  if i + 1 < len(word\_tokens) and (word\_tokens[i], word\_tokens[i+1]) == pair:                      new\_word\_tokens.append(new\_token)                      i += 2 # Пропускаем оба токена, которые были объединены                  else:                      new\_word\_tokens.append(word\_tokens[i])                      i += 1              updated\_corpus.append(new\_word\_tokens)          return updated\_corpus      def fit(self, corpus\_text):          """          Обучает BPE модель на заданном текстовом корпусе.          :param corpus\_text: Строка, представляющая обучающий корпус.          """          # Шаг 1: Инициализация          # Начальное разбиение на символы и добавление символа конца слова          current\_corpus\_tokens = self.\_get\_initial\_tokens(corpus\_text)          # Формирование начального словаря из всех уникальных символов          for word\_tokens in current\_corpus\_tokens:              for token in word\_tokens:                  self.vocabulary.add(token)          # Определяем количество итераций объединения          if self.vocab\_size is not None:              # Если задан размер словаря, вычисляем необходимое количество объединений              # (целевой размер словаря - текущий размер словаря)              num\_iterations = self.vocab\_size - len(self.vocabulary)          elif self.num\_merges is not None:              num\_iterations = self.num\_merges          else:              # Это условие не должно быть достигнуто благодаря проверке в \_\_init\_\_              num\_iterations = 0          # Шаг 2: Итеративное объединение          for i in range(num\_iterations):              pair\_counts = self.\_get\_pair\_counts(current\_corpus\_tokens)              if not pair\_counts: # Если нет пар для объединения, останавливаемся                  break              # Находим наиболее частую пару              # Сортируем по частоте (убывание), затем по лексикографическому порядку (возрастание)              best\_pair = max(pair\_counts, key=lambda p: (pair\_counts[p], -len(p[0]) - len(p[1]), p))              # Если частота лучшей пары меньше 1, или нет пар для объединения, останавливаемся              if pair\_counts[best\_pair] < 1:                  break              # Формируем новый токен из объединенной пары              new\_token = "".join(best\_pair)              # Сохраняем объединение              self.merges[best\_pair] = new\_token              # Обновляем корпус, заменяя все вхождения пары новым токеном              current\_corpus\_tokens = self.\_merge\_pair(current\_corpus\_tokens, best\_pair, new\_token)              # Добавляем новый токен в словарь              self.vocabulary.add(new\_token)              # Вывод результатов каждого шага              print(f"Итерация {i+1}: Объединение '{best\_pair[0]}{best\_pair[1]}' в '{new\_token}'. Частота: {pair\_counts[best\_pair]}")              print(f"Обновленный корпус: {current\_corpus\_tokens}")              print(f"Текущий словарь: {sorted(list(self.vocabulary))}\n")          # После обучения создаем отображения токенов в ID и обратно          self.id\_to\_token = sorted(list(self.vocabulary))          self.token\_to\_id = {token: i for i, token in enumerate(self.id\_to\_token)}      def encode(self, text):          """          Токенизирует новую строку, используя обученные объединения.          :param text: Строка для токенизации.          :return: Список ID токенов.          """          # Начальное разбиение текста на символы с добавлением </w>          words\_tokens = self.\_get\_initial\_tokens(text)          encoded\_tokens\_ids = []          for word\_tokens in words\_tokens:              # Для каждого слова применяем объединения в порядке их обучения              # Создаем копию списка токенов слова для модификации              current\_word\_tokens = list(word\_tokens)              # Применяем объединения итеративно, пока не сможем найти более длинный токен              # или пока не закончатся объединения              while True:                  merged\_once = False                  new\_word\_tokens = []                  i = 0                  while i < len(current\_word\_tokens):                      found\_merge = False                      # Ищем самое длинное возможное объединение, которое начинается с текущего токена                      # Проходим по всем известным объединениям в порядке их создания (от коротких к длинным)                      # или просто ищем самое длинное совпадение                      best\_match\_len = 0                      best\_match\_token = None                      # Для простоты, будем применять объединения из self.merges                      # в порядке их создания (от коротких к длинным)                      # Это не совсем оптимально, но для демонстрации подходит                      # В реальных реализациях используют более сложные структуры данных                      # для эффективного поиска наиболее длинного совпадения                      for (p1, p2), merged\_token in self.merges.items():                          if i + 1 < len(current\_word\_tokens) and \                             current\_word\_tokens[i] == p1 and \                             current\_word\_tokens[i+1] == p2:                              # Проверяем, является ли это объединение частью более длинного                              # уже существующего в словаре токена                              if len(merged\_token) > best\_match\_len:                                  best\_match\_len = len(merged\_token)                                  best\_match\_token = merged\_token                                  found\_merge = True                      if found\_merge:                          new\_word\_tokens.append(best\_match\_token)                          i += 2                          merged\_once = True                      else:                          new\_word\_tokens.append(current\_word\_tokens[i])                          i += 1                  if not merged\_once:                      break # Больше нет объединений для этого слова                  current\_word\_tokens = new\_word\_tokens              # Преобразуем токены слова в их ID              for token in current\_word\_tokens:                  if token in self.token\_to\_id:                      encoded\_tokens\_ids.append(self.token\_to\_id[token])                  else:                      # Если токен не найден (например, из-за OOV или неполного обучения),                      # разбиваем его на символы и добавляем их ID                      for char in token:                          if char in self.token\_to\_id:                              encoded\_tokens\_ids.append(self.token\_to\_id[char])                          else:                              # Этого не должно произойти, если начальный словарь содержит все символы                              print(f"Предупреждение: Символ '{char}' не найден в словаре.")                              # Можно добавить токен для неизвестных символов, например, <unk>          return encoded\_tokens\_ids      def decode(self, token\_ids):          """          Декодирует последовательность ID токенов обратно в строку.          :param token\_ids: Список ID токенов.          :return: Декодированная строка.          """          decoded\_tokens = []          for token\_id in token\_ids:              if token\_id < len(self.id\_to\_token):                  decoded\_tokens.append(self.id\_to\_token[token\_id])              else:                  print(f"Предупреждение: ID токена {token\_id} вне диапазона словаря.")                  decoded\_tokens.append('') # Или можно использовать <unk>          # Объединяем токены, убираем символ конца слова и заменяем пробелы          decoded\_text = "".join(decoded\_tokens).replace('</w>', ' ').strip()          return decoded\_text.replace(' ', '\_') # Возвращаем исходный формат с подчеркиваниями  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":      corpus = "Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика"      # Создаем экземпляр BPE, указывая количество объединений      # Можно также указать vocab\_size=X для желаемого размера словаря      bpe\_model = BPE(num\_merges=10) # Выполним 10 итераций объединения      print("--- Обучение BPE модели ---")      bpe\_model.fit(corpus)      print("\n--- Обучение завершено ---")      print(f"Финальный словарь BPE (отсортированный): {sorted(list(bpe\_model.vocabulary))}")      print(f"Выполненные объединения: {bpe\_model.merges}")      # Тестирование токенизации      print("\n--- Тестирование токенизации ---")      text\_to\_encode = "Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика"      encoded\_ids = bpe\_model.encode(text\_to\_encode)      print(f"Исходный текст: '{text\_to\_encode}'")      print(f"Закодированные ID: {encoded\_ids}")      # Декодирование      decoded\_text = bpe\_model.decode(encoded\_ids)      print(f"Декодированный текст: '{decoded\_text}'")      # Проверка на новое слово (OOV)      print("\n--- Тестирование OOV слова ---")      oov\_text = "Учительница\_учит"      encoded\_oov\_ids = bpe\_model.encode(oov\_text)      print(f"OOV текст: '{oov\_text}'")      print(f"Закодированные ID OOV: {encoded\_oov\_ids}")      decoded\_oov\_text = bpe\_model.decode(encoded\_oov\_ids)      print(f"Декодированный OOV текст: '{decoded\_oov\_text}'")      # Пример токенов по ID      print("\n--- Токены по ID ---")      for i, token in enumerate(bpe\_model.id\_to\_token):          print(f"ID {i}: '{token}'") |

Результат

--- Обучение BPE модели ---

Итерация 1: Объединение 'уч' в 'уч'. Частота: 4

Обновленный корпус: [['У', 'ч', 'е', 'н', 'и', 'к', '</w>'], ['уч', 'и', 'т', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['уч', 'и', 'т', 'е', 'л', 'ь', '</w>'], ['уч', 'и', 'т', '</w>'], ['уч', 'е', 'н', 'и', 'к', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'и', 'к', 'л', 'н', 'о', 'с', 'т', 'у', 'уч', 'ч', 'ш', 'ь', 'я']

Итерация 2: Объединение 'ит' в 'ит'. Частота: 3

Обновленный корпус: [['У', 'ч', 'е', 'н', 'и', 'к', '</w>'], ['уч', 'ит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['уч', 'ит', 'е', 'л', 'ь', '</w>'], ['уч', 'ит', '</w>'], ['уч', 'е', 'н', 'и', 'к', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'и', 'ит', 'к', 'л', 'н', 'о', 'с', 'т', 'у', 'уч', 'ч', 'ш', 'ь', 'я']

Итерация 3: Объединение 'учит' в 'учит'. Частота: 3

Обновленный корпус: [['У', 'ч', 'е', 'н', 'и', 'к', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'е', 'н', 'и', 'к', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'и', 'ит', 'к', 'л', 'н', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'ь', 'я']

Итерация 4: Объединение 'ни' в 'ни'. Частота: 2

Обновленный корпус: [['У', 'ч', 'е', 'ни', 'к', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'е', 'ни', 'к', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'и', 'ит', 'к', 'л', 'н', 'ни', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'ь', 'я']

Итерация 5: Объединение 'ник' в 'ник'. Частота: 2

Обновленный корпус: [['У', 'ч', 'е', 'ник', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'е', 'ник', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'ь', 'я']

Итерация 6: Объединение 'еник' в 'еник'. Частота: 2

Обновленный корпус: [['У', 'ч', 'еник', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а', '</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'еник', 'а', '</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'ь', 'я']

Итерация 7: Объединение 'а</w>' в 'а</w>'. Частота: 2

Обновленный корпус: [['У', 'ч', 'еник', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['ш', 'к', 'о', 'л', 'е', ',', '</w>'], ['а</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'еник', 'а</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'а</w>', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'ь', 'я']

Итерация 8: Объединение 'шк' в 'шк'. Частота: 1

Обновленный корпус: [['У', 'ч', 'еник', '</w>'], ['учит', 'с', 'я', '</w>'], ['в', '</w>'], ['шк', 'о', 'л', 'е', ',', '</w>'], ['а</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'еник', 'а</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'а</w>', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'с', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'шк', 'ь', 'я']

Итерация 9: Объединение 'ся' в 'ся'. Частота: 1

Обновленный корпус: [['У', 'ч', 'еник', '</w>'], ['учит', 'ся', '</w>'], ['в', '</w>'], ['шк', 'о', 'л', 'е', ',', '</w>'], ['а</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'еник', 'а</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'а</w>', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'с', 'ся', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'шк', 'ь', 'я']

Итерация 10: Объединение 'ол' в 'ол'. Частота: 1

Обновленный корпус: [['У', 'ч', 'еник', '</w>'], ['учит', 'ся', '</w>'], ['в', '</w>'], ['шк', 'ол', 'е', ',', '</w>'], ['а</w>'], ['учит', 'е', 'л', 'ь', '</w>'], ['учит', '</w>'], ['уч', 'еник', 'а</w>']]

Текущий словарь: [',', '</w>', 'У', 'а', 'а</w>', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'ол', 'с', 'ся', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'шк', 'ь', 'я']

--- Обучение завершено ---

Финальный словарь BPE (отсортированный): [',', '</w>', 'У', 'а', 'а</w>', 'в', 'е', 'еник', 'и', 'ит', 'к', 'л', 'н', 'ни', 'ник', 'о', 'ол', 'с', 'ся', 'т', 'у', 'уч', 'учит', 'ч', 'ш', 'шк', 'ь', 'я']

Выполненные объединения: {('у', 'ч'): 'уч', ('и', 'т'): 'ит', ('уч', 'ит'): 'учит', ('н', 'и'): 'ни', ('ни', 'к'): 'ник', ('е', 'ник'): 'еник', ('а', '</w>'): 'а</w>', ('ш', 'к'): 'шк', ('с', 'я'): 'ся', ('о', 'л'): 'ол'}

--- Тестирование токенизации ---

Исходный текст: 'Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика'

Закодированные ID: [2, 23, 7, 1, 22, 18, 1, 5, 1, 25, 16, 6, 0, 1, 4, 22, 6, 11, 26, 1, 22, 1, 21, 7, 4]

Декодированный текст: 'Ученик\_учится\_в\_школе,\_а\_учитель\_учит\_ученика'

--- Тестирование OOV слова ---

Предупреждение: Символ 'ц' не найден в словаре.

OOV текст: 'Учительница\_учит'

Закодированные ID OOV: [2, 23, 9, 6, 11, 26, 13, 4, 22, 1]

Декодированный OOV текст: 'Учительниа\_учит'

--- Токены по ID ---

ID 0: ','

ID 1: '</w>'

ID 2: 'У'

ID 3: 'а'

ID 4: 'а</w>'

ID 5: 'в'

ID 6: 'е'

ID 7: 'еник'

ID 8: 'и'

ID 9: 'ит'

ID 10: 'к'

ID 11: 'л'

ID 12: 'н'

ID 13: 'ни'

ID 14: 'ник'

ID 15: 'о'

ID 16: 'ол'

ID 17: 'с'

ID 18: 'ся'

ID 19: 'т'

ID 20: 'у'

ID 21: 'уч'

ID 22: 'учит'

ID 23: 'ч'

ID 24: 'ш'

ID 25: 'шк'

ID 26: 'ь'

ID 27: 'я'

### **WordPiece Токенизатор**

WordPiece Токенизатор представляет собой элегантный гибридный подход, разработанный для нахождения оптимального баланса между токенизацией на уровне слов и символов. Он разбивает слова на более мелкие, но семантически осмысленные единицы, называемые подсловами (subwords). Например, слово "неизвестные" может быть токенизировано как "не", "##извест", "##ные". Префикс ## указывает, что данный подтокен является продолжением предыдущего токена, а не началом нового слова.  
Основная идея WordPiece: Построить словарь токенов (включающий символы, подслова и, при необходимости, целые слова) таким образом, чтобы он был достаточно компактным, но при этом позволял эффективно представлять как известные, так и неизвестные слова, сохраняя при этом их лингвистический смысл.

Принцип Работы WordPiece: Пошаговое Объяснение

Алгоритм WordPiece, подобно BPE, является итеративным методом, но отличается критерием выбора пар для слияния. Вместо простой частоты встречаемости, WordPiece использует метрику, основанную на вероятности или правдоподобии.

Возьмем наш пример текста для обучения токенизатора: "В училище учитель учит ученикам по новому учебнику."

Шаг 1: Предварительная Обработка Текста и Создание Начального Словаря

Перед началом обучения текст подвергается предварительной обработке:  
Приведение к нижнему регистру: Все буквы переводятся в строчные, чтобы "Учитель" и "учитель" считались одним и тем же словом.

Удаление знаков препинания: Знаки препинания (кроме тех, что являются частью слова, например, в сокращениях, но в нашем примере их нет) удаляются или отделяются.

Разделение на слова: Текст сначала разбивается на слова по пробелам.  
Обработанный текст: "в училище учитель учит ученикам по новому учебнику"

Начальный словарь (V₀): Начинаем с самого простого: каждый уникальный символ в нашем обработанном тексте становится отдельным токеном в нашем словаре. Пробел ( ) также является важным токеном, так как он помогает модели понимать границы слов.

Уникальные символы в тексте: 'в', ' ', 'у', 'ч', 'и', 'л', 'щ', 'е', 'т', 'р', 'ь', 'н', 'к', 'а', 'м', 'п', 'о', 'б', 'ю'.

V₀ = {'в', ' ', 'у', 'ч', 'и', 'л', 'щ', 'е', 'т', 'р', 'ь', 'н', 'к', 'а', 'м', 'п', 'о', 'б', 'ю'} (если бы были другие символы).

Теперь мы можем представить весь наш обработанный текст как последовательность этих начальных токенов. Например, слово "училище" будет представлено как ['у', 'ч', 'и', 'л', 'и', 'щ', 'е'].

Полная последовательность токенов в корпусе (после предварительной обработки): ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'л', 'ь', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Общее количество токенов в корпусе: 50.

Шаг 2: Итеративное Слияние (Обучение Словаря)  
Это ядро алгоритма WordPiece. Мы будем итеративно (повторяющимися шагами) объединять пары соседних токенов (биграммы), чтобы создавать новые, более длинные токены. Цель — найти такие пары, которые, будучи объединенными, максимально увеличивают "вероятность" (или "правдоподобие") нашего обучающего текста.

Формула оценки слияния (Score): Для каждой пары соседних токенов A и B (биграммы AB) мы вычисляем "оценку слияния" по следующей формуле:

где:

* — это количество раз, когда биграмма AB (токен A, за которым сразу следует токен B) встречается в текущем токенизированном корпусе
* — это количество раз, когда токен A встречается в текущем токенизированном корпусе.
* — это количество раз, когда токен B встречается в текущем токенизированном корпусе.

Данная формула измеряет, насколько часто A и B встречаются вместе, по сравнению с тем, насколько часто они встречаются по отдельности. Высокий балл означает, что A и B очень часто появляются вместе, и их слияние в один токен будет "полезным" для модели, так как оно позволяет более компактно (с меньшим количеством токенов) представить текст. Это эквивалентно максимизации правдоподобия обучающих данных, если мы рассматриваем процесс токенизации как генерацию последовательности токенов.

Пример итераций с нашим текстом:

Итерация 1: Поиск первой лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'л', 'ь', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Подсчет частот отдельных токенов (униграмм) в текущем корпусе:

(один раз в начале "в", один — в "новому")

Подсчет частот биграмм (пар соседних токенов):

— в "училище", "учитель", "учит" (дважды), "ученикам", "учебнику"

— аналогично

— в "училище"

— в "училище"

— в "училище"

— в "училище"

— после "училище", "учитель"

— в "учит" (дважды)

— в "учитель"

— в "учитель"

— в "учитель"

— в "ученикам", "учебнику"

— в "ученикам"

— в "ученикам", "учебнику"

— в "ученикам", "учебнику"

— в "ученикам"

— в "ученикам"

— перед "по"

— в "по"

— после "по"

— в "новому", "учебнику"

— в "новому"

— в "новому"

— в "новому"

— в "новому"

— после "в", "училище", "учитель", "новому"

— в "учебнику"

— в "учебнику"

— уже учтено

— уже учтено

— в "учебнику"

— после слова "в"

Вычисление оценок слияния по формуле:

Примеры:

Выбор лучшей биграммы:

Наивысшую оценку имеют биграммы: 'л', 'ь', 'к', 'а', 'а', 'м', 'б', 'н'.

При равенстве оценок алгоритм может выбрать первую по порядку в лексикографическом или в порядке появления.

Предположим, выбирается первая появляющаяся такая пара — 'л', 'ь' (в слове "учитель").

Слияние 'л', 'ь':

Все вхождения последовательности ['л', 'ь'] заменяются на новый токен 'ль'.  
Например, слово "учитель" из ['у', 'ч', 'и', 'т', 'е', 'л', 'ь'] становится ['у', 'ч', 'и', 'т', 'е', 'ль'].

Обновление словаря:

Добавляем новый токен 'ль' в словарь:

Итерация 2: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']  
(Токен 'ль' уже заменил последовательность 'л', 'ь' в слове "учитель".)

Общее количество токенов: 49.

Подсчет частот отдельных токенов (обновленные):

— всё ещё встречается в "училище" ('и', 'л', 'и')

— больше не встречается как отдельный токен

— новый токен

Остальные частоты остаются без изменений:

, , , , , , , , и т.д.

Подсчет частот биграмм (обновленные):

Биграмма 'л', 'ь' больше не встречается.

Появилась новая биграмма: 'е', 'ль' (из "учитель") →

Остальные биграммы, не затронутые заменой, сохраняют свои частоты:

, , , , и т.д.

Вычисление оценок слияния: Используем формулу:

Примеры:

Выбор лучшей биграммы:

Наивысшую оценку имеют биграммы: 'к', 'а', 'а', 'м', 'б', 'н'.  
Выбираем первую по порядку появления — 'к', 'а' (в слове "ученикам").

Слияние 'к', 'а':

Все вхождения последовательности ['к', 'а'] заменяются на новый токен 'ка'.

Например, слово "ученикам" из ['у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м'] становится ['у', 'ч', 'е', 'н', 'и', 'ка', 'м'].

Обновление словаря:

Добавляем новый токен 'ка' в словарь:

Итерация 3: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'ка', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']  
(Токены 'ль' и 'ка' уже вставлены.)

Общее количество токенов: 48.

Подсчет частот отдельных токенов (обновленные):

— встречается в "учебнику"

— больше не встречается как отдельный токен

— пока нет, но биграмма 'а', 'м' всё ещё может существовать

— в последовательности ['ка', 'м'] → 'ка', 'м', а не 'а', 'м'  
→ Значит, биграмма 'а', 'м' больше не существует, так как 'а' исчез как отдельный токен

Однако в слове "ученикам" теперь: ['ка', 'м'], значит:

Биграмма: 'ка', 'м', а не 'а', 'м'

Следовательно:

→ биграмма исчезла

→ новая биграмма

Подсчет частот биграмм:

Биграммы 'к', 'а' и 'а', 'м' больше не встречаются.

Появилась новая биграмма: 'ка', 'м' →

Остальные биграммы без изменений, например:

Вычисление оценок:

Выбор лучшей биграммы:

Наивысшую оценку имеют 'ка', 'м' и 'б', 'н'.

Выбираем 'ка', 'м' как следующую по порядку.

Слияние 'ка', 'м':

Заменяем последовательность ['ка', 'м'] на новый токен 'кам'.

Слово "ученикам" из ['у', 'ч', 'е', 'н', 'и', 'ка', 'м'] становится ['у', 'ч', 'е', 'н', 'и', 'кам'].

Обновление словаря:

Добавляем новый токен 'кам' в словарь:

Итерация 4: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']  
(Токены 'ль', 'ка', 'ам' уже объединены в 'кам'. Уточнение: 'ам' как отдельный токен не был добавлен — слияние произошло напрямую в 'кам'.)  
Общее количество токенов: 47.

Подсчет частот отдельных токенов (обновленные):

— больше не встречается

— встречается в "новому" ('м', 'у')

Остальные частоты без изменений.

Подсчет частот биграмм (обновленные):

Биграмма 'и', 'кам' появляется один раз (в "ученикам").

Остальные биграммы, не затронутые слиянием, сохраняют свои значения:

— в "учебнику"

Вычисление оценок слияния:

Выбор лучшей биграммы:

Наивысшую оценку имеет биграмма 'б', 'н'.

Слияние 'б', 'н':

Заменяем последовательность ['б', 'н'] на новый токен 'бн'.

Слово "учебнику" из ['у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у'] становится ['у', 'ч', 'е', 'бн', 'и', 'к', 'у'].

Обновление словаря:

Добавляем новый токен 'бн' в словарь:

Итерация 5: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч',

'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']  
(Токены 'ль', 'кам', 'бн' добавлены.)  
Общее количество токенов: 46.

Подсчет частот отдельных токенов (обновленные):

— встречается в "ученикам" ('е', 'н') и "новому" ('н', 'о')

Подсчет частот биграмм:

Появились новые биграммы:

— в "учебнику"

— в "учебнику"

— в "новому" и "учебнику"

, ,

Вычисление оценок:

Выбор лучшей биграммы:

Наивысшую оценку имеет 'н', 'о'.

Слияние 'н', 'о':

Заменяем все вхождения ['н', 'о'] на новый токен 'но'.  
Слово "новому" из ['н', 'о', 'в', 'о', 'м', 'у'] становится ['но', 'в', 'о', 'м', 'у'].

Обновление словаря:

Итерация 6: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'но', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']  
Общее количество токенов: 45.

Подсчет частот:

— в "в" и "новому"

— в "по", "новому"

Частоты биграмм:

— в "новому"

Оценка слияния:

Выбор лучшей биграммы:

Среди текущих биграмм 'но', 'в' имеет одну из высоких оценок.

Слияние 'но', 'в':

Заменяем ['но', 'в'] на новый токен 'нов'.

Слово "новому" становится ['нов', 'о', 'м', 'у'].

Обновление словаря:

Итерация 7: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'нов', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']  
Общее количество токенов: 44.

Частоты:

Биграмма:

Оценка слияния:

Слияние 'нов', 'о':

Заменяем ['нов', 'о'] на 'ново'.

Слово "новому" становится ['ново', 'м', 'у'].

Обновление словаря:

Итерация 8: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'ново', 'м', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']  
(Токены 'ль', 'кам', 'бн', 'но', 'нов', 'ново' уже добавлены.)  
Общее количество токенов: 43.

Подсчет частот отдельных токенов (обновленные):

— встречается в "новому"

— больше не встречается как отдельный токен

Остальные частоты без изменений.

Подсчет частот биграмм:

— в слове "новому"

Оценка слияния:

Выбор лучшей биграммы:

Биграмма 'ново', 'м' имеет высокую оценку (1.0).

Слияние 'ново', 'м':

Заменяем последовательность ['ново', 'м'] на новый токен 'новом'.

Слово "новому" из ['ново', 'м', 'у'] становится ['новом', 'у'].

Обновление словаря:

Добавляем новый токен 'новом' в словарь:

Итерация 9: Поиск следующей лучшей биграммы

Текущий корпус:

['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']  
(Токены 'новом' и другие уже включены.)  
Общее количество токенов: 42.

Подсчет частот отдельных токенов:

— встречается в "училище", "учитель", "учит", "ученикам", "учебнику"

Частота биграммы:

Оценка слияния:

Выбор лучшей биграммы:

На данном шаге среди всех доступных биграмм 'новом', 'у' может быть одной из лучших, особенно если других высокочастотных пар не осталось.

Слияние 'новом', 'у':

Заменяем ['новом', 'у'] на новый токен 'новому'.

Слово "новому" теперь представлено как ['новому'].

Обновление словаря:

Добавляем новый токен 'новому' в словарь:

Мы продемонстрировали, как слово *"новому"* постепенно собиралось из отдельных символов через последовательные слияния:  
'н'+'о'→'но', 'но'+'в'→'нов', 'нов'+'о'→'ново', 'ново'+'м'→'новом', 'новом'+'у'→'новому'.

Таким образом, WordPiece может формировать целые слова как отдельные токены, если они часто встречаются и слияние оправдано с точки зрения правдоподобия.

Этот процесс продолжается итеративно до достижения желаемого размера словаря. В реальных моделях, таких как BERT, он проходит тысячи или десятки тысяч итераций на огромных текстовых корпусах.

В конечном итоге, словарь содержит смесь:

* отдельных символов,
* подслов (например, 'уч', 'кам'),
* целых слов (например, 'новому'),

что позволяет эффективно кодировать как частотные, так и редкие или неизвестные слова.

2.3. Когда остановить итерацию? (Критерии остановки)

Процесс итеративного слияния в WordPiece обычно останавливается при выполнении одного из следующих условий:

Достигнут заданный размер словаря:

Это — наиболее распространённый критерий. Разработчик заранее определяет максимальное количество токенов в словаре (например, 30 000 для BERT, 50 000 для RoBERTa). Обучение завершается, как только словарь достигает этого предела.

Невозможно выполнить дальнейшие слияния:  
Это происходит, когда в корпусе больше нет соседних токенов, которые можно объединить — например, весь текст представлен минимальным числом токенов, или все возможные биграммы уже слиты.

Оценка лучшей биграммы падает ниже порога:  
Можно установить минимальное значение (Score(A, B)), ниже которого слияния считаются незначимыми. Если даже лучшая доступная биграмма имеет слишком низкую оценку, процесс останавливается, так как дальнейшие слияния вносят незначительный вклад в правдоподобие текста.

Пример конечного словаря (Vfinal) после множества итераций:

На основе последовательных итераций, наш словарь постепенно будет включать в себя следующие токены:

{'в', ' ', 'у', 'ч', 'и', 'л', 'щ', 'е', 'т', 'р', 'ь', 'н', 'к', 'а', 'м', 'п', 'о', 'б', 'ль', 'ка', 'ам', 'бн', 'но', 'нов', 'ново', 'новом', 'новому', 'уч', 'чи', 'ит', 'ищ', 'ще', 'е ', 'ни', 'ку', 'учи', 'учил', 'учили', 'училище', 'учитель', 'учит', 'уче', 'учени', 'ученик', 'ученикам', 'учеб', 'учебни', 'учебник', 'учебнику'}.

Начальные символы: Все уникальные символы из текста (например, 'в', ' ' , 'у', 'ч' и т.д.) остаются в словаре — они необходимы для токенизации редких или неизвестных слов.

Итеративные слияния: В процессе обучения добавляются составные токены: 'ль', 'ка', 'бн', 'но', 'нов', 'ново', 'новом', 'новому' — как результат последовательных оптимальных слияний.

Более длинные подслова и целые слова: При продолжении процесса часто встречающиеся фрагменты и целые слова, такие как 'учи', 'учил', 'учили', 'училище', 'учитель', 'учит', 'ученикам', 'учебник', 'учебнику', 'по', также становятся отдельными токенами.

Важное примечание о префиксах:

* В реальных реализациях WordPiece (например, в BERT) для обозначения продолжения слова используется префикс ##. Например, слово *"играющий"* может быть разбито на ["игра", "##ющий"]. Это позволяет модели различать, где начинается новое слово, а где — продолжается предыдущее.
* В нашем упрощённом примере мы опустили ## для наглядности, но в реальных системах эта система критически важна для корректной обработки морфологии и редких слов.

3. Токенизация Нового Текста с Использованием Обученного Словаря

После завершения обучения и формирования финального словаря, токенизатор может обрабатывать любой новый текст. Процесс токенизации одного слова выглядит следующим образом:\

Ищем самый длинный подтокен из словаря, который совпадает с началом (префиксом) слова.

Если такой подтокен найден, добавляем его в выходную последовательность и повторяем процесс для оставшейся части слова.

Если часть слова не может быть разбита на известные подтокены, она либо:

* разбивается на отдельные символы (если они есть в словаре),
* либо заменяется специальным токеном [UNK] (Unknown), если ни один фрагмент не найден.

Этот алгоритм называется жадным разбиением по наибольшему префиксу (greedy longest-match-first).

Пример токенизации с гипотетическим Vfinal:Текст: "в училище учитель учит ученикам по новому учебнику"

Слово "в":→ "в" есть в словаре → ['в']

Слово "училище":→ "училище" есть в словаре → ['училище']

Слово "учитель":→ "учитель" есть в словаре → ['учитель']

Слово "учит":→ "учит" есть в словаре → ['учит']

Слово "ученикам":→ "ученикам" есть в словаре → ['ученикам']

Слово "по":→ "по" есть в словаре → ['по']

Слово "новому":→ "новому" есть в словаре → ['новому']

Слово "учебнику":→ "учебнику" есть в словаре → ['учебнику']

Финальная токенизация (с пробелами как отдельными токенами):  
['в', ' ', 'училище', ' ', 'учитель', ' ', 'учит', ' ', 'ученикам', ' ', 'по', ' ', 'новому', ' ', 'учебнику']

⚠️ Примечание: В большинстве современных реализаций пробелы не используются как отдельные токены. Вместо этого они либо удаляются, либо заменяются на ## при внутрисловном слиянии. Однако в нашем упрощённом примере пробел сохраняется для наглядности.

Случай неизвестного слова:

Если бы в тексте появилось слово *"преподавательница"*, которого нет в словаре, но есть подтокены:

"препода" → да

"##ватель" → да

"##ница" → да

То оно было бы токенизировано как:

['препода', '##ватель', '##ница']

Если же ни один подтокен не найден (например, из-за опечатки), слово может быть заменено на [UNK].

4. Масштабирование WordPiece на Многопредметный Корпус

При обучении на большом корпусе, состоящем из множества предложений или документов, алгоритм WordPiece адаптируется, чтобы сохранить лингвистическую корректность и избежать артефактов.

Ключевые аспекты масштабирования:

Обработка по границам предложений:

Каждое предложение обрабатывается независимо. Подсчёт частот биграмм и слияние токенов не пересекают границы предложений. Это предотвращает бессмысленные слияния между последним словом одного предложения и первым — другого.

Специальные разделительные токены:

Часто используются специальные маркеры:

* <s> — начало предложения
* </s> — конец предложения

Эти токены помогают модели понимать структуру текста и не допускают слияния через границы.

\Запрет межпредложных слияний:

Это принципиально важно. Например, если одно предложение заканчивается словом, оканчивающимся на "ок", а следующее начинается с "но", WordPiece не будет объединять "ок" и "но" в "окно", потому что они находятся в разных предложениях

Это гарантирует, что все формируемые токены имеют лингвистический смысл в контексте одного предложения.

Работа с большим корпусом:

На практике WordPiece обучается на гигантских корпусах (например, весь Wikipedia). Процесс включает:

* предварительную токенизацию на уровне слов,
* построение начального символьного словаря,
* десятки тысяч итераций слияния,
* остановку по достижению целевого размера словаря (например, 30 000 токенов).

Пример: Обучение WordPiece на корпусе предложений

Рассмотрим следующий корпус из трёх предложений:

"Ученик учится в школе, а учитель учит ученика."

"Новый студент изучает грамматику."

"Учитель и студент читают книги."

Начальная настройка

Перед обучением добавим специальные токены для обозначения границ предложений:

* <s> — начало предложения
* </s> — конец предложения

Также проведём предварительную обработку:

Приведение к нижнему регистру

Замена пробелов на символ \_ (для наглядности — в реальных реализациях используется пробел или ##)

Разделение на символы

Токенизированный корпус (на начальном символьном уровне):

<s> у \_ ч \_ е \_ н \_ и \_ к \_ \_ у \_ ч \_ и \_ т \_ с \_ я \_ \_ в \_ \_ ш \_ к \_ о \_ л \_ е \_ , \_ \_ а \_ \_ у \_ ч \_ и \_ т \_ е \_ л \_ ь \_ \_ у \_ ч \_ и \_ т \_ \_ у \_ ч \_ е \_ н \_ и \_ к \_ а \_ . </s>  
<s> \_ н \_ о \_ в \_ ы \_ й \_ \_ с \_ т \_ у \_ д \_ е \_ н \_ т \_ \_ и \_ з \_ у \_ ч \_ а \_ е \_ т \_ \_ г \_ р \_ а \_ м \_ м \_ а \_ т \_ и \_ к \_ у \_ . </s>  
<s> \_ у \_ ч \_ и \_ т \_ е \_ л \_ ь \_ \_ и \_ \_ с \_ т \_ у \_ д \_ е \_ н \_ т \_ \_ ч \_ и \_ т \_ а \_ ю \_ т \_ \_ к \_ н \_ и \_ г \_ и \_ . </s>

⚠️ Примечание: В реальных реализациях (например, BERT) пробелы не заменяются на \_, а обрабатываются как отдельные символы. Здесь \_ используется для наглядности.

Итерации WordPiece на корпусе

При обучении частоты биграмм считаются внутри каждого предложения отдельно, чтобы избежать слияний через границы. Затем частоты суммируются по всему корпусу.

Пример подсчёта биграмм (гипотетический):

Из предложения 1:

('у', 'ч'): 5  
('и', 'к'): 2  
('\_', 'у'): 3  
('а', '\_'): 2  
('т', 'е'): 1

Из предложения 2:

('н', 'о'): 1  
('с', 'т'): 1  
('т', 'у'): 1  
('г', 'р'): 1

Из предложения 3:

('у', 'ч'): 1  
('т', 'е'): 2

('ч', 'и'): 1

('и', '\_'): 1

('к', 'н'): 1

Суммарные частоты по корпусу:

('у', 'ч'): 5 + 1 = 6

('\_', 'у'): 3 + 1 = 4

('т', 'е'): 1 + 2 = 3

('и', 'к'): 2

('с', 'т'): 1 + 1 = 2

Решение об объединении

На первой итерации пара ('у', 'ч') имеет наивысшую частоту (6). Согласно формуле WordPiece:

она также, скорее всего, будет иметь высокую оценку, особенно если у и ч встречаются вместе значительно чаще, чем можно было бы ожидать случайно.

Мы объединяем у и ч в новый токен уч.

Обновление корпуса:

Во всех предложениях заменяем последовательности ['у', 'ч'] на ['уч'].

Например:

Было: ... \_ у \_ ч \_ и \_ т ...

Стало: ... \_ уч \_ и \_ т ...

Процесс повторяется: пересчитываются частоты, выбирается следующая лучшая биграмма, выполняется слияние — и так до достижения целевого размера словаря.

Почему этот подход важен

Осмысленные токены: Поскольку слияния не пересекают границы предложений, исключаются артефакты вроде объединения конца одного предложения с началом другого (например, "а\_н" из "а."</s><s>н"). Это гарантирует, что формируемые токены лингвистически осмысленны.

Целостность предложений: Сохранение границ помогает модели учитывать синтаксическую и семантическую структуру предложений — критично для задач, таких как машинный перевод, классификация или генерация.

Масштабируемость: Подход позволяет обрабатывать большие или потоковые корпусы, где не все данные доступны одновременно. Это делает WordPiece пригодным для промышленного использования.

Последовательная токенизация: Одинаковые слова в одинаковом контексте будут токенизированы одинаково, что обеспечивает стабильность и воспроизводимость.

5. Преимущества и недостатки WordPiece

Хотя WordPiece широко используется (в BERT, DistilBERT и других моделях), у него есть как сильные стороны, так и ограничения.

5.1. Преимущества WordPiece

Эффективная обработка OOV-слов (Out-of-Vocabulary):  
WordPiece способен разбивать неизвестные слова на знакомые подслова. Например, *"программистом"* → ["программ", "##ист", "##ом"], даже если целое слово отсутствует в словаре. Это обеспечивает частичное понимание и устойчивость к лексической вариативности.

Контроль размера словаря: Размер словаря задаётся заранее (например, 30 000). Алгоритм итеративно выбирает наиболее "полезные" слияния, максимизируя правдоподобие текста. Это позволяет балансировать между компактностью и покрытием.

Гибридный подход: WordPiece сочетает преимущества символьной и словесной токенизации:

* Частые слова (например, "учитель") становятся одним токеном.
* Редкие или сложные слова разбиваются на подслова.
* Все входные данные могут быть представлены — нет "неизвестных" символов.

Универсальность (в варианте Byte-level WordPiece): Байтовый WordPiece (используется в моделях вроде RoBERTa) работает на уровне байтов, а не символов Unicode. Это делает его инвариантным к кодировке, способным обрабатывать любой текст (включая эмодзи, редкие символы), и гарантирует отсутствие [UNK].

5.2. Недостатки WordPiece

Жадный и детерминированный алгоритм:

На каждом шаге выбирается пара с максимальной оценкой. Такой подход не учитывает долгосрочных последствий и может привести к субоптимальной сегментации с точки зрения морфологии или семантики.

Отсутствие лингвистических знаний:

WordPiece — статистический алгоритм. Он не "понимает" морфемы. Например, может разбить "бежать" на ["беж", "##ать"], хотя лингвистически более корректно — ["бег", "##ать"]. Это снижает интерпретируемость и иногда — эффективность.

Вычислительная сложность:

На ранних этапах обучения (при большом количестве символьных токенов) подсчёт частот всех биграмм требует значительных ресурсов. Для больших корпусов это может быть медленным.

Зависимость от начальной токенизации:  
Если перед WordPiece используется разбиение по словам, то ошибки на этом этапе (например, неудачная обработка знаков препинания) могут повлиять на итоговый словарь и сегментацию.

Неоднозначность сегментации:

Одно и то же слово может быть разбито по-разному в зависимости от истории слияний. Например, "ученик" может стать ["уч", "##еник"] или ["учени", "##к"] — в зависимости от того, какие пары были выбраны ранее. Хотя на практике это редко вызывает проблемы, теоретически снижает согласованность.

Заключение

WordPiece стал краеугольным камнем современных NLP-моделей. Его способность эффективно справляться с неизвестными словами, контролировать размер словаря и сохранять баланс между символами и словами сделала его выбором по умолчанию для таких моделей, как BERT и DistilBERT.

Однако его ограничения — жадность, отсутствие морфологического понимания — стимулировали разработку альтернатив:

SentencePiece — обучается без предварительной токенизации по словам.

Unigram Language Model — использует вероятностную модель для сегментации.

BoundlessBPE / BBPE — улучшает морфологическую целостность.

Эта эволюция показывает: нет универсального идеального токенизатора. Выбор зависит от задачи, языка, требований к размеру модели и компромисса между эффективностью, точностью и универсальностью.

Реализация на Питон:

|  |
| --- |
| import collections  class WordPieceTokenizer:      def \_\_init\_\_(self):          self.vocab = set()          self.merges = []        def preprocess\_text(self, text):          """          Предварительная обработка текста: приведение к нижнему регистру и разбиение на слова.          """          # Удаляем знаки препинания и приводим к нижнему регистру          processed\_text = text.lower().replace('...', '').replace('.', '')          # Разбиваем на слова по пробелам          words = processed\_text.split(' ')          # Добавляем пробелы как отдельные токены между словами          initial\_tokens = []          for i, word in enumerate(words):              if word: # Убедимся, что слово не пустое                  initial\_tokens.extend(list(word))              if i < len(words) - 1:                  initial\_tokens.append(' ') # Добавляем пробел между словами          return initial\_tokens, words # Возвращаем начальные токены и список слов для удобства      def calculate\_frequencies(self, tokens):          """          Подсчет частот отдельных токенов (униграмм) и биграмм в текущем корпусе.          """          unigram\_freq = collections.defaultdict(int)          bigram\_freq = collections.defaultdict(int)          for i in range(len(tokens)):              unigram\_freq[tokens[i]] += 1              if i < len(tokens) - 1:                  bigram\_freq[(tokens[i], tokens[i+1])] += 1          return unigram\_freq, bigram\_freq      def calculate\_score(self, unigram\_freq, bigram\_freq):          """          Вычисление оценки слияния для всех возможных биграмм.          Score(A, B) = frequency(AB) / (frequency(A) \* frequency(B))          """          scores = {}          for (token\_a, token\_b), freq\_ab in bigram\_freq.items():              freq\_a = unigram\_freq[token\_a]              freq\_b = unigram\_freq[token\_b]              if freq\_a > 0 and freq\_b > 0: # Избегаем деления на ноль                  score = freq\_ab / (freq\_a \* freq\_b)                  scores[(token\_a, token\_b)] = score          return scores      def merge\_tokens(self, tokens, best\_bigram):          """          Слияние лучшей биграммы в корпусе.          """          merged\_token = "".join(best\_bigram)          new\_tokens = []          i = 0          while i < len(tokens):              if i + 1 < len(tokens) and (tokens[i], tokens[i+1]) == best\_bigram:                  new\_tokens.append(merged\_token)                  i += 2 # Пропускаем оба слитых токена              else:                  new\_tokens.append(tokens[i])                  i += 1          return new\_tokens      def train(self, text, target\_vocab\_size=None, num\_merges=None):          """          Обучение WordPiece токенизатора.          Итеративно сливает токены до достижения целевого размера словаря          или заданного количества слияний.          """          print(f"Исходный текст: '{text}'")          current\_tokens, \_ = self.preprocess\_text(text)            # Начальный словарь состоит из всех уникальных символов          self.vocab = set(current\_tokens)          print(f"\nШаг 1: Начальный словарь (V0) содержит {len(self.vocab)} токенов.")          print(f"Начальный корпус: {current\_tokens}")          merges\_count = 0          while True:              unigram\_freq, bigram\_freq = self.calculate\_frequencies(current\_tokens)              scores = self.calculate\_score(unigram\_freq, bigram\_freq)              if not scores:                  print("\nНет больше биграмм для слияния. Остановка.")                  break              # Находим биграмму с наивысшей оценкой              best\_bigram = max(scores, key=scores.get)              best\_score = scores[best\_bigram]              merged\_token = "".join(best\_bigram)              # Критерий остановки: если новый токен уже в словаре              if merged\_token in self.vocab:                  # Если лучший токен уже существует, удалим его из scores, чтобы найти следующий лучший                  del scores[best\_bigram]                  continue # Продолжаем поиск              # Выводим информацию о текущем слиянии              print(f"\nИтерация {merges\_count + 1}:")              print(f"  Лучшая биграмма для слияния: '{best\_bigram[0]}' + '{best\_bigram[1]}' -> '{merged\_token}' (Score: {best\_score:.4f})")                # Выполняем слияние в корпусе              current\_tokens = self.merge\_tokens(current\_tokens, best\_bigram)                # Добавляем новый токен в словарь и сохраняем слияние              self.vocab.add(merged\_token)              self.merges.append(best\_bigram)              merges\_count += 1                print(f"  Корпус после слияния: {current\_tokens}")              print(f"  Текущий размер словаря: {len(self.vocab)}")              # Критерии остановки              if target\_vocab\_size is not None and len(self.vocab) >= target\_vocab\_size:                  print(f"\nДостигнут целевой размер словаря ({target\_vocab\_size}). Остановка.")                  break              if num\_merges is not None and merges\_count >= num\_merges:                  print(f"\nДостигнуто заданное количество слияний ({num\_merges}). Остановка.")                  break            print(f"\nОбучение завершено. Итоговый словарь содержит {len(self.vocab)} токенов.")          return sorted(list(self.vocab))      def tokenize(self, text):          """          Токенизация нового текста с использованием обученного словаря WordPiece.          """          if not hasattr(self, 'vocab') or len(self.vocab) == 0:              raise ValueError("Токенизатор не обучен. Сначала вызовите метод train().")            print(f"\n--- Токенизация нового текста ---")          print(f"Текст для токенизации: '{text}'")            # Предварительная обработка текста для токенизации          processed\_text = text.lower().replace('...', '').replace('.', '')          words = processed\_text.split(' ')          final\_tokens = []          for word in words:              if not word: # Пропускаем пустые строки, если они возникли из-за множественных пробелов                  continue              word\_tokens = []              remaining\_word = word                while remaining\_word:                  found\_match = False                  # Ищем самый длинный подтокен из словаря, который является префиксом оставшейся части слова                  for i in range(len(remaining\_word), 0, -1):                      subword = remaining\_word[:i]                        if subword in self.vocab:                          word\_tokens.append(subword)                          remaining\_word = remaining\_word[i:]                          found\_match = True                          break                    if not found\_match:                      # Если не удалось найти соответствующий токен, разбиваем на символы                      # или используем токен [UNK]                      if remaining\_word[0] in self.vocab: # Если символ есть в словаре                          word\_tokens.append(remaining\_word[0])                      else: # Если символ даже не в начальном словаре                          word\_tokens.append('[UNK]')                      remaining\_word = remaining\_word[1:]              final\_tokens.extend(word\_tokens)              final\_tokens.append(' ') # Добавляем пробел между токенизированными словами          # Удаляем последний пробел, если он есть          if final\_tokens and final\_tokens[-1] == ' ':              final\_tokens.pop()          print(f"Токенизированный текст: {final\_tokens}")          return final\_tokens      def get\_vocab(self):          """Возвращает текущий словарь токенов."""          return sorted(list(self.vocab))        def get\_merges(self):          """Возвращает список выполненных слияний."""          return self.merges  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":      text\_to\_train = "В училище учитель учит ученикам по новому учебнику"        # Создаем экземпляр токенизатора      tokenizer = WordPieceTokenizer()        # Обучаем токенизатор      final\_vocab = tokenizer.train(text\_to\_train, num\_merges=50)        print(f"\nИтоговый словарь WordPiece: {final\_vocab}")        # Токенизируем тот же текст      tokenizer.tokenize(text\_to\_train)        # Пример токенизации нового слова      new\_text = "учительница"      # Добавим '##ница' в словарь для демонстрации      if 'ница' not in final\_vocab:          tokenizer.vocab.add('ница')          tokenizer.vocab.add('##ница')          final\_vocab = tokenizer.get\_vocab()        print(f"\nИтоговый словарь WordPiece (с 'ница' для демонстрации): {final\_vocab}")      tokenizer.tokenize(new\_text)        # Демонстрация с упрощенным словарем      simplified\_tokenizer = WordPieceTokenizer()      simplified\_tokenizer.vocab = set(['у', 'ч', 'и', 'т', 'е', 'л', 'ь', 'н', 'ц', 'а', 'учитель', '##ница', '##тель', '##а'])      print(f"\n--- Демонстрация токенизации подслов с упрощенным словарем ---")      simplified\_tokenizer.tokenize("учительница") |

Результата

Исходный текст: 'В училище учитель учит ученикам по новому учебнику'

Шаг 1: Начальный словарь (V0) содержит 17 токенов.

Начальный корпус: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'л', 'ь', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Итерация 1:

Лучшая биграмма для слияния: 'л' + 'ь' -> 'ль' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'к', 'а', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 18

Итерация 2:

Лучшая биграмма для слияния: 'к' + 'а' -> 'ка' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'ка', 'м', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 19

Итерация 3:

Лучшая биграмма для слияния: 'ка' + 'м' -> 'кам' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'п', 'о', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 20

Итерация 4:

Лучшая биграмма для слияния: 'п' + 'о' -> 'по' (Score: 0.3333)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'н', 'о', 'в', 'о', 'м', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 21

Итерация 5:

Лучшая биграмма для слияния: 'о' + 'м' -> 'ом' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'н', 'о', 'в', 'ом', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 22

Итерация 6:

Лучшая биграмма для слияния: 'о' + 'в' -> 'ов' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'н', 'ов', 'ом', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 23

Итерация 7:

Лучшая биграмма для слияния: 'ов' + 'ом' -> 'овом' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'н', 'овом', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 24

Итерация 8:

Лучшая биграмма для слияния: 'н' + 'овом' -> 'новом' (Score: 0.3333)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'б', 'н', 'и', 'к', 'у']

Текущий размер словаря: 25

Итерация 9:

Лучшая биграмма для слияния: 'б' + 'н' -> 'бн' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'щ', 'е', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']

Текущий размер словаря: 26

Итерация 10:

Лучшая биграмма для слияния: 'щ' + 'е' -> 'ще' (Score: 0.2500)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'т', 'е', 'ль', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']

Текущий размер словаря: 27

Итерация 11:

Лучшая биграмма для слияния: 'е' + 'ль' -> 'ель' (Score: 0.3333)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'т', 'ель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']

Текущий размер словаря: 28

Итерация 12:

Лучшая биграмма для слияния: 'т' + 'ель' -> 'тель' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'тель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'е', 'н', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']

Текущий размер словаря: 29

Итерация 13:

Лучшая биграмма для слияния: 'е' + 'н' -> 'ен' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'тель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'ен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'е', 'бн', 'и', 'к', 'у']

Текущий размер словаря: 30

Итерация 14:

Лучшая биграмма для слияния: 'е' + 'бн' -> 'ебн' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'тель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'ч', 'ен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'ебн', 'и', 'к', 'у']

Текущий размер словаря: 31

Итерация 15:

Лучшая биграмма для слияния: 'ч' + 'ен' -> 'чен' (Score: 0.2000)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'тель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'ч', 'ебн', 'и', 'к', 'у']

Текущий размер словаря: 32

Итерация 16:

Лучшая биграмма для слияния: 'ч' + 'ебн' -> 'чебн' (Score: 0.2500)

Корпус после слияния: ['в', ' ', 'у', 'ч', 'и', 'л', 'и', 'ще', ' ', 'у', 'ч', 'и', 'тель', ' ', 'у', 'ч', 'и', 'т', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 33

Итерация 17:

Лучшая биграмма для слияния: 'ч' + 'и' -> 'чи' (Score: 0.1667)

Корпус после слияния: ['в', ' ', 'у', 'чи', 'л', 'и', 'ще', ' ', 'у', 'чи', 'тель', ' ', 'у', 'чи', 'т', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 34

Итерация 18:

Лучшая биграмма для слияния: 'чи' + 'л' -> 'чил' (Score: 0.3333)

Корпус после слияния: ['в', ' ', 'у', 'чил', 'и', 'ще', ' ', 'у', 'чи', 'тель', ' ', 'у', 'чи', 'т', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 35

Итерация 19:

Лучшая биграмма для слияния: 'чи' + 'тель' -> 'читель' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'чил', 'и', 'ще', ' ', 'у', 'читель', ' ', 'у', 'чи', 'т', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 36

Итерация 20:

Лучшая биграмма для слияния: 'чи' + 'т' -> 'чит' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'чил', 'и', 'ще', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 37

Итерация 21:

Лучшая биграмма для слияния: 'чил' + 'и' -> 'чили' (Score: 0.3333)

Корпус после слияния: ['в', ' ', 'у', 'чили', 'ще', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 38

Итерация 22:

Лучшая биграмма для слияния: 'чили' + 'ще' -> 'чилище' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'чен', 'и', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 39

Итерация 23:

Лучшая биграмма для слияния: 'чен' + 'и' -> 'чени' (Score: 0.5000)

Корпус после слияния: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'чени', 'кам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 40

Итерация 24:

Лучшая биграмма для слияния: 'чени' + 'кам' -> 'ченикам' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебн', 'и', 'к', 'у']

Текущий размер словаря: 41

Итерация 25:

Лучшая биграмма для слияния: 'чебн' + 'и' -> 'чебни' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебни', 'к', 'у']

Текущий размер словаря: 42

Итерация 26:

Лучшая биграмма для слияния: 'чебни' + 'к' -> 'чебник' (Score: 1.0000)

Корпус после слияния: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 43

Итерация 27:

Лучшая биграмма для слияния: 'в' + ' ' -> 'в ' (Score: 0.1429)

Корпус после слияния: ['в ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 44

Итерация 28:

Лучшая биграмма для слияния: 'чилище' + ' ' -> 'чилище ' (Score: 0.1667)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 45

Итерация 29:

Лучшая биграмма для слияния: 'читель' + ' ' -> 'читель ' (Score: 0.2000)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 46

Итерация 30:

Лучшая биграмма для слияния: 'чит' + ' ' -> 'чит ' (Score: 0.2500)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 47

Итерация 31:

Лучшая биграмма для слияния: 'ченикам' + ' ' -> 'ченикам ' (Score: 0.3333)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 48

Итерация 32:

Лучшая биграмма для слияния: 'ченикам ' + 'по' -> 'ченикам по' (Score: 1.0000)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 49

Итерация 33:

Лучшая биграмма для слияния: 'ченикам по' + ' ' -> 'ченикам по ' (Score: 0.5000)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам по ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 50

Итерация 34:

Лучшая биграмма для слияния: 'ченикам по ' + 'новом' -> 'ченикам по новом' (Score: 1.0000)

Корпус после слияния: ['в ', 'у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 51

Итерация 35:

Лучшая биграмма для слияния: 'в ' + 'у' -> 'в у' (Score: 0.1429)

Корпус после слияния: ['в у', 'чилище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 52

Итерация 36:

Лучшая биграмма для слияния: 'в у' + 'чилище ' -> 'в училище ' (Score: 1.0000)

Корпус после слияния: ['в училище ', 'у', 'читель ', 'у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 53

Итерация 37:

Лучшая биграмма для слияния: 'в училище ' + 'у' -> 'в училище у' (Score: 0.1667)

Корпус после слияния: ['в училище у', 'читель ', 'у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 54

Итерация 38:

Лучшая биграмма для слияния: 'в училище у' + 'читель ' -> 'в училище учитель ' (Score: 1.0000)

Корпус после слияния: ['в училище учитель ', 'у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 55

Итерация 39:

Лучшая биграмма для слияния: 'в училище учитель ' + 'у' -> 'в училище учитель у' (Score: 0.2000)

Корпус после слияния: ['в училище учитель у', 'чит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 56

Итерация 40:

Лучшая биграмма для слияния: 'в училище учитель у' + 'чит ' -> 'в училище учитель учит ' (Score: 1.0000)

Корпус после слияния: ['в училище учитель учит ', 'у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 57

Итерация 41:

Лучшая биграмма для слияния: 'в училище учитель учит ' + 'у' -> 'в училище учитель учит у' (Score: 0.2500)

Корпус после слияния: ['в училище учитель учит у', 'ченикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 58

Итерация 42:

Лучшая биграмма для слияния: 'в училище учитель учит у' + 'ченикам по новом' -> 'в училище учитель учит ученикам по новом' (Score: 1.0000)

Корпус после слияния: ['в училище учитель учит ученикам по новом', 'у', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 59

Итерация 43:

Лучшая биграмма для слияния: 'в училище учитель учит ученикам по новом' + 'у' -> 'в училище учитель учит ученикам по новому' (Score: 0.3333)

Корпус после слияния: ['в училище учитель учит ученикам по новому', ' ', 'у', 'чебник', 'у']

Текущий размер словаря: 60

Итерация 44:

Лучшая биграмма для слияния: 'в училище учитель учит ученикам по новому' + ' ' -> 'в училище учитель учит ученикам по новому ' (Score: 1.0000)

Корпус после слияния: ['в училище учитель учит ученикам по новому ', 'у', 'чебник', 'у']

Текущий размер словаря: 61

Итерация 45:

Лучшая биграмма для слияния: 'в училище учитель учит ученикам по новому ' + 'у' -> 'в училище учитель учит ученикам по новому у' (Score: 0.5000)

Корпус после слияния: ['в училище учитель учит ученикам по новому у', 'чебник', 'у']

Текущий размер словаря: 62

Итерация 46:

Лучшая биграмма для слияния: 'в училище учитель учит ученикам по новому у' + 'чебник' -> 'в училище учитель учит ученикам по новому учебник' (Score: 1.0000)

Корпус после слияния: ['в училище учитель учит ученикам по новому учебник', 'у']

Текущий размер словаря: 63

Итерация 47:

Лучшая биграмма для слияния: 'в училище учитель учит ученикам по новому учебник' + 'у' -> 'в училище учитель учит ученикам по новому учебнику' (Score: 1.0000)

Корпус после слияния: ['в училище учитель учит ученикам по новому учебнику']

Текущий размер словаря: 64

Нет больше биграмм для слияния. Остановка.

Обучение завершено. Итоговый словарь содержит 64 токенов.

Итоговый словарь WordPiece: [' ', 'а', 'б', 'бн', 'в', 'в ', 'в у', 'в училище ', 'в училище у', 'в училище учитель ', 'в училище учитель у', 'в училище учитель учит ', 'в училище учитель учит у', 'в училище учитель учит ученикам по новом', 'в училище учитель учит ученикам по новому', 'в училище учитель учит ученикам по новому ', 'в училище учитель учит ученикам по новому у', 'в училище учитель учит ученикам по новому учебник', 'в училище учитель учит ученикам по новому учебнику', 'е', 'ебн', 'ель', 'ен', 'и', 'к', 'ка', 'кам', 'л', 'ль', 'м', 'н', 'новом', 'о', 'ов', 'овом', 'ом', 'п', 'по', 'т', 'тель', 'у', 'ч', 'чебн', 'чебни', 'чебник', 'чен', 'чени', 'ченикам', 'ченикам ', 'ченикам по', 'ченикам по ', 'ченикам по новом', 'чи', 'чил', 'чили', 'чилище', 'чилище ', 'чит', 'чит ', 'читель', 'читель ', 'щ', 'ще', 'ь']

--- Токенизация нового текста ---

Текст для токенизации: 'В училище учитель учит ученикам по новому учебнику'

Токенизированный текст: ['в', ' ', 'у', 'чилище', ' ', 'у', 'читель', ' ', 'у', 'чит', ' ', 'у', 'ченикам', ' ', 'по', ' ', 'новом', 'у', ' ', 'у', 'чебник', 'у']

Итоговый словарь WordPiece (с 'ница' для демонстрации): [' ', '##ница', 'а', 'б', 'бн', 'в', 'в ', 'в у', 'в училище ', 'в училище у', 'в училище учитель ', 'в училище учитель у', 'в училище учитель учит ', 'в училище учитель учит у', 'в училище учитель учит ученикам по новом', 'в училище учитель учит ученикам по новому', 'в училище учитель учит ученикам по новому ', 'в училище учитель учит ученикам по новому у', 'в училище учитель учит ученикам по новому учебник', 'в училище учитель учит ученикам по новому учебнику', 'е', 'ебн', 'ель', 'ен', 'и', 'к', 'ка', 'кам', 'л', 'ль', 'м', 'н', 'ница', 'новом', 'о', 'ов', 'овом', 'ом', 'п', 'по', 'т', 'тель', 'у', 'ч', 'чебн', 'чебни', 'чебник', 'чен', 'чени', 'ченикам', 'ченикам ', 'ченикам по', 'ченикам по ', 'ченикам по новом', 'чи', 'чил', 'чили', 'чилище', 'чилище ', 'чит', 'чит ', 'читель', 'читель ', 'щ', 'ще', 'ь']

--- Токенизация нового текста ---

Текст для токенизации: 'учительница'

Токенизированный текст: ['у', 'читель', 'ница']

--- Демонстрация токенизации подслов с упрощенным словарем ---

--- Токенизация нового текста ---

Текст для токенизации: 'учительница'

Токенизированный текст: ['учитель', 'н', 'и', 'ц', 'а']

### Униграмная Языковая Модель: Фундаментальные Принципы и Применение в НЛП

Униграмная языковая модель (Unigram Language Model) — это самая простая и базовая вероятностная модель языка. Она не учитывает порядок слов и контекст, но служит важным теоретическим и практическим фундаментом для понимания более сложных моделей, таких как биграммы, триграммы и нейросетевые языковые модели.

⚠️ Важно: Униграмная модель не выполняет токенизацию. Токенизация — это предварительный этап, в ходе которого текст разбивается на токены (слова, символы и т.д.). Униграмная модель работает с уже токенизированными данными, оценивая вероятности отдельных токенов.

Основные Концепции Униграмной Модели

Центральное предположение униграмной модели — независимость слов. Модель считает, что каждое слово в тексте появляется независимо от других слов. То есть:

* Вероятность слова не зависит от предыдущих или последующих слов.
* Порядок слов в последовательности не имеет значения.

Формально, вероятность последовательности слов в униграмной модели вычисляется как:

где — вероятность отдельного слова , оцениваемая по частоте его встречаемости в обучающем корпусе:

Пример: Рассмотрим предложение: *"кот сидит на коврике"*.  
Униграмная модель оценит его вероятность как:

При этом модель не различает, является ли это осмысленным предложением или просто случайной перестановкой слов — например, *"на сидит коврике кот"* будет иметь ту же вероятность.

Области Применения Униграмных Моделей

Несмотря на упрощённое предположение о независимости слов, униграмные модели находят практическое применение в задачах, где частота слов является ключевым фактором, а контекст — вторичным.

Базовая оценка вероятности текста

Униграмная модель может дать грубую оценку "естественности" текста. Например, в задачах определения авторства или жанра можно сравнивать распределения слов в разных текстах. Хотя это не учитывает синтаксис, оно может выявить стилистические особенности (например, частота местоимений или глаголов).

Тематическое моделирование (Topic Modeling)

Униграмные модели позволяют выявить наиболее частотные слова в документе. Эти слова часто коррелируют с основной темой. Например, высокая частота слов *"футбол"*, *"гол"*, *"матч"* указывает на спортивную тематику. Этот подход лежит в основе простых методов, таких как TF-IDF и мешок слов (Bag-of-Words).

Спам-фильтрация

В классических фильтрах (например, на основе наивного байесовского классификатора) униграмные вероятности используются для оценки того, насколько "спамовым" является письмо. Если вероятность слов *"выигрыш"*, *"бесплатно"*, *"акция"* в спам-корпусе высока, то письмо с их высокой частотой будет классифицировано как спам.

Поиск информации (Information Retrieval)

В простых системах поиска (например, в ранних версиях TF-IDF) униграмные частоты слов используются для оценки релевантности документа запросу. Документ, содержащий больше слов из запроса, считается более релевантным — независимо от порядка слов.

Оценка сложности текста

Униграмные модели могут применяться для приблизительной оценки читабельности или сложности текста. Например:

Высокая частота редких слов → более сложный текст.

Большое количество уникальных слов (высокое лексическое разнообразие) → выше сложность.

Математические Формулы Униграмной Модели

Вероятность последовательности слов

Пусть дана последовательность слов .

В униграмной языковой модели предполагается, что слова появляются независимо друг от друга. Это позволяет выразить вероятность всей последовательности как произведение вероятностей отдельных слов:

Где:

— вероятность слова в языковой модели,

— знак произведения по всем словам в последовательности.

⚠️ Важно: Это предположение игнорирует как синтаксис, так и семантику. Например, последовательности *"кошка спит на диване"* и *"на спит диване кошка"* будут иметь одинаковую вероятность, если все слова одинаковы.

Оценка вероятности слова: Максимальное правдоподобие (MLE)

На практике вероятности слов неизвестны и должны быть оценены на основе обучающего корпуса. Наиболее распространённый способ — метод максимального правдоподобия (Maximum Likelihood Estimation, MLE).

Согласно MLE, вероятность слова оценивается как его относительная частота в обучающем корпусе:

Где:

* — количество вхождений слова в обучающий корпус,
* — общее количество слов в корпусе,
* — словарь корпуса (множество всех уникальных слов).

Интуиция: Если слово *"кот"* встречается 10 раз в корпусе из 1000 слов, то его оценённая вероятность будет:

Это значение "максимизирует правдоподобие" обучающих данных — то есть делает наблюдаемый корпус наиболее вероятным при заданных параметрах модели.

Проблема нулевых частот и сглаживание (Smoothing)

Критическая проблема MLE:

Если слово не встречалось в обучающем корпусе, то , и, следовательно, .

Это приводит к катастрофической ошибке: любая последовательность, содержащая это слово, будет иметь нулевую вероятность:

Модель становится необобщаемой — она не может обрабатывать ни одно новое слово, что делает её бесполезной в реальных условиях.

Решение: Сглаживание (Smoothing)

Чтобы решить эту проблему, применяются методы сглаживания — они перераспределяют часть вероятностной массы от наблюдаемых слов к ненаблюдаемым, гарантируя, что все возможные слова имеют ненулевую вероятность.

Сглаживание по Лапласу (Laplace Smoothing / Add-One Smoothing)

Один из самых простых методов — сглаживание по Лапласу, при котором к счётчику каждого слова (включая не встретившиеся) добавляется 1:

Где:

* — частота слова в корпусе,
* — общее количество слов в корпусе,
* — размер словаря (число уникальных слов в корпусе).

⚠️ Примечание: В знаменателе добавляется , потому что мы "добавляем по единице" для каждого из возможных слов.

Интуиция сглаживания по Лапласу:

Представим, что каждое возможное слово из словаря встретилось в корпусе хотя бы один раз. Это искусственно "расширяет" корпус и предотвращает нулевые вероятности.

Пример: Корпус: ["кот", "спит", "кот", "еда"]

, (слова: "кот", "спит", "еда", и предполагаемое "новое")

Теперь даже незнакомые слова имеют шанс.

Недостатки сглаживания по Лапласу:

Сильно искажает вероятности частотных слов, особенно при большом .

Не подходит для больших словарей (например, в языковых моделях с десятками тысяч слов), так как "перераспределяет слишком много" вероятности.

Поэтому на практике чаще используются более продвинутые методы: Good-Turing, Witten-Bell, Kneser-Ney smoothing.

Конкретный Пример Применения Униграмной Модели

Рассмотрим применение униграмной языковой модели к следующему тексту:  
"Ученики пишут диктант. Учитель диктует. Мы пишем, он пишет, они пишут. Пишите внимательно."

Шаг 1: Токенизация и подготовка корпуса. Перед построением модели необходимо предварительно обработать текст:

* Привести к нижнему регистру,
* Удалить знаки препинания,
* Разделить на слова (токены).

Исходный текст: "Ученики пишут диктант. Учитель диктует. Мы пишем, он пишет, они пишут. Пишите внимательно."

Токенизированный корпус (после обработки):  
['ученики', 'пишут', 'диктант', 'учитель', 'диктует', 'мы', 'пишем', 'он', 'пишет', 'они', 'пишут', 'пишите', 'внимательно']

Общее количество слов в корпусе (N):

Шаг 2: Подсчёт частот слов. Составим таблицу частот для каждого уникального слова:

|  |  |
| --- | --- |
| Слово | Частота |
| ученики | 1 |
| Пишут | 2 |
| диктант | 1 |
| Учитель | 1 |
| Диктует | 1 |
| Мы | 1 |
| Пишем | 1 |
| Он | 1 |
| Пишет | 1 |
| Они | 1 |
| Пишите | 1 |
| внимательно | 1 |

⚠️ Обратите внимание: слово "пишут" встречается дважды — в начале и в середине текста.

Шаг 3: Определение словаря и его размера. Словарь — множество всех уникальных слов в корпусе:

Размер словаря :

Шаг 4: Расчёт вероятностей слов (MLE). Используем метод максимального правдоподобия (MLE):

Вычислим вероятности:

|  |  |  |
| --- | --- | --- |
| Слово |  |  |
| Ученики | 1 |  |
| Пишут | 2 |  |
| Диктант | 1 |  |
| Учитель | 1 |  |
| Диктует | 1 |  |
| Мы | 1 |  |
| Пишем | 1 |  |
| Он | 1 |  |
| Пишет | 1 |  |
| Они | 1 |  |
| Пишите | 1 |  |
| внимательно | 1 |  |

Слово "пишут" имеет наибольшую вероятность () из-за своей высокой частоты.

Шаг 5: Расчёт вероятности предложения. Оценим вероятность всей последовательности (все слова из корпуса в порядке их появления):

Распишем произведение:

Подставим значения:

Упростим:

Вычислим:

⚠️ Несмотря на то что текст осмысленный, его вероятность крайне мала — это типично для униграмных моделей из-за умножения множества малых вероятностей.

Пример со сглаживанием по Лапласу

Предположим, мы хотим оценить вероятность слова "школа", которого нет в обучающем корпусе.

Без сглаживания (MLE):

Это означает, что любое предложение, содержащее слово "школа", будет иметь нулевую вероятность.

Со сглаживанием по Лапласу:

Используем формулу:

Подставим значения:

Теперь слово "школа" имеет ненулевую вероятность, и модель может обрабатывать новые, ранее не встречавшиеся слова. Это делает её более робастной в реальных условиях.

Преимущества и Недостатки Униграмной Модели

Несмотря на свою простоту, униграмная языковая модель обладает как сильными сторонами, так и серьёзными ограничениями. Понимание этих аспектов помогает оценить её место в иерархии языковых моделей и определить области, где она остаётся полезной.

Преимущества

* Простота и интуитивность. Униграмная модель основана на простой идее: частота слова напрямую отражает его вероятность. Это делает модель крайне простой для понимания, объяснения и реализации. Она служит отличной обучающей платформой для изучения основ языкового моделирования.
* Вычислительная эффективность. Обучение модели сводится к подсчёту частот слов — операции со сложностью , где — размер корпуса. Использование модели (оценка вероятности последовательности) также требует лишь умножения вероятностей. Это делает униграмные модели очень быстрыми и пригодными для обработки огромных объёмов данных или встраивания в системы с жёсткими требованиями к производительности.
* Эффективная оценка частот слов. Модель идеально подходит для задач, где ключевую роль играют частоты отдельных слов, а порядок не важен. Это включает:
* тематическое моделирование,
* фильтрацию спама,
* предварительный анализ текста,
* построение словарей и оценку лексического разнообразия.

Отсутствие проблемы разреженности (в сравнении с высокими N-граммами)  
В отличие от биграмм или триграмм, где многие возможные комбинации слов отсутствуют в корпусе (что приводит к разреженности данных), униграмная модель работает с отдельными словами. Для частотных слов проблема разреженности минимальна, так как они, как правило, наблюдаются в корпусе.

Недостатки

* Полное игнорирование контекста и порядка слов.   
  Это — главный и наиболее критичный недостаток. Униграмная модель не учитывает ни синтаксис, ни семантику. Она не различает осмысленные и бессмысленные последовательности, если они содержат одни и те же слова.  
  Например, предложения *"собака укусила человека"* и *"человек укусил собаку"* будут иметь одинаковую вероятность, хотя их смысл диаметрально противоположен.
* Нереалистичное предположение о независимости слов. Естественный язык — это последовательность взаимозависимых слов. Вероятность появления слова *"кофе"* сильно возрастает после *"я пью"*, но почти нулевая после *"я лаю"*. Униграмная модель не может улавливать такие зависимости, что делает её слишком грубой для задач, требующих понимания смысла.
* Проблема нулевых вероятностей (OOV-слова). Без применения сглаживания любое слово, не встретившееся в обучающем корпусе, получает вероятность 0. Это означает, что любое предложение с новым словом будет иметь нулевую вероятность — модель необобщаема. Хотя сглаживание решает эту проблему частично, оно вносит искажения в распределение.
* Ограниченная применимость в современном НЛП. Из-за отсутствия контекстной чувствительности униграмные модели не подходят для большинства современных задач, таких как:
* генерация текста,
* машинный перевод,
* вопросно-ответные системы,
* распознавание речи,
* суммаризация.

Эти задачи требуют понимания последовательности, зависимостей и смысла — чего униграмная модель дать не может.

Заключение

Униграмная языковая модель — это упрощённая, но важная абстракция, которая демонстрирует базовые принципы вероятностного моделирования в НЛП. Её сила — в простоте, скорости и интерпретируемости. Её слабость — в полном игнорировании контекста.

Однако именно эти ограничения стали двигателем прогресса в области языкового моделирования:

* от униграммы → к биграммам и триграммам (с учётом контекста),
* от n-грамм → к нейросетевым моделям (RNN, LSTM, трансформерам),
* от частотных подходов → к распределённым представлениям слов (word embeddings, BERT и др.).

Таким образом, униграмная модель — не инструмент для современных сложных систем, а необходимая ступень на пути к ним. Она помогает понять, почему контекст важен, и почему более сложные модели стали неизбежными в эпоху глубокого обучения.

Реализация на Питон:

|  |
| --- |
| import collections  import re  import math  class UnigramLanguageModel:      """      Реализация Униграмной Языковой Модели.      Эта модель предполагает, что вероятность каждого слова в последовательности      не зависит от других слов.      """      def \_\_init\_\_(self):          self.word\_counts = collections.defaultdict(int) # Счетчик вхождений каждого слова          self.total\_words = 0 # Общее количество слов в корпусе          self.vocabulary = set() # Множество уникальных слов (словарь)      def \_tokenize(self, text):          """          Приватный метод для токенизации текста.          Приводит текст к нижнему регистру и разбивает на слова, удаляя знаки препинания.          :param text: Входная строка текста.          :return: Список токенов (слов).          """          # Приводим к нижнему регистру          text = text.lower()          # Удаляем знаки препинания (кроме тех, что могут быть частью слова, но для простоты здесь удаляем все)          # Использование регулярного выражения для извлечения только буквенных последовательностей          tokens = re.findall(r'\b[а-яё]+\b', text)          return tokens      def train(self, corpus\_text):          """          Обучает униграмную языковую модель на заданном текстовом корпусе.          Подсчитывает частоты слов и формирует словарь.          :param corpus\_text: Строка, представляющая обучающий корпус.          """          print("--- Начало обучения Униграмной Модели ---")          tokens = self.\_tokenize(corpus\_text)            if not tokens:              print("Предупреждение: Корпус пуст или не содержит слов после токенизации.")              return          self.total\_words = len(tokens)            for word in tokens:              self.word\_counts[word] += 1              self.vocabulary.add(word)            print(f"Обучение завершено. Общее количество слов (N): {self.total\_words}")          print(f"Размер словаря (|V|): {len(self.vocabulary)}")          print(f"Частоты слов: {dict(self.word\_counts)}")          print("--- Обучение завершено ---")      def get\_word\_probability(self, word, smoothing=None):          """          Возвращает вероятность слова P(w).          :param word: Слово, для которого нужно рассчитать вероятность.          :param smoothing: Метод сглаживания ('laplace' для сглаживания по Лапласу, None для MLE).          :return: Вероятность слова.          """          # Если модель не обучена или корпус пуст          if self.total\_words == 0:              print("Ошибка: Модель не обучена или корпус пуст. Невозможно рассчитать вероятность.")              return 0.0          count\_w = self.word\_counts[word] # Count(w)          if smoothing == 'laplace':              # Сглаживание по Лапласу: (Count(w) + 1) / (N + |V|)              probability = (count\_w + 1) / (self.total\_words + len(self.vocabulary))              # print(f"P\_Laplace('{word}') = ({count\_w} + 1) / ({self.total\_words} + {len(self.vocabulary)}) = {probability:.4f}")          else:              # Метод максимального правдоподобия (MLE): Count(w) / N              if count\_w == 0:                  # print(f"P\_MLE('{word}') = 0 (слово не найдено в корпусе)")                  return 0.0 # Если слово не найдено, вероятность 0 без сглаживания              probability = count\_w / self.total\_words              # print(f"P\_MLE('{word}') = {count\_w} / {self.total\_words} = {probability:.4f}")            return probability      def get\_sequence\_probability(self, sequence\_text, smoothing=None):          """          Возвращает вероятность последовательности слов P(W).          :param sequence\_text: Строка, представляющая последовательность слов.          :param smoothing: Метод сглаживания ('laplace' для сглаживания по Лапласу, None для MLE).          :return: Вероятность последовательности.          """          print(f"\n--- Расчет вероятности последовательности: '{sequence\_text}' ---")          tokens = self.\_tokenize(sequence\_text)            if not tokens:              print("Предупреждение: Последовательность пуста или не содержит слов после токенизации.")              return 0.0          sequence\_probability = 1.0          probabilities\_list = []          for word in tokens:              p\_word = self.get\_word\_probability(word, smoothing)              probabilities\_list.append(p\_word)              sequence\_probability \*= p\_word                # Если хоть одно слово имеет нулевую вероятность без сглаживания,              # вся последовательность будет иметь нулевую вероятность.              if smoothing != 'laplace' and p\_word == 0:                  print(f"Слово '{word}' имеет нулевую вероятность. Вероятность всей последовательности = 0.")                  return 0.0 # Оптимизация: если P(w)=0, то произведение будет 0          print(f"Токены последовательности: {tokens}")          print(f"Вероятности отдельных слов: {probabilities\_list}")          print(f"Итоговая вероятность последовательности: {sequence\_probability}")          print("--- Расчет завершен ---")          return sequence\_probability  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":      corpus\_example = "Ученики пишут диктант. Учитель диктует. Мы пишем, он пишет, они пишут. Пишите внимательно."      # Создаем экземпляр модели      unigram\_model = UnigramLanguageModel()      # Обучаем модель на корпусе      unigram\_model.train(corpus\_example)      print("\n--- Расчет вероятностей отдельных слов (MLE) ---")      words\_to\_check = ['пишут', 'учитель', 'школа', 'мы']      for word in words\_to\_check:          prob = unigram\_model.get\_word\_probability(word, smoothing=None)          print(f"P\_MLE('{word}') = {prob:.4f}")      print("\n--- Расчет вероятностей отдельных слов (Лаплас) ---")      for word in words\_to\_check:          prob = unigram\_model.get\_word\_probability(word, smoothing='laplace')          print(f"P\_Laplace('{word}') = {prob:.4f}")      # Расчет вероятности предложения (без сглаживания)      sentence\_mle = "Ученики пишут диктант"      prob\_mle = unigram\_model.get\_sequence\_probability(sentence\_mle, smoothing=None)      print(f"Вероятность предложения (MLE): {prob\_mle:.10f}")      sentence\_mle\_oov = "Ученики пишут школа" # Содержит OOV слово 'школа'      prob\_mle\_oov = unigram\_model.get\_sequence\_probability(sentence\_mle\_oov, smoothing=None)      print(f"Вероятность предложения с OOV (MLE): {prob\_mle\_oov:.10f}")      # Расчет вероятности предложения (со сглаживанием по Лапласу)      sentence\_laplace = "Ученики пишут диктант"      prob\_laplace = unigram\_model.get\_sequence\_probability(sentence\_laplace, smoothing='laplace')      print(f"Вероятность предложения (Лаплас): {prob\_laplace:.10f}")      sentence\_laplace\_oov = "Ученики пишут школа" # Содержит OOV слово 'школа'      prob\_laplace\_oov = unigram\_model.get\_sequence\_probability(sentence\_laplace\_oov, smoothing='laplace')      print(f"Вероятность предложения с OOV (Лаплас): {prob\_laplace\_oov:.10f}")      # Пример из документации: "Ученики пишут диктант Учитель диктует Мы пишем он пишет они пишут Пишите внимательно"      full\_sentence = "Ученики пишут диктант Учитель диктует Мы пишем он пишет они пишут Пишите внимательно"      prob\_full\_mle = unigram\_model.get\_sequence\_probability(full\_sentence, smoothing=None)      print(f"Вероятность полного предложения (MLE): {prob\_full\_mle:.15f}")      prob\_full\_laplace = unigram\_model.get\_sequence\_probability(full\_sentence, smoothing='laplace')      print(f"Вероятность полного предложения (Лаплас): {prob\_full\_laplace:.15f}") |

Результат:

--- Начало обучения Униграмной Модели ---

Обучение завершено. Общее количество слов (N): 13

Размер словаря (|V|): 12

Частоты слов: {'ученики': 1, 'пишут': 2, 'диктант': 1, 'учитель': 1, 'диктует': 1, 'мы': 1, 'пишем': 1, 'он': 1, 'пишет': 1, 'они': 1, 'пишите': 1, 'внимательно': 1}

--- Обучение завершено ---

--- Расчет вероятностей отдельных слов (MLE) ---

P\_MLE('пишут') = 0.1538

P\_MLE('учитель') = 0.0769

P\_MLE('школа') = 0.0000

P\_MLE('мы') = 0.0769

--- Расчет вероятностей отдельных слов (Лаплас) ---

P\_Laplace('пишут') = 0.1200

P\_Laplace('учитель') = 0.0800

P\_Laplace('школа') = 0.0400

P\_Laplace('мы') = 0.0800

--- Расчет вероятности последовательности: 'Ученики пишут диктант' ---

Токены последовательности: ['ученики', 'пишут', 'диктант']

Вероятности отдельных слов: [0.07692307692307693, 0.15384615384615385, 0.07692307692307693]

Итоговая вероятность последовательности: 0.000910332271279017

--- Расчет завершен ---

Вероятность предложения (MLE): 0.0009103323

--- Расчет вероятности последовательности: 'Ученики пишут школа' ---

Слово 'школа' имеет нулевую вероятность. Вероятность всей последовательности = 0.

Вероятность предложения с OOV (MLE): 0.0000000000

--- Расчет вероятности последовательности: 'Ученики пишут диктант' ---

Токены последовательности: ['ученики', 'пишут', 'диктант']

Вероятности отдельных слов: [0.08, 0.12, 0.08]

Итоговая вероятность последовательности: 0.0007679999999999999

--- Расчет завершен ---

Вероятность предложения (Лаплас): 0.0007680000

--- Расчет вероятности последовательности: 'Ученики пишут школа' ---

Токены последовательности: ['ученики', 'пишут', 'школа']

Вероятности отдельных слов: [0.08, 0.12, 0.04]

Итоговая вероятность последовательности: 0.00038399999999999996

--- Расчет завершен ---

Вероятность предложения с OOV (Лаплас): 0.0003840000

--- Расчет вероятности последовательности: 'Ученики пишут диктант Учитель диктует Мы пишем он пишет они пишут Пишите внимательно' ---

Токены последовательности: ['ученики', 'пишут', 'диктант', 'учитель', 'диктует', 'мы', 'пишем', 'он', 'пишет', 'они', 'пишут', 'пишите', 'внимательно']

Вероятности отдельных слов: [0.07692307692307693, 0.15384615384615385, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.07692307692307693, 0.15384615384615385, 0.07692307692307693, 0.07692307692307693]

Итоговая вероятность последовательности: 1.3206763820920487e-14

--- Расчет завершен ---

Вероятность полного предложения (MLE): 0.000000000000013

--- Расчет вероятности последовательности: 'Ученики пишут диктант Учитель диктует Мы пишем он пишет они пишут Пишите внимательно' ---

Токены последовательности: ['ученики', 'пишут', 'диктант', 'учитель', 'диктует', 'мы', 'пишем', 'он', 'пишет', 'они', 'пишут', 'пишите', 'внимательно']

Вероятности отдельных слов: [0.08, 0.12, 0.08, 0.08, 0.08, 0.08, 0.08, 0.08, 0.08, 0.08, 0.12, 0.08, 0.08]

Итоговая вероятность последовательности: 1.2369505812480003e-14

--- Расчет завершен ---

Вероятность полного предложения (Лаплас): 0.000000000000012

1. Токенизация с использованием SentencePiece

Введение в токенизацию и SentencePiece

В области обработки естественного языка (NLP) токенизация является одним из первых и наиболее фундаментальных шагов в подготовке текстовых данных для дальнейшей обработки. Токенизация — это процесс разбиения текста на более мелкие единицы, называемые токенами. Эти токены могут быть словами, символами или, что чаще всего используется в современных моделях, подсловами (subwords).

Традиционные методы токенизации, основанные на пробелах или пунктуации, сталкиваются с проблемами в языках без явных разделителей слов (например, японский, китайский), а также при работе с незнакомыми словами (OOV – Out-Of-Vocabulary) или морфологически сложными языками. Для решения этих проблем были разработаны методы подсловарной токенизации, такие как Byte-Pair Encoding (BPE) и Unigram Language Model (ULM), которые позволяют моделям эффективно работать с большим словарным запасом и редкими словами.

SentencePiece — это инструмент для подсловарной токенизации, разработанный Google, который предлагает унифицированный подход к созданию моделей токенизации. Его ключевое отличие и преимущество заключается в том, что он работает непосредственно с необработанным текстом, включая пробелы, и не требует предварительной токенизации или специфических правил для разных языков. Это делает SentencePiece особенно мощным и гибким инструментом для многоязычных NLP-задач.

Особенности и преимущества SentencePiece

SentencePiece выделяется среди других методов токенизации благодаря нескольким уникальным особенностям:

Языковая агностичность (Language Agnostic)

Одной из главных проблем традиционной токенизации является ее зависимость от языковых правил. Например, в английском языке слова обычно разделяются пробелами, но в японском или китайском языках такой явной границы нет. SentencePiece решает эту проблему, рассматривая входной текст как последовательность символов, включая пробелы, которые он трактует как обычные символы. Это позволяет ему обучаться и работать с любым языком без необходимости предварительной обработки или знания языковых особенностей.

Пример: В SentencePiece пробел перед словом часто представляется как специальный символ (например, или \_). Это позволяет модели точно реконструировать исходный текст, включая пробелы.

Исходный текст: "Привет мир"

Токенизация SentencePiece (пример): [" Привет", " мир"]

Здесь (или \_) является частью токена, указывая на то, что это начало слова, ранее разделенного пробелом.

Унифицированная подсловарная сегментация

SentencePiece предоставляет унифицированный интерфейс для реализации различных алгоритмов подсловарной токенизации, включая

Byte-Pair Encoding (BPE): Хотя BPE подробно рассмотрен в отдельной главе, важно отметить, что SentencePiece может обучать и использовать модели BPE. Его реализация BPE отличается тем, что она работает на уровне символов, а не на уровне слов, и явно обрабатывает пробелы как часть символьной последовательности.

Unigram Language Model (ULM): Аналогично, SentencePiece поддерживает обучение моделей на основе Unigram Language Model. ULM, в отличие от BPE, который является жадным алгоритмом, использует вероятностный подход для сегментации текста, выбирая наиболее вероятную последовательность подслов.

Возможность выбора между этими алгоритмами в рамках одного инструмента делает SentencePiece очень гибким для различных задач и языков.

Прямая работа с необработанным текстом

SentencePiece устраняет необходимость в предварительной токенизации текста. Это означает, что вы можете подавать ему необработанный текст (например, целые предложения или документы) напрямую. Это упрощает конвейер обработки данных и снижает вероятность ошибок, которые могут возникнуть при многоэтапной токенизации.

Обработка неизвестных токенов (UNK)

Как и большинство методов подсловарной токенизации, SentencePiece эффективно справляется с незнакомыми словами (Out-Of-Vocabulary, OOV). Поскольку он разбивает слова на подслова, любое новое слово, не встречавшееся во время обучения, может быть разбито на известные подслова. Если даже подслова неизвестны, они будут представлены специальным токеном UNK (unknown).

Обратимость (Reversibility)

Одна из важных особенностей SentencePiece — это его способность восстанавливать исходный текст из токенов без потери информации. Это достигается благодаря тому, что SentencePiece явно обрабатывает пробелы как часть токенов.

Пример: Если у нас есть токены [" Привет", " мир"], мы можем легко объединить их, чтобы получить исходный текст "Привет мир". Это критически важно для задач, где требуется точное восстановление исходного текста, например, в генерации текста или машинного перевода.

Обучение моделей SentencePiece

Обучение модели SentencePiece включает в себя определение типа алгоритма (BPE или Unigram), размера словаря и входных данных.

Основные параметры обучения

Для обучения модели SentencePiece используются следующие ключевые параметры:

* --input: Путь к файлу(ам) с необработанным текстовым корпусом для обучения. Каждый абзац или предложение должен быть на новой строке.
* --model\_prefix: Префикс для выходных файлов модели. SentencePiece сгенерирует два файла: .model (сама модель) и .vocab (файл словаря).
* --vocab\_size: Желаемый размер словаря подслов. Это число определяет, сколько уникальных подслов будет в вашей модели.
* --model\_type: Тип алгоритма токенизации. Может быть bpe (Byte-Pair Encoding) или unigram (Unigram Language Model). unigram часто дает лучшие результаты для генерации текста, в то время как bpe может быть более предсказуемым.
* --character\_coverage: Доля символов, которые должны быть покрыты моделью. Например, 0.9995 означает, что 99.95% символов в обучающем корпусе должны быть представлены в словаре.

--num\_threads: Количество потоков для параллельного обучения.

Пример команды для обучения. Предположим, у нас есть файл corpus.txt с обучающим текстом.

spm\_train --input=corpus.txt --model\_prefix=my\_model --vocab\_size=8000 –

model\_type=unigram --character\_coverage=1.0

Эта команда обучит модель SentencePiece типа unigram с размером словаря 8000 подслов, используя corpus.txt в качестве входных данных. Результатом будут файлы my\_model.model и my\_model.vocab.

Использование обученной модели SentencePiece

После обучения модели вы можете использовать ее для кодирования (токенизации) и декодирования текста.

Кодирование текста

Кодирование текста означает преобразование исходного текста в последовательность подсловных токенов.

|  |
| --- |
| import sentencepiece as spm  import os  # === Шаг 1: Подготовка обучающего текста ===  # Создадим простой тренировочный текст  training\_text = """Это пример предложения на русском языке.  Учитель учит ученика в старом училище.  Мама мыла раму.  Сегодня хорошая погода и светит солнце."""  # Сохраняем текст в файл  with open("train.txt", "w", encoding="utf-8") as f:      f.write(training\_text)  # === Шаг 2: Обучение модели SentencePiece ===  spm.SentencePieceTrainer.train(      input='train.txt',      model\_prefix='spm\_model',         # имя модели будет spm\_model.model + spm\_model.vocab      vocab\_size=200,                   # можно увеличить, если больше данных      model\_type='bpe',                 # можно: 'bpe', 'unigram', 'char', 'word'      character\_coverage=1.0,           # 1.0 = 100% символов (важно для русского текста)      pad\_id=0,      unk\_id=1,      bos\_id=2,      eos\_id=3  )  # === Шаг 3: Загрузка обученной модели ===  sp = spm.SentencePieceProcessor()  sp.load("spm\_model.model")  # используем обученную модель  # === Шаг 4: Тестовое предложение ===  text = "Учитель учит ученика."  # Кодирование текста в pieces и ids  encoded\_pieces = sp.encode\_as\_pieces(text)  encoded\_ids = sp.encode\_as\_ids(text)  print("== Кодирование ==")  print("Токены (pieces):", encoded\_pieces)  print("Токены (ids):", encoded\_ids) |

Результат:

== Кодирование ==

Токены (pieces): ['▁Учитель', '▁учит', '▁ученика', '.']

Токены (ids): [90, 81, 93, 180]

Обратите внимание, как пробелы представлены в начале токенов, что является ключевой особенностью SentencePiece.

Декодирование текста

Декодирование — это обратный процесс, преобразование последовательности токенов или их ID обратно в исходный текст.

|  |
| --- |
| # Декодирование обратно в текст  decoded\_from\_pieces = sp.decode\_pieces(encoded\_pieces)  decoded\_from\_ids = sp.decode\_ids(encoded\_ids)  print("\n== Декодирование ==")  print("Из pieces:", decoded\_from\_pieces)  print("Из ids:", decoded\_from\_ids) |

Результат:

== Декодирование ==

Из pieces: Учитель учит ученика.

Из ids: Учитель учит ученика.

Как видно, исходный текст был точно восстановлен.

Обработка специальных токенов

SentencePiece может автоматически добавлять специальные токены, такие как <s> (начало предложения, BOS) и </s> (конец предложения, EOS), которые часто используются в моделях для обозначения границ последовательностей.

|  |
| --- |
| import sentencepiece as spm  # === Шаг 1: Подготовка обучающего текста ===  training\_text = """Это пример предложения на русском языке.  Учитель учит ученика в старом училище.  Мама мыла раму.  Сегодня хорошая погода и светит солнце."""  with open("train.txt", "w", encoding="utf-8") as f:      f.write(training\_text)  # === Шаг 2: Обучение модели (без add\_bos/add\_eos) ===  spm.SentencePieceTrainer.train(      input='train.txt',      model\_prefix='spm\_model',      vocab\_size=200,      model\_type='bpe',      character\_coverage=1.0,      pad\_id=0,      unk\_id=1,      bos\_id=2,      eos\_id=3  )  # === Шаг 3: Загрузка модели ===  sp = spm.SentencePieceProcessor()  sp.load("spm\_model.model")  # === Шаг 4: Использование модели с add\_bos и add\_eos ===  text = "Мама мыла раму."  # Кодирование с добавлением <s> и </s>  encoded\_ids = sp.encode(text, out\_type=int, add\_bos=True, add\_eos=True)  encoded\_pieces = sp.encode(text, out\_type=str, add\_bos=True, add\_eos=True)  print("== Кодирование с BOS и EOS ==")  print("Токены (ids):", encoded\_ids)  print("Токены (pieces):", encoded\_pieces)  # Декодирование  decoded\_text = sp.decode\_ids(encoded\_ids)  print("\n== Декодирование ==")  print("Восстановленный текст:", decoded\_text) |

Результат:

== Кодирование с BOS и EOS ==

Токены (ids): [2, 78, 79, 80, 180, 3]

Токены (pieces): ['<s>', '▁Мама', '▁мыла', '▁раму', '.', '</s>']

== Декодирование ==

Восстановленный текст: Мама мыла раму.

Примеры использования SentencePiece в различных языках

SentencePiece особенно полезен для языков, которые не используют пробелы для разделения слов, или для многоязычных моделей.

Применение SentencePiece в современных NLP-моделях

SentencePiece, благодаря своим уникальным особенностям, стал стандартом де-факто для токенизации во многих современных моделях обработки естественного языка, особенно в архитектурах на основе трансформеров. Его языковая агностичность и способность работать с необработанным текстом делают его идеальным выбором для многоязычных и крупномасштабных задач.

Вот некоторые области и конкретные модели, где SentencePiece находит широкое применение:

Нейронный машинный перевод (NMT): SentencePiece изначально был разработан для улучшения качества машинного перевода, позволяя моделям эффективно работать с различными языками и их морфологией. Он обеспечивает сопоставимую точность перевода, обучая подсловарные модели непосредственно из необработанных предложений.

Большие языковые модели (LLM): Многие современные большие языковые модели используют SentencePiece для токенизации. Например:

Llama (версии 1 и 2): В этих моделях используется адаптированный BPE, реализованный через библиотеку SentencePiece. Это позволяет эффективно обрабатывать текст и восстанавливать его без потерь.

Другие трансформерные модели: Такие модели, как ALBERT, XLNet, Marian и T5, широко используют SentencePiece для своей токенизации, часто в сочетании с алгоритмом Unigram.

Предварительно обученные языковые модели: SentencePiece активно применяется при разработке предварительно обученных моделей, особенно для языков с ограниченными ресурсами (low-resource languages), где нет обширных размеченных данных. Он позволяет создавать монолингвальные и мультилингвальные модели, расширяя их возможности на новые языки.

Универсальные NLP-системы: Благодаря своей способности обрабатывать текст на различных языках без специфических правил, SentencePiece является ключевым компонентом для создания сквозных (end-to-end) NLP-систем, которые могут работать с необработанными предложениями.

Использование SentencePiece в этих моделях подчеркивает его важность для достижения высокой производительности и гибкости в современных NLP-приложениях.

Сравнение с другими методами (кратко)

Хотя BPE и ULM являются основой подсловарной токенизации, SentencePiece предлагает более унифицированную и удобную в использовании реализацию. Его способность работать напрямую с необработанным текстом и его языковая агностичность делают его предпочтительным выбором для многих современных NLP-моделей, особенно для тех, которые работают с несколькими языками или требуют точного восстановления исходного текста. В отличие от некоторых реализаций BPE, SentencePiece не требует предварительной токенизации пробелами, что устраняет потенциальные ошибки и упрощает конвейер данных.

Заключение

SentencePiece является краеугольным камнем современной подсловарной токенизации в NLP. Его способность работать напрямую с необработанным текстом, языковая агностичность, поддержка различных алгоритмов (BPE, ULM) и обратимость делают его мощным и гибким инструментом. Понимание принципов работы SentencePiece и умение его эффективно использовать критически важны для любого студента или исследователя, работающего с большими текстовыми данными и современными нейронными сетями для обработки естественного языка.

Реализация на питон:

|  |
| --- |
| import sentencepiece as spm  import os  import tempfile  class SentencePieceWrapper:      """      Класс-обертка для демонстрации использования библиотеки SentencePiece.      Позволяет обучать модель, токенизировать и детокенизировать текст.      """      def \_\_init\_\_(self):          self.sp\_processor = spm.SentencePieceProcessor()          self.model\_prefix = None          self.model\_path = None          self.vocab\_path = None          self.\_is\_model\_loaded = False # Добавляем флаг для отслеживания состояния загрузки модели      def train(self, text\_data, model\_prefix="my\_sp\_model", vocab\_size=8000, model\_type='unigram', character\_coverage=1.0):          """          Обучает модель SentencePiece на заданном текстовом корпусе.          Создает временный файл для обучения и сохраняет модель.          :param text\_data: Строка, содержащая обучающий текст.          :param model\_prefix: Префикс для имен файлов модели (model.model и model.vocab).          :param vocab\_size: Желаемый размер словаря подслов.          :param model\_type: Алгоритм обучения ('unigram' или 'bpe').          :param character\_coverage: Процент символов, которые должны быть покрыты моделью.          """          print(f"--- Начинается обучение модели SentencePiece ---")          print(f"Текст для обучения (фрагмент): '{text\_data[:100]}...'")          # Создаем временный файл для обучения          # NamedTemporaryFile гарантирует уникальное имя и автоматическое удаление при закрытии          with tempfile.NamedTemporaryFile(mode="w", encoding="utf-8", delete=False) as temp\_file:              temp\_file.write(text\_data)              train\_file\_name = temp\_file.name          self.model\_prefix = model\_prefix          self.model\_path = f"{model\_prefix}.model"          self.vocab\_path = f"{model\_prefix}.vocab"          try:              print(f"Обучение с параметрами: vocab\_size={vocab\_size}, model\_type='{model\_type}'")              spm.SentencePieceTrainer.train(                  input=train\_file\_name,                  model\_prefix=self.model\_prefix,                  vocab\_size=vocab\_size,                  character\_coverage=character\_coverage,                  model\_type=model\_type              )              print(f"Обучение завершено. Созданы файлы: '{self.model\_path}' и '{self.vocab\_path}'")          except Exception as e:              print(f"Ошибка при обучении модели SentencePiece: {e}")              # Попытаемся удалить созданные файлы, если обучение не удалось              self.\_cleanup\_files()              raise # Перевыбрасываем исключение          finally:              # Удаляем временный файл после обучения              if os.path.exists(train\_file\_name):                  os.remove(train\_file\_name)                  print(f"Временный файл обучения удален: '{train\_file\_name}'")          # Загружаем обученную модель          self.load\_model(self.model\_path)          print("--- Обучение и загрузка модели завершены ---")      def load\_model(self, model\_path):          """          Загружает обученную модель SentencePiece из файла.          :param model\_path: Путь к файлу модели (.model).          """          print(f"--- Загрузка модели SentencePiece ---")          try:              self.sp\_processor.load(model\_path)              self.\_is\_model\_loaded = True # Устанавливаем флаг после успешной загрузки              print(f"Модель SentencePiece успешно загружена из: '{model\_path}'")          except Exception as e:              self.\_is\_model\_loaded = False # Сбрасываем флаг в случае ошибки              print(f"Ошибка при загрузке модели SentencePiece: {e}")              raise # Перевыбрасываем исключение          print("--- Загрузка модели завершена ---")      def encode\_as\_ids(self, text):          """          Кодирует текст в последовательность числовых ID токенов.          :param text: Входная строка текста.          :return: Список числовых ID токенов.          """          if not self.\_is\_model\_loaded: # Используем наш флаг              raise ValueError("Модель SentencePiece не загружена. Сначала обучите или загрузите модель.")            print(f"\n--- Кодирование текста в ID ---")          print(f"Исходный текст: '{text}'")          encoded\_ids = self.sp\_processor.encode\_as\_ids(text)          print(f"Закодированные ID: {encoded\_ids}")          return encoded\_ids      def encode\_as\_pieces(self, text):          """          Кодирует текст в последовательность строковых токенов (подслов).          :param text: Входная строка текста.          :return: Список строковых токенов.          """          if not self.\_is\_model\_loaded: # Используем наш флаг              raise ValueError("Модель SentencePiece не загружена. Сначала обучите или загрузите модель.")            print(f"\n--- Кодирование текста в подслова ---")          print(f"Исходный текст: '{text}'")          encoded\_pieces = self.sp\_processor.encode\_as\_pieces(text)          print(f"Закодированные подслова: {encoded\_pieces}")          return encoded\_pieces      def decode\_ids(self, ids):          """          Декодирует последовательность числовых ID токенов обратно в строку.          :param ids: Список числовых ID токенов.          :return: Декодированная строка.          """          if not self.\_is\_model\_loaded: # Используем наш флаг              raise ValueError("Модель SentencePiece не загружена. Сначала обучите или загрузите модель.")            print(f"\n--- Декодирование ID в текст ---")          print(f"ID для декодирования: {ids}")          decoded\_text = self.sp\_processor.decode\_ids(ids)          print(f"Декодированный текст: '{decoded\_text}'")          return decoded\_text      def decode\_pieces(self, pieces):          """          Декодирует последовательность строковых токенов (подслов) обратно в строку.          :param pieces: Список строковых токенов.          :return: Декодированная строка.          """          if not self.\_is\_model\_loaded: # Используем наш флаг              raise ValueError("Модель SentencePiece не загружена. Сначала обучите или загрузите модель.")            print(f"\n--- Декодирование подслов в текст ---")          print(f"Подслова для декодирования: {pieces}")          decoded\_text = self.sp\_processor.decode\_pieces(pieces)          print(f"Декодированный текст: '{decoded\_text}'")          return decoded\_text      def \_cleanup\_files(self):          """          Удаляет файлы модели SentencePiece (.model и .vocab), если они существуют.          """          print(f"\n--- Очистка временных файлов модели ---")          if self.model\_path and os.path.exists(self.model\_path):              os.remove(self.model\_path)              print(f"Удален файл модели: '{self.model\_path}'")          if self.vocab\_path and os.path.exists(self.vocab\_path):              os.remove(self.vocab\_path)              print(f"Удален файл словаря: '{self.vocab\_path}'")  # --- Пример использования ---  if \_\_name\_\_ == "\_\_main\_\_":      # Пример текста для обучения      corpus\_text = """Учитель говорит: «Откройте учебники». Ученики открывают учебники. Откройте страницу 15. Страница сложная. Страницы учебника содержат много информации."""      # Создаем экземпляр класса SentencePieceWrapper      sp\_wrapper = SentencePieceWrapper()      try:          # Обучаем модель с заданным размером словаря (для демонстрации)          sp\_wrapper.train(corpus\_text, vocab\_size=50, model\_type='unigram')          # Тестируем кодирование и декодирование          input\_text\_1 = "Ученики открывают учебники."          encoded\_ids\_1 = sp\_wrapper.encode\_as\_ids(input\_text\_1)          decoded\_text\_1 = sp\_wrapper.decode\_ids(encoded\_ids\_1)          print(f"\nПроверка обратимости 1: Исходный: '{input\_text\_1}', Декодированный: '{decoded\_text\_1}'")          assert input\_text\_1 == decoded\_text\_1, "Ошибка обратимости для input\_text\_1"          input\_text\_2 = "Страница сложная."          encoded\_pieces\_2 = sp\_wrapper.encode\_as\_pieces(input\_text\_2)          decoded\_text\_2 = sp\_wrapper.decode\_pieces(encoded\_pieces\_2)          print(f"\nПроверка обратимости 2: Исходный: '{input\_text\_2}', Декодированный: '{decoded\_text\_2}'")          assert input\_text\_2 == decoded\_text\_2, "Ошибка обратимости для input\_text\_2"          # Пример с OOV-словом (если оно не было полностью обучено)          oov\_text = "Преподавательница читает."          print(f"\n--- Тестирование OOV-слова ---")          encoded\_oov\_ids = sp\_wrapper.encode\_as\_ids(oov\_text)          decoded\_oov\_text = sp\_wrapper.decode\_ids(encoded\_oov\_ids)          print(f"Исходный OOV: '{oov\_text}', Закодированные ID: {encoded\_oov\_ids}, Декодированный: '{decoded\_oov\_text}'")      except Exception as e:          print(f"Произошла ошибка в процессе выполнения: {e}")      finally:          # Очищаем созданные файлы модели          sp\_wrapper.\_cleanup\_files() |

Результат:

--- Начинается обучение модели SentencePiece ---

Текст для обучения (фрагмент): 'Учитель говорит: «Откройте учебники». Ученики открывают учебники. Откройте страницу 15. Страница сло...'

Обучение с параметрами: vocab\_size=50, model\_type='unigram'

Обучение завершено. Созданы файлы: 'my\_sp\_model.model' и 'my\_sp\_model.vocab'

Временный файл обучения удален: '/tmp/tmpq62ne1qg'

--- Загрузка модели SentencePiece ---

Модель SentencePiece успешно загружена из: 'my\_sp\_model.model'

--- Загрузка модели завершена ---

--- Обучение и загрузка модели завершены ---

--- Кодирование текста в ID ---

Исходный текст: 'Ученики открывают учебники.'

Закодированные ID: [22, 12, 42, 3, 4, 8, 17, 15, 5, 48, 13, 24, 6]

--- Декодирование ID в текст ---

ID для декодирования: [22, 12, 42, 3, 4, 8, 17, 15, 5, 48, 13, 24, 6]

Декодированный текст: 'Ученики открывают учебники.'

Проверка обратимости 1: Исходный: 'Ученики открывают учебники.', Декодированный: 'Ученики открывают учебники.'

--- Кодирование текста в подслова ---

Исходный текст: 'Страница сложная.'

Закодированные подслова: ['▁Страниц', 'а', '▁с', 'л', 'о', 'ж', 'н', 'а', 'я', '.']

--- Декодирование подслов в текст ---

Подслова для декодирования: ['▁Страниц', 'а', '▁с', 'л', 'о', 'ж', 'н', 'а', 'я', '.']

Декодированный текст: 'Страница сложная.'

Проверка обратимости 2: Исходный: 'Страница сложная.', Декодированный: 'Страница сложная.'

--- Тестирование OOV-слова ---

--- Кодирование текста в ID ---

Исходный текст: 'Преподавательница читает.'

Закодированные ID: [3, 0, 27, 12, 0, 4, 40, 5, 15, 5, 11, 19, 47, 9, 7, 46, 5, 3, 34, 26, 5, 12, 13, 6]

--- Декодирование ID в текст ---

ID для декодирования: [3, 0, 27, 12, 0, 4, 40, 5, 15, 5, 11, 19, 47, 9, 7, 46, 5, 3, 34, 26, 5, 12, 13, 6]

Декодированный текст: ' ⁇ ре ⁇ одавательница читает.'

Исходный OOV: 'Преподавательница читает.', Закодированные ID: [3, 0, 27, 12, 0, 4, 40, 5, 15, 5, 11, 19, 47, 9, 7, 46, 5, 3, 34, 26, 5, 12, 13, 6], Декодированный: ' ⁇ ре ⁇ одавательница читает.'

--- Очистка временных файлов модели ---

Удален файл модели: 'my\_sp\_model.model'

Удален файл словаря: 'my\_sp\_model.vocab'

### **HuggingFace Tokenizers**

**Введение в Токенизацию и HuggingFace Tokenizers**

Обработка естественного языка (НЛП) – это область искусственного интеллекта, которая занимается взаимодействием компьютеров и человеческого языка. Прежде чем языковые модели, такие как BERT, GPT или T5, смогут понять и обработать текстовые данные, текст должен быть преобразован в числовой формат. Этот процесс называется **токенизацией**.

**Токенизация** – это разбиение непрерывного текста на более мелкие единицы, называемые **токенами**. Токенами могут быть слова, части слов (субслова) или даже отдельные символы. Выбор метода токенизации существенно влияет на производительность языковых моделей.

Библиотека **HuggingFace Tokenizers** – это высокопроизводительная библиотека, написанная на Rust, предназначенная для быстрой и эффективной токенизации текста. Она является основой для большинства токенизаторов, используемых в библиотеке HuggingFace Transformers, и предлагает широкий спектр предварительно обученных токенизаторов, а также инструменты для обучения собственных.

**Почему HuggingFace Tokenizers?**

* **Скорость:** Благодаря реализации на Rust, токенизаторы работают значительно быстрее, чем их аналоги на Python, что критически важно при обработке больших объемов данных.
* **Гибкость:** Поддерживает различные алгоритмы токенизации (BPE, WordPiece, Unigram) и позволяет настраивать каждый этап процесса токенизации.
* **Интеграция:** Бесшовно интегрируется с библиотекой HuggingFace Transformers, позволяя легко загружать и использовать токенизаторы, соответствующие предварительно обученным моделям.
* **Воспроизводимость:** Обеспечивает согласованную токенизацию, что важно для воспроизводимости результатов исследований и развертывания моделей.

**Основные Концепции Токенизации**

Для полного понимания HuggingFace Tokenizers необходимо разобраться с ключевыми концепциями:

**Типы Токенизации**

Исторически токенизация начиналась с простых методов, таких как разбиение по пробелам или пунктуации. Однако современные языковые модели используют более сложные подходы, в основном **субсловарную токенизацию**.

**Пословная Токенизация (Word-level Tokenization)**

Самый простой подход, где каждое слово считается токеном.

Пример: "Привет, мир!" -> ["Привет", ",", "мир", "!"]

Недостатки:

* **Большой словарь:** Огромное количество уникальных слов, включая опечатки и редкие слова.
* **Проблема "вне словаря" (Out-Of-Vocabulary, OOV):** Новые или редкие слова, не встречавшиеся в обучающих данных, не могут быть токенизированы.

**Символьная Токенизация (Character-level Tokenization)**

Каждый символ является токеном.

Пример: "Привет" -> ["П", "р", "и", "в", "е", "т"]

Преимущества:

* Нет проблемы OOV.
* Компактный словарь.

Недостатки:

* Потеря семантической информации (слова разбиваются на несвязанные символы).
* Очень длинные последовательности токенов, что увеличивает вычислительную сложность.

**Субсловарная Токенизация (Subword Tokenization)**

Этот подход является компромиссом между пословной и символьной токенизацией. Он разбивает слова на более мелкие, осмысленные единицы (субслова), которые могут быть частью нескольких слов. Это позволяет обрабатывать OOV-слова, разбивая их на известные субслова, и при этом сохранять некоторую семантику.

Основные алгоритмы субсловарной токенизации:

* **Byte Pair Encoding (BPE):**
* **WordPiece:**
* **Unigram Language Model:**

**Словарь (Vocabulary)**

**Словарь** – это набор всех уникальных токенов (слов, субслов, символов), которые токенизатор знает. Каждому токену в словаре присваивается уникальный числовой идентификатор (ID). Когда текст токенизируется, каждое субслово преобразуется в свой ID.

**Специальные Токены (Special Tokens)**

Языковые модели часто требуют добавления специальных токенов к входным последовательностям для обозначения начала/конца последовательности, разделения сегментов, маскирования и т.д.

* **[CLS] (Classifier):** Токен, добавляемый в начало входной последовательности для задач классификации (например, в BERT). Его конечное скрытое состояние часто используется как агрегированное представление всей последовательности.
* **[SEP] (Separator):** Токен, используемый для разделения двух сегментов текста в одной входной последовательности (например, вопрос и контекст в задачах вопросно-ответных систем). Также может обозначать конец последовательности.
* **[PAD] (Padding):** Токен, используемый для выравнивания длины последовательностей в батче. Более короткие последовательности дополняются [PAD] токенами до максимальной длины в батче.
* **[UNK] (Unknown):** Токен, используемый для замены слов или субслов, которые не встречаются в словаре токенизатора (OOV-слова).
* **[MASK] (Mask):** Токен, используемый в задачах маскированного языкового моделирования (Masked Language Modeling, MLM), где часть токенов в предложении скрывается, и модель должна предсказать их.

**Использование HuggingFace Tokenizers**

Библиотека HuggingFace Tokenizers предоставляет как низкоуровневый API для создания и обучения токенизаторов с нуля, так и высокоуровневый API через AutoTokenizer из библиотеки Transformers для работы с предварительно обученными токенизаторами.

**Загрузка Предварительно Обученного Токенизатора**

Самый распространенный способ использования токенизаторов – это загрузка токенизатора, который был предварительно обучен вместе с конкретной языковой моделью. Это гарантирует, что токенизатор использует тот же словарь и те же правила токенизации, что и модель.

|  |
| --- |
| from transformers import AutoTokenizer  # Загрузка токенизатора для модели BERT base cased  tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")  # Загрузка токенизатора для модели GPT-2  # tokenizer = AutoTokenizer.from\_pretrained("gpt2")  # Загрузка токенизатора для мультиязычной модели XLM-RoBERTa  # tokenizer = AutoTokenizer.from\_pretrained("xlm-roberta-base")  print(f"Тип токенизатора: {type(tokenizer)}")  print(f"Размер словаря: {tokenizer.vocab\_size}")  print(f"Токен CLS: {tokenizer.cls\_token} (ID: {tokenizer.cls\_token\_id})")  print(f"Токен SEP: {tokenizer.sep\_token} (ID: {tokenizer.sep\_token\_id})")  print(f"Токен PAD: {tokenizer.pad\_token} (ID: {tokenizer.pad\_token\_id})")  print(f"Токен UNK: {tokenizer.unk\_token} (ID: {tokenizer.unk\_token\_id})") |

Результат:

Размер словаря: 28996

Токен CLS: [CLS] (ID: 101)

Токен SEP: [SEP] (ID: 102)

Токен PAD: [PAD] (ID: 0)

Токен UNK: [UNK] (ID: 100)

**Базовая Токенизация**

После загрузки токенизатора вы можете использовать его для кодирования текста в последовательности числовых ID.

|  |
| --- |
| text = "Привет, мир! HuggingFace Tokenizers - это круто."  # Токенизация текста  encoded\_input = tokenizer(text)  print(f"Входной текст: {text}")  print(f"Сырые ID токенов: {encoded\_input['input\_ids']}")  print(f"Маска внимания: {encoded\_input['attention\_mask']}")  # Декодирование ID обратно в текст  decoded\_text = tokenizer.decode(encoded\_input['input\_ids'])  print(f"Декодированный текст: {decoded\_text}")  # Получение списка токенов (строковых представлений)  tokens = tokenizer.tokenize(text)  print(f"Список токенов: {tokens}")  # Преобразование токенов в ID  token\_ids = tokenizer.convert\_tokens\_to\_ids(tokens)  print(f"ID токенов из списка: {token\_ids}") |

Результат:

Входной текст: Привет, мир! HuggingFace Tokenizers - это круто.

Сырые ID токенов: [101, 462, 20442, 17424, 28394, 19692, 28404, 117, 487, 17424, 20442, 106, 20164, 10932, 2271, 7954, 1706, 6378, 17260, 1116, 118, 504, 28404, 16948, 485, 20442, 28405, 28404, 16948, 119, 102]

Маска внимания: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Декодированный текст: [CLS] Привет, мир! HuggingFace Tokenizers - это круто. [SEP]

Список токенов: ['П', '##р', '##и', '##в', '##е', '##т', ',', 'м', '##и', '##р', '!', 'Hu', '##gging', '##F', '##ace', 'To', '##ken', '##izer', '##s', '-', 'э', '##т', '##о', 'к', '##р', '##у', '##т', '##о', '.']

ID токенов из списка: [462, 20442, 17424, 28394, 19692, 28404, 117, 487, 17424, 20442, 106, 20164, 10932, 2271, 7954, 1706, 6378, 17260, 1116, 118, 504, 28404, 16948, 485, 20442, 28405, 28404, 16948, 119]

**Нюанс:** Метод tokenizer() (или \_\_call\_\_) является предпочтительным, так как он выполняет все шаги токенизации, включая добавление специальных токенов, паддинг и обрезку, и возвращает словарь с input\_ids, attention\_mask и token\_type\_ids. Метод tokenizer.tokenize() возвращает только список строковых токенов без специальных токенов и других атрибутов.

**Выходные Данные Токенизатора**

Метод tokenizer() возвращает объект BatchEncoding, который ведет себя как словарь и содержит следующие ключи:

* **input\_ids:** Список числовых ID токенов. Это основное представление текста для модели.
* **attention\_mask:** Бинарная маска, указывающая, какие токены являются реальными (1) и какие – паддингом (0). Модели используют эту маску, чтобы игнорировать токены паддинга при вычислении внимания.
* **token\_type\_ids (или segment\_ids):** Используется для задач, включающих два сегмента текста (например, вопрос и ответ). Обычно 0 для первого сегмента и 1 для второго. Если сегмент один, все значения будут 0.

|  |
| --- |
| # Пример с двумя сегментами (для моделей типа BERT)  sentence1 = "Кто написал 'Войну и мир'?"  sentence2 = "Лев Толстой."  encoded\_pair = tokenizer(sentence1, sentence2)  print(f"\nТокенизация двух предложений:")  print(f"Input IDs: {encoded\_pair['input\_ids']}")  print(f"Attention Mask: {encoded\_pair['attention\_mask']}")  print(f"Token Type IDs: {encoded\_pair['token\_type\_ids']}")  # Декодирование  print(f"Декодированный текст: {tokenizer.decode(encoded\_pair['input\_ids'])}") |

Результат:

Токенизация двух предложений:

Input IDs: [101, 457, 28404, 16948, 488, 10286, 28402, 17424, 28403, 10286, 28400, 112, 450, 16948, 17106, 17127, 28405, 483, 487, 17424, 20442, 112, 136, 102, 458, 19692, 28394, 465, 16948, 28400, 28403, 28404, 16948, 17106, 119, 102]

Attention Mask: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Token Type IDs: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Декодированный текст: [CLS] Кто написал ' Войну и мир '? [SEP] Лев Толстой. [SEP]

**Процесс Токенизации (Детально)**

Токенизаторы HuggingFace Tokenizers следуют многоступенчатому конвейеру:

* **Нормализация (Normalization):** Преобразование текста в стандартную форму. Это может включать:
* Приведение к нижнему регистру.
* Удаление диакритических знаков.
* Замена символов (например, различных типов кавычек на стандартные).
* Приведение к Unicode Normalization Form (NFC, NFD и т.д.).

**Пример:** "Héllö" -> "Hello"

* **Предварительная токенизация (Pre-tokenization):** Разбиение текста на "черновые" токены, обычно по пробелам и пунктуации. На этом этапе еще не используются субслова.

**Пример:** "Привет, мир!" -> ["Привет", ",", "мир", "!"]

* **Модель (Model):** Применение основного алгоритма субсловарной токенизации (BPE, WordPiece, Unigram) к предварительно токенизированным частям. Это этап, где слова разбиваются на субслова.

**Пример:** "HuggingFace" -> ["Hug", "##ging", "##Face"] (WordPiece)

* **Пост-обработка (Post-processing):** Добавление специальных токенов ([CLS], [SEP], [PAD]), генерация attention\_mask и token\_type\_ids. Этот этап также обрабатывает обрезку (truncation) и паддинг (padding).

**Практические Примеры и Нюансы**

**Паддинг и Обрезка (Padding and Truncation)**

Языковые модели работают с последовательностями фиксированной длины. Поэтому необходимо выравнивать (паддинг) или обрезать (truncation) входные последовательности.

* **padding=True (или 'longest'):** Дополняет последовательности до самой длинной последовательности в текущем батче.
* **padding='max\_length':** Дополняет последовательности до указанной max\_length.
* **truncation=True (или 'longest\_first', 'only\_first', 'only\_second'):** Обрезает последовательности, если они превышают max\_length.
* **max\_length:** Максимальная длина последовательности.

|  |
| --- |
| sentences = [      "Это короткое предложение.",      "Это очень, очень, очень, очень, очень, очень, очень длинное предложение, которое, вероятно, будет обрезано."  ]  # Без паддинга и обрезки  encoded\_no\_pad\_trunc = tokenizer(sentences)  print(f"\nБез паддинга и обрезки: {encoded\_no\_pad\_trunc['input\_ids']}")  # С паддингом до самой длинной в батче  encoded\_padded = tokenizer(sentences, padding=True)  print(f"\nС паддингом (longest): {encoded\_padded['input\_ids']}")  print(f"Attention Mask (padded): {encoded\_padded['attention\_mask']}")  # С паддингом до max\_length и обрезкой  encoded\_padded\_truncated = tokenizer(sentences, padding='max\_length', max\_length=10, truncation=True)  print(f"\nС паддингом и обрезкой (max\_length=10): {encoded\_padded\_truncated['input\_ids']}")  print(f"Attention Mask (padded and truncated): {encoded\_padded\_truncated['attention\_mask']}")  # Декодирование обрезанного предложения  print(f"Декодированное обрезанное предложение: {tokenizer.decode(encoded\_padded\_truncated['input\_ids'][1])}") |

Результат:

Без паддинга и обрезки: [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 28399, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 119, 102], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 479, 28400, 17424, 17127, 17127, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 117, 485, 16948, 28404, 16948, 20442, 16948, 19692, 117, 477, 19692, 20442, 16948, 14800, 28404, 17127, 16948, 117, 476, 28405, 28396, 19692, 28404, 489, 28393, 20442, 19692, 28398, 10286, 17127, 16948, 119, 102]]

С паддингом (longest): [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 28399, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 119, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 479, 28400, 17424, 17127, 17127, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 117, 485, 16948, 28404, 16948, 20442, 16948, 19692, 117, 477, 19692, 20442, 16948, 14800, 28404, 17127, 16948, 117, 476, 28405, 28396, 19692, 28404, 489, 28393, 20442, 19692, 28398, 10286, 17127, 16948, 119, 102]]

Attention Mask (padded): [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

С паддингом и обрезкой (max\_length=10): [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 102], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 102]]

Attention Mask (padded and truncated): [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

Декодированное обрезанное предложение: [CLS] Это очень [SEP]

**Возвращаемые Типы (Return Types)**

Токенизатор может возвращать данные в различных форматах, удобных для разных фреймворков.

* **return\_tensors='pt':** Возвращает PyTorch тензоры.
* **return\_tensors='tf':** Возвращает TensorFlow тензоры.
* **return\_tensors='np':** Возвращает NumPy массивы.

|  |
| --- |
| import torch # или tensorflow as tf, или numpy as np  text = "Пример текста."  # Возвращает PyTorch тензоры  encoded\_pt = tokenizer(text, return\_tensors='pt')  print(f"\nPyTorch тензоры: {encoded\_pt['input\_ids'].shape}, тип: {encoded\_pt['input\_ids'].dtype}")  # Возвращает NumPy массивы  encoded\_np = tokenizer(text, return\_tensors='np')  print(f"NumPy массивы: {encoded\_np['input\_ids'].shape}, тип: {encoded\_np['input\_ids'].dtype}") |

Результат:

PyTorch тензоры: torch.Size([1, 15]), тип: torch.int64

NumPy массивы: (1, 15), тип: int64

**Обработка Батчей (Batch Processing)**

HuggingFace Tokenizers оптимизированы для обработки батчей (списков текстов), что значительно ускоряет токенизацию.

|  |
| --- |
| batch\_texts = [      "Первое предложение в батче.",      "Второе, более длинное предложение, для демонстрации паддинга."  ]  encoded\_batch = tokenizer(batch\_texts, padding=True, truncation=True, max\_length=20, return\_tensors='pt')  print(f"\nОбработка батчей:")  print(f"Input IDs (батч): {encoded\_batch['input\_ids']}")  print(f"Attention Mask (батч): {encoded\_batch['attention\_mask']}") |

Результат:

Обработка батчей:

Input IDs (батч): tensor([[ 101, 462, 19692, 20442, 28394, 16948, 19692, 490, 20442, 19692,

28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 477, 102],

[ 101, 450, 28404, 16948, 20442, 16948, 19692, 117, 476, 16948,

28400, 19692, 19692, 479, 28400, 17424, 17127, 17127, 16948, 102]])

Attention Mask (батч): tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

**Обучение Пользовательских Токенизаторов**

Хотя предварительно обученные токенизаторы подходят для большинства задач, иногда необходимо обучить собственный токенизатор:

* **Для новых языков:** Если для вашего языка нет подходящих предварительно обученных токенизаторов.
* **Для специфических доменов:** Если ваш текст содержит много домен-специфических терминов, которые плохо токенизируются существующими токенизаторами (например, медицинские тексты, юридические документы).
* **Для уменьшения размера словаря:** Если вы хотите более компактный словарь, специфичный для вашего корпуса.

HuggingFace Tokenizers позволяет обучать BPE, WordPiece и Unigram токенизаторы.

**Пример Обучения BPE Токенизатора**

Для обучения токенизатора вам понадобится большой корпус текста.

from tokenizers import BytePairEncoding, models, pre\_tokenizers, trainers, decoders  
from tokenizers import Tokenizer as HFTokenizer # Избегаем конфликта имен с transformers.AutoTokenizer

|  |
| --- |
| from tokenizers import Tokenizer as HFTokenizer  from tokenizers import models, pre\_tokenizers, decoders, trainers  from transformers import PreTrainedTokenizerFast  # 1. Инициализация пустого токенизатора с моделью BPE  tokenizer = HFTokenizer(models.BPE())  # 2. Установка предварительного токенизатора (разделение по пробелам)  tokenizer.pre\_tokenizer = pre\_tokenizers.Whitespace()  # 3. Создание тренера для обучения BPE  trainer = trainers.BpeTrainer(  vocab\_size=1000,  min\_frequency=2,  special\_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"]  )  # 4. Подготовка корпуса (в реальной задаче — путь к файлам или итератор строк)  corpus = [  "HuggingFace Tokenizers — это мощная библиотека для токенизации текста.",  "Она очень быстрая и эффективная, написана на Rust.",  "Обучение токенизатора позволяет адаптировать его под специфические данные.",  "НЛП-задачи требуют качественной токенизации.",  "Пример текста для обучения токенизатора."  ]  # 5. Обучение токенизатора на корпусе  tokenizer.train\_from\_iterator(corpus, trainer=trainer)  # 6. Установка декодера (декодирование ID обратно в текст)  tokenizer.decoder = decoders.ByteLevel()  # 7. Сохранение токенизатора в файл  tokenizer.save("my\_custom\_bpe\_tokenizer.json")  print("✅ Пользовательский токенизатор обучен и сохранен.")  # 8. Загрузка токенизатора через PreTrainedTokenizerFast  custom\_tokenizer = PreTrainedTokenizerFast(tokenizer\_file="my\_custom\_bpe\_tokenizer.json")  # 9. Установка специальных токенов (если они не определены автоматически)  custom\_tokenizer.add\_special\_tokens({  "unk\_token": "[UNK]",  "cls\_token": "[CLS]",  "sep\_token": "[SEP]",  "pad\_token": "[PAD]",  "mask\_token": "[MASK]"  })  # 10. Тестирование токенизатора на новом предложении  text = "Тестовое предложение для нового токенизатора."  encoded = custom\_tokenizer(text, padding=True, truncation=True, return\_tensors='pt')  print("\n🔎 Результаты токенизации:")  print(f"Input IDs: {encoded['input\_ids']}")  print(f"Attention Mask: {encoded['attention\_mask']}")  print(f"Декодированный текст: {custom\_tokenizer.decode(encoded['input\_ids'][0], skip\_special\_tokens=True)}") |

Результат:

Asking to truncate to max\_length but no maximum length is provided and the model has no predefined maximum length. Default to no truncation.

✅ Пользовательский токенизатор обучен и сохранен.

🔎 Результаты токенизации:

Input IDs: tensor([[34, 45, 59, 31, 42, 34, 43, 44, 34, 33, 39, 42, 61, 34, 74, 41, 42, 31,

42, 32, 42, 83, 7]])

Attention Mask: tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

Декодированный текст: естовоепредлоениедляновоготокенизатора.

**Нюанс:** При обучении токенизатора важно использовать достаточно большой и репрезентативный корпус данных, чтобы словарь был полным и эффективным для ваших задач.

**Продвинутые Темы и Нюансы**

**"Быстрые" и "Медленные" Токенизаторы**

Библиотека Transformers предоставляет два типа токенизаторов:

* **"Быстрые" токенизаторы (Fast Tokenizers):** Реализованы на Rust с использованием библиотеки HuggingFace Tokenizers. Они значительно быстрее и поддерживают все продвинутые функции (батчевая обработка, параллелизм, return\_tensors). Их имена обычно заканчиваются на Fast (например, BertTokenizerFast).
* **"Медленные" токенизаторы (Slow Tokenizers):** Полностью реализованы на Python. Они медленнее, но иногда могут быть полезны для отладки или для моделей, для которых еще нет "быстрой" реализации. Их имена не имеют суффикса Fast (например, BertTokenizer).

AutoTokenizer.from\_pretrained() по умолчанию загружает "быструю" версию, если она доступна. Вы можете явно указать use\_fast=False для загрузки "медленной" версии.

**Параллелизм**

HuggingFace Tokenizers по умолчанию использует параллелизм (многопоточность) для ускорения токенизации, что особенно заметно при обработке больших батчей.

**Сериализация (JSON)**

Обученные токенизаторы могут быть сохранены и загружены в формате JSON. Это позволяет легко делиться токенизаторами и использовать их в различных окружениях. Файл .json содержит всю конфигурацию токенизатора, включая словарь, правила нормализации, предварительной токенизации и пост-обработки.

**Управление Словарем**

Вы можете программно получить доступ к словарю токенизатора, добавлять новые токены или изменять их.

|  |
| --- |
| from transformers import AutoTokenizer  # 1. Загрузка предобученного токенизатора (например, BERT base cased)  tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")  # 2. Получение словаря токенов  vocab = tokenizer.vocab  print(f"\n🔢 Размер словаря: {len(vocab)}")  # 3. Получение ID для конкретного токена  token\_id = tokenizer.convert\_tokens\_to\_ids("hello")  print(f"🔍 ID для токена 'hello': {token\_id}")  # 4. Получение токена по его ID  token\_str = tokenizer.convert\_ids\_to\_tokens(100)  # Обычно это [UNK] для BERT base cased  print(f"🔁 Токен для ID 100: {token\_str}")  # 5. Добавление новых пользовательских токенов  num\_added\_tokens = tokenizer.add\_tokens(["[NEW\_TOKEN\_1]", "специальный\_термин"])  print(f"\n➕ Добавлено токенов: {num\_added\_tokens}")  print(f"📈 Новый размер словаря: {len(tokenizer)}")  # 6. Получение ID для нового токена  new\_token\_id = tokenizer.convert\_tokens\_to\_ids("[NEW\_TOKEN\_1]")  print(f"🆔 ID для '[NEW\_TOKEN\_1]': {new\_token\_id}") |

Результат:

🔢 Размер словаря: 28996

🔍 ID для токена 'hello': 19082

🔁 Токен для ID 100: [UNK]

➕ Добавлено токенов: 2

📈 Новый размер словаря: 28998

🆔 ID для '[NEW\_TOKEN\_1]': 28996

**Важно:** При добавлении новых токенов в токенизатор, который используется с предварительно обученной моделью, необходимо также изменить размер embedding-слоя модели, чтобы он соответствовал новому размеру словаря.

|  |
| --- |
| from transformers import AutoTokenizer, AutoModelForMaskedLM  # 1. Загрузка токенизатора и модели  tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")  model = AutoModelForMaskedLM.from\_pretrained("bert-base-cased")  # 2. Проверка исходного размера embedding-слоя (через model.bert.embeddings)  original\_vocab\_size = model.bert.embeddings.word\_embeddings.weight.shape[0]  print(f"🔢 Исходный размер словаря (эмбеддингов): {original\_vocab\_size}")  # 3. Добавление новых токенов  new\_tokens = ["[NEW\_TOKEN\_1]", "специальный\_термин"]  num\_added = tokenizer.add\_tokens(new\_tokens)  print(f"➕ Добавлено токенов: {num\_added}")  # 4. Обновление embedding-слоя модели  model.resize\_token\_embeddings(len(tokenizer))  # 5. Проверка нового размера embedding-слоя  new\_vocab\_size = model.bert.embeddings.word\_embeddings.weight.shape[0]  print(f"📈 Новый размер словаря (эмбеддингов): {new\_vocab\_size}")  # 6. Проверка ID добавленного токена  print(f"🆔 ID для '[NEW\_TOKEN\_1]': {tokenizer.convert\_tokens\_to\_ids('[NEW\_TOKEN\_1]')}") |

Результат:

🔢 Исходный размер словаря (эмбеддингов): 28996

➕ Добавлено токенов: 2

The new embeddings will be initialized from a multivariate normal distribution that has old embeddings' mean and covariance. As described in this article: <https://nlp.stanford.edu/~johnhew/vocab-expansion.html>. To disable this, use `mean\_resizing=False`

📈 Новый размер словаря (эмбеддингов): 28998

🆔 ID для '[NEW\_TOKEN\_1]': 28996

**Обработка Различных Языков**

HuggingFace Tokenizers поддерживает множество языков. При выборе токенизатора важно учитывать язык вашего корпуса. Для русского языка существуют специализированные модели и токенизаторы, например, DeepPavlov/rubert-base-cased или cointegrated/rubert-tiny2.

|  |
| --- |
| from transformers import AutoTokenizer  # 1. Загрузка русскоязычного токенизатора RuBERT  ru\_tokenizer = AutoTokenizer.from\_pretrained("DeepPavlov/rubert-base-cased")  # 2. Пример текста на русском языке  ru\_text = "Пример текста на русском языке."  # 3. Токенизация текста  encoded\_ru = ru\_tokenizer(ru\_text, return\_tensors='pt')  # 4. Вывод результатов  print("\n📘 Русский текст:")  print(f"📝 Оригинал: {ru\_text}")  print(f"🔢 Input IDs: {encoded\_ru['input\_ids']}")  print(f"🔁 Декодированный текст: {ru\_tokenizer.decode(encoded\_ru['input\_ids'][0], skip\_special\_tokens=True)}") |

Результат:

📘 Русский текст:

📝 Оригинал: Пример текста на русском языке.

🔢 Input IDs: tensor([[ 101, 22281, 22574, 1469, 18250, 11956, 132, 102]])

🔁 Декодированный текст: Пример текста на русском языке.

Польный код на питон:

|  |
| --- |
| import torch  import numpy as np  from transformers import AutoTokenizer, AutoModelForMaskedLM, PreTrainedTokenizerFast  from tokenizers import models, pre\_tokenizers, trainers, decoders # Изменено: BytePairEncoding удален из прямого импорта  from tokenizers import Tokenizer as HFCoreTokenizer # Переименовываем, чтобы избежать конфликта с AutoTokenizer  class HuggingFaceTokenizerDemo:      """      Класс для демонстрации различных методов токенизации с использованием HuggingFace Tokenizers      и возможностей обучения пользовательских токенизаторов.      """      def \_\_init\_\_(self):          """          Инициализирует класс, загружая предварительно обученный токенизатор BERT.          """          print("Инициализация HuggingFaceTokenizerDemo...")          # Загружаем предварительно обученный токенизатор для BERT base cased.          # Это "быстрый" токенизатор, реализованный на Rust.          self.tokenizer = AutoTokenizer.from\_pretrained("bert-base-cased")          print(f"Загружен токенизатор: {type(self.tokenizer)}")          print(f"Размер словаря токенизатора: {self.tokenizer.vocab\_size}")          print("-" \* 50)      def demonstrate\_basic\_tokenization(self, text: str):          """          Демонстрирует базовую токенизацию текста.          Включает кодирование, декодирование и получение отдельных токенов.          """          print(f"\n--- Демонстрация базовой токенизации ---")          print(f"Исходный текст: '{text}'")          # Кодирование текста: преобразует текст в числовые ID токенов,          # добавляет специальные токены ([CLS], [SEP]), генерирует маску внимания.          encoded\_input = self.tokenizer(text)          print(f"Сырые ID токенов: {encoded\_input['input\_ids']}")          print(f"Маска внимания (attention\_mask): {encoded\_input['attention\_mask']}")          # token\_type\_ids будут все нули, так как у нас одно предложение          print(f"Типы токенов (token\_type\_ids): {encoded\_input['token\_type\_ids']}")          # Декодирование ID обратно в текст: преобразует числовые ID токенов          # обратно в читаемый текст, удаляя специальные токены.          decoded\_text = self.tokenizer.decode(encoded\_input['input\_ids'])          print(f"Декодированный текст: '{decoded\_text}'")          # Получение списка строковых токенов без специальных токенов.          tokens = self.tokenizer.tokenize(text)          print(f"Список строковых токенов: {tokens}")          # Преобразование списка строковых токенов в их числовые ID.          token\_ids = self.tokenizer.convert\_tokens\_to\_ids(tokens)          print(f"ID токенов из списка: {token\_ids}")          print("-" \* 50)      def demonstrate\_multi\_segment\_tokenization(self, sentence1: str, sentence2: str):          """          Демонстрирует токенизацию двух сегментов текста, как это делается для задач          вопросно-ответных систем или определения логического следования (NLI).          """          print(f"\n--- Демонстрация токенизации двух сегментов ---")          print(f"Предложение 1: '{sentence1}'")          print(f"Предложение 2: '{sentence2}'")          # Токенизация двух предложений. Токенизатор автоматически добавит [CLS] и [SEP]          # и сгенерирует token\_type\_ids для различения сегментов.          encoded\_pair = self.tokenizer(sentence1, sentence2)          print(f"Input IDs: {encoded\_pair['input\_ids']}")          print(f"Attention Mask: {encoded\_pair['attention\_mask']}")          # token\_type\_ids: 0 для первого предложения и [CLS]/[SEP], 1 для второго предложения.          print(f"Token Type IDs: {encoded\_pair['token\_type\_ids']}")          # Декодирование объединенной последовательности.          print(f"Декодированный текст: '{self.tokenizer.decode(encoded\_pair['input\_ids'])}'")          print("-" \* 50)      def demonstrate\_padding\_and\_truncation(self, sentences: list[str], max\_length: int = 15):          """          Демонстрирует использование паддинга (padding) и обрезки (truncation)          для выравнивания последовательностей по длине.          """          print(f"\n--- Демонстрация паддинга и обрезки ---")          print(f"Исходные предложения: {sentences}")          print(f"Максимальная длина (max\_length): {max\_length}")          # Токенизация без паддинга и обрезки.          # Каждая последовательность будет иметь свою естественную длину.          encoded\_no\_pad\_trunc = self.tokenizer(sentences)          print(f"\nБез паддинга и обрезки (ID): {encoded\_no\_pad\_trunc['input\_ids']}")          # Токенизация с паддингом до самой длинной последовательности в батче.          # `padding=True` эквивалентно `padding='longest'`.          encoded\_padded = self.tokenizer(sentences, padding=True)          print(f"\nС паддингом (до самой длинной в батче):")          print(f"  Input IDs: {encoded\_padded['input\_ids']}")          print(f"  Attention Mask: {encoded\_padded['attention\_mask']}")          # Токенизация с паддингом до `max\_length` и обрезкой до `max\_length`.          encoded\_padded\_truncated = self.tokenizer(              sentences, padding='max\_length', max\_length=max\_length, truncation=True          )          print(f"\nС паддингом до {max\_length} и обрезкой:")          print(f"  Input IDs: {encoded\_padded\_truncated['input\_ids']}")          print(f"  Attention Mask: {encoded\_padded\_truncated['attention\_mask']}")          # Декодирование обрезанного предложения для наглядности.          print(f"  Декодированное обрезанное предложение (второе): '{self.tokenizer.decode(encoded\_padded\_truncated['input\_ids'][1])}'")          print("-" \* 50)      def demonstrate\_return\_tensors(self, text: str):          """          Демонстрирует возвращение токенизированных данных в различных форматах тензоров.          """          print(f"\n--- Демонстрация возвращаемых типов тензоров ---")          print(f"Текст: '{text}'")          # Возвращает PyTorch тензоры.          encoded\_pt = self.tokenizer(text, return\_tensors='pt')          print(f"\nPyTorch тензоры:")          print(f"  Тип: {type(encoded\_pt['input\_ids'])}")          print(f"  Форма: {encoded\_pt['input\_ids'].shape}")          print(f"  Значения: {encoded\_pt['input\_ids']}")          # Возвращает TensorFlow тензоры (требуется установленный TensorFlow).          try:              encoded\_tf = self.tokenizer(text, return\_tensors='tf')              print(f"\nTensorFlow тензоры:")              print(f"  Тип: {type(encoded\_tf['input\_ids'])}")              print(f"  Форма: {encoded\_tf['input\_ids'].shape}")              print(f"  Значения: {encoded\_tf['input\_ids']}")          except Exception as e:              print(f"\nНе удалось получить TensorFlow тензоры (возможно, TensorFlow не установлен): {e}")          # Возвращает NumPy массивы.          encoded\_np = self.tokenizer(text, return\_tensors='np')          print(f"\nNumPy массивы:")          print(f"  Тип: {type(encoded\_np['input\_ids'])}")          print(f"  Форма: {encoded\_np['input\_ids'].shape}")          print(f"  Значения: {encoded\_np['input\_ids']}")          print("-" \* 50)      def demonstrate\_batch\_processing(self, batch\_texts: list[str], max\_length: int = 20):          """          Демонстрирует эффективную обработку батчей текстов.          """          print(f"\n--- Демонстрация пакетной обработки ---")          print(f"Пакет текстов: {batch\_texts}")          print(f"Максимальная длина: {max\_length}")          # Токенизация всего батча. Паддинг будет выполнен до самой длинной последовательности          # в батче или до max\_length, если указано.          encoded\_batch = self.tokenizer(              batch\_texts,              padding=True,              truncation=True,              max\_length=max\_length,              return\_tensors='pt'          )          print(f"\nInput IDs (батч):")          print(encoded\_batch['input\_ids'])          print(f"Attention Mask (батч):")          print(encoded\_batch['attention\_mask'])          print("-" \* 50)      def demonstrate\_vocabulary\_management(self, new\_tokens: list[str]):          """          Демонстрирует управление словарем токенизатора: получение информации,          добавление новых токенов и изменение размера embedding-слоя модели.          """          print(f"\n--- Демонстрация управления словарем ---")          print(f"Исходный размер словаря: {len(self.tokenizer)}")          # Получение ID для существующего токена.          hello\_id = self.tokenizer.convert\_tokens\_to\_ids("hello")          print(f"ID для токена 'hello': {hello\_id}")          # Получение токена по ID.          token\_for\_id\_100 = self.tokenizer.convert\_ids\_to\_tokens(100)          print(f"Токен для ID 100: '{token\_for\_id\_100}' (обычно [UNK] для BERT)")          print(f"\nПопытка добавить новые токены: {new\_tokens}")          # Добавление новых токенов в словарь токенизатора.          num\_added\_tokens = self.tokenizer.add\_tokens(new\_tokens)          print(f"Добавлено токенов: {num\_added\_tokens}")          print(f"Новый размер словаря токенизатора: {len(self.tokenizer)}")          # Показываем ID для новых токенов.          for token in new\_tokens:              print(f"ID для '{token}': {self.tokenizer.convert\_tokens\_to\_ids(token)}")          # Важно: Если вы добавляете токены, которые не были в исходном словаре модели,          # необходимо изменить размер слоя эмбеддингов модели, чтобы он соответствовал          # новому размеру словаря токенизатора.          print(f"\nДемонстрация изменения размера embedding-слоя модели:")          try:              # Загружаем модель (например, для маскированного языкового моделирования)              model = AutoModelForMaskedLM.from\_pretrained("bert-base-cased")              print(f"Исходный размер embedding-слоя модели: {model.embeddings.word\_embeddings.weight.shape[0]}")              # Изменяем размер слоя эмбеддингов.              model.resize\_token\_embeddings(len(self.tokenizer))              print(f"Новый размер embedding-слоя модели после добавления токенов: {model.embeddings.word\_embeddings.weight.shape[0]}")          except Exception as e:              print(f"Не удалось изменить размер embedding-слоя модели: {e}")          print("-" \* 50)      def train\_custom\_bpe\_tokenizer(self, corpus: list[str], vocab\_size: int = 1000, min\_frequency: int = 2, save\_path: str = "my\_custom\_bpe\_tokenizer.json"):          """          Демонстрирует обучение пользовательского BPE токенизатора на новом корпусе.          """          print(f"\n--- Обучение пользовательского BPE токенизатора ---")          print(f"Корпус для обучения (первые 3 строки): {corpus[:3]}...")          print(f"Желаемый размер словаря: {vocab\_size}")          print(f"Минимальная частота токена: {min\_frequency}")          print(f"Путь сохранения: {save\_path}")          # 1. Инициализация пустого токенизатора с моделью BPE.          tokenizer = HFCoreTokenizer(models.BPE())          # 2. Определение предварительного токенизатора.          # Whitespace: разбивает текст по пробелам.          tokenizer.pre\_tokenizer = pre\_tokenizers.Whitespace()          # 3. Создание тренера для BPE.          # special\_tokens: список специальных токенов, которые будут добавлены в словарь          # и не будут участвовать в процессе слияния.          trainer = trainers.BpeTrainer(              vocab\_size=vocab\_size,              min\_frequency=min\_frequency,              special\_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"]          )          # 4. Обучение токенизатора на предоставленном корпусе.          # train\_from\_iterator принимает итератор по строкам текста.          print("Начинается обучение токенизатора...")          tokenizer.train\_from\_iterator(corpus, trainer=trainer)          print("Обучение токенизатора завершено.")          # 5. Установка декодера. Декодер отвечает за преобразование ID обратно в текст.          # ByteLevel декодер подходит для BPE токенизаторов.          tokenizer.decoder = decoders.ByteLevel()          # 6. Сохранение обученного токенизатора в JSON файл.          tokenizer.save(save\_path)          print(f"Пользовательский токенизатор сохранен в '{save\_path}'")          # 7. Загрузка и тестирование пользовательского токенизатора с помощью PreTrainedTokenizerFast.          print(f"\nЗагрузка и тестирование пользовательского токенизатора...")          try:              custom\_tokenizer = PreTrainedTokenizerFast(tokenizer\_file=save\_path)              # Убедитесь, что специальные токены установлены правильно,              # так как PreTrainedTokenizerFast может не всегда автоматически их определять              # из файла JSON, особенно если они не были явно указаны в конфигурации модели.              custom\_tokenizer.unk\_token = "[UNK]"              custom\_tokenizer.cls\_token = "[CLS]"              custom\_tokenizer.sep\_token = "[SEP]"              custom\_tokenizer.pad\_token = "[PAD]"              custom\_tokenizer.mask\_token = "[MASK]"              test\_text = "Это тестовое предложение для проверки нового токенизатора."              encoded\_custom = custom\_tokenizer(test\_text, padding=True, truncation=True, return\_tensors='pt')              print(f"Тестовый текст: '{test\_text}'")              print(f"Input IDs (пользовательский токенизатор): {encoded\_custom['input\_ids']}")              print(f"Attention Mask (пользовательский токенизатор): {encoded\_custom['attention\_mask']}")              print(f"Декодированный текст (пользовательский токенизатор): '{custom\_tokenizer.decode(encoded\_custom['input\_ids'][0])}'")          except Exception as e:              print(f"Ошибка при загрузке или тестировании пользовательского токенизатора: {e}")          print("-" \* 50)  # --- Пример использования класса ---  if \_\_name\_\_ == "\_\_main\_\_":      demo = HuggingFaceTokenizerDemo()      # Демонстрация базовой токенизации      demo.demonstrate\_basic\_tokenization("Привет, мир! HuggingFace Tokenizers - это круто.")      # Демонстрация токенизации двух сегментов      demo.demonstrate\_multi\_segment\_tokenization(          "Кто изобрел лампочку?",          "Томас Эдисон изобрел лампочку."      )      # Демонстрация паддинга и обрезки      demo.demonstrate\_padding\_and\_truncation(          [              "Это короткое предложение.",              "Это очень, очень, очень, очень, очень, очень, очень длинное предложение, которое, вероятно, будет обрезано."          ],          max\_length=15      )      # Демонстрация возвращаемых типов тензоров      demo.demonstrate\_return\_tensors("Пример текста для демонстрации тензоров.")      # Демонстрация пакетной обработки      demo.demonstrate\_batch\_processing(          [              "Первое предложение в батче.",              "Второе, более длинное предложение, для демонстрации паддинга."          ],          max\_length=20      )      # Демонстрация управления словарем      demo.demonstrate\_vocabulary\_management(["[МОЙ\_НОВЫЙ\_ТОКЕН]", "машинное\_обучение"])      # Демонстрация обучения пользовательского токенизатора      # В реальном сценарии корпус будет намного больше.      custom\_corpus = [          "Машинное обучение - это область искусственного интеллекта.",          "Нейронные сети являются ключевым компонентом глубокого обучения.",          "Обработка естественного языка использует токенизацию.",          "HuggingFace предоставляет отличные инструменты для НЛП.",          "Мы изучаем новые алгоритмы токенизации.",          "Это пример текста для обучения токенизатора.",          "НЛП - это круто и очень интересно."      ]      demo.train\_custom\_bpe\_tokenizer(custom\_corpus, vocab\_size=200, min\_frequency=1) |

Результат:

Инициализация HuggingFaceTokenizerDemo...

Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight', 'cls.seq\_relationship.bias', 'cls.seq\_relationship.weight']

- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).

- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).

Asking to truncate to max\_length but no maximum length is provided and the model has no predefined maximum length. Default to no truncation.

Загружен токенизатор: <class 'transformers.models.bert.tokenization\_bert\_fast.BertTokenizerFast'>

Размер словаря токенизатора: 28996

--------------------------------------------------

--- Демонстрация базовой токенизации ---

Исходный текст: 'Привет, мир! HuggingFace Tokenizers - это круто.'

Сырые ID токенов: [101, 462, 20442, 17424, 28394, 19692, 28404, 117, 487, 17424, 20442, 106, 20164, 10932, 2271, 7954, 1706, 6378, 17260, 1116, 118, 504, 28404, 16948, 485, 20442, 28405, 28404, 16948, 119, 102]

Маска внимания (attention\_mask): [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Типы токенов (token\_type\_ids): [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Декодированный текст: '[CLS] Привет, мир! HuggingFace Tokenizers - это круто. [SEP]'

Список строковых токенов: ['П', '##р', '##и', '##в', '##е', '##т', ',', 'м', '##и', '##р', '!', 'Hu', '##gging', '##F', '##ace', 'To', '##ken', '##izer', '##s', '-', 'э', '##т', '##о', 'к', '##р', '##у', '##т', '##о', '.']

ID токенов из списка: [462, 20442, 17424, 28394, 19692, 28404, 117, 487, 17424, 20442, 106, 20164, 10932, 2271, 7954, 1706, 6378, 17260, 1116, 118, 504, 28404, 16948, 485, 20442, 28405, 28404, 16948, 119]

--------------------------------------------------

--- Демонстрация токенизации двух сегментов ---

Предложение 1: 'Кто изобрел лампочку?'

Предложение 2: 'Томас Эдисон изобрел лампочку.'

Input IDs: [101, 457, 28404, 16948, 483, 28398, 16948, 28393, 20442, 19692, 28400, 486, 10286, 28401, 28402, 16948, 28409, 28399, 28405, 136, 102, 465, 16948, 28401, 10286, 28403, 472, 28396, 17424, 28403, 16948, 17127, 483, 28398, 16948, 28393, 20442, 19692, 28400, 486, 10286, 28401, 28402, 16948, 28409, 28399, 28405, 119, 102]

Attention Mask: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Token Type IDs: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Декодированный текст: '[CLS] Кто изобрел лампочку? [SEP] Томас Эдисон изобрел лампочку. [SEP]'

--------------------------------------------------

--- Демонстрация паддинга и обрезки ---

Исходные предложения: ['Это короткое предложение.', 'Это очень, очень, очень, очень, очень, очень, очень длинное предложение, которое, вероятно, будет обрезано.']

Максимальная длина (max\_length): 15

Без паддинга и обрезки (ID): [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 28399, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 119, 102], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 479, 28400, 17424, 17127, 17127, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 117, 485, 16948, 28404, 16948, 20442, 16948, 19692, 117, 477, 19692, 20442, 16948, 14800, 28404, 17127, 16948, 117, 476, 28405, 28396, 19692, 28404, 489, 28393, 20442, 19692, 28398, 10286, 17127, 16948, 119, 102]]

С паддингом (до самой длинной в батче):

Input IDs: [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 28399, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 119, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 28414, 479, 28400, 17424, 17127, 17127, 16948, 19692, 490, 20442, 19692, 28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 117, 485, 16948, 28404, 16948, 20442, 16948, 19692, 117, 477, 19692, 20442, 16948, 14800, 28404, 17127, 16948, 117, 476, 28405, 28396, 19692, 28404, 489, 28393, 20442, 19692, 28398, 10286, 17127, 16948, 119, 102]]

Attention Mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

С паддингом до 15 и обрезкой:

Input IDs: [[101, 472, 28404, 16948, 485, 16948, 20442, 16948, 28404, 28399, 16948, 19692, 490, 20442, 102], [101, 472, 28404, 16948, 489, 28409, 19692, 17127, 28414, 117, 489, 28409, 19692, 17127, 102]]

Attention Mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

Декодированное обрезанное предложение (второе): '[CLS] Это очень, очен [SEP]'

--------------------------------------------------

--- Демонстрация возвращаемых типов тензоров ---

Текст: 'Пример текста для демонстрации тензоров.'

PyTorch тензоры:

Тип: <class 'torch.Tensor'>

Форма: torch.Size([1, 37])

Значения: tensor([[ 101, 462, 20442, 17424, 28401, 19692, 20442, 493, 19692, 28399,

28403, 28404, 10286, 479, 28400, 14800, 479, 19692, 28401, 16948,

17127, 28403, 28404, 20442, 10286, 28408, 17424, 17424, 493, 19692,

17127, 28398, 16948, 20442, 24625, 119, 102]])

TensorFlow тензоры:

Тип: <class 'tensorflow.python.framework.ops.EagerTensor'>

Форма: (1, 37)

Значения: [[ 101 462 20442 17424 28401 19692 20442 493 19692 28399 28403 28404

10286 479 28400 14800 479 19692 28401 16948 17127 28403 28404 20442

10286 28408 17424 17424 493 19692 17127 28398 16948 20442 24625 119

102]]

NumPy массивы:

Тип: <class 'numpy.ndarray'>

Форма: (1, 37)

Значения: [[ 101 462 20442 17424 28401 19692 20442 493 19692 28399 28403 28404

10286 479 28400 14800 479 19692 28401 16948 17127 28403 28404 20442

10286 28408 17424 17424 493 19692 17127 28398 16948 20442 24625 119

102]]

--------------------------------------------------

--- Демонстрация пакетной обработки ---

Пакет текстов: ['Первое предложение в батче.', 'Второе, более длинное предложение, для демонстрации паддинга.']

Максимальная длина: 20

Input IDs (батч):

tensor([[ 101, 462, 19692, 20442, 28394, 16948, 19692, 490, 20442, 19692,

28396, 28400, 16948, 28397, 19692, 17127, 17424, 19692, 477, 102],

[ 101, 450, 28404, 16948, 20442, 16948, 19692, 117, 476, 16948,

28400, 19692, 19692, 479, 28400, 17424, 17127, 17127, 16948, 102]])

Attention Mask (батч):

tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

--------------------------------------------------

--- Демонстрация управления словарем ---

Исходный размер словаря: 28996

ID для токена 'hello': 19082

Токен для ID 100: '[UNK]' (обычно [UNK] для BERT)

Попытка добавить новые токены: ['[МОЙ\_НОВЫЙ\_ТОКЕН]', 'машинное\_обучение']

Добавлено токенов: 2

Новый размер словаря токенизатора: 28998

ID для '[МОЙ\_НОВЫЙ\_ТОКЕН]': 28996

ID для 'машинное\_обучение': 28997

Демонстрация изменения размера embedding-слоя модели:

Не удалось изменить размер embedding-слоя модели: 'BertForMaskedLM' object has no attribute 'embeddings'

--------------------------------------------------

--- Обучение пользовательского BPE токенизатора ---

Корпус для обучения (первые 3 строки): ['Машинное обучение - это область искусственного интеллекта.', 'Нейронные сети являются ключевым компонентом глубокого обучения.', 'Обработка естественного языка использует токенизацию.']...

Желаемый размер словаря: 200

Минимальная частота токена: 1

Путь сохранения: my\_custom\_bpe\_tokenizer.json

Начинается обучение токенизатора...

Обучение токенизатора завершено.

Пользовательский токенизатор сохранен в 'my\_custom\_bpe\_tokenizer.json'

Загрузка и тестирование пользовательского токенизатора...

Тестовый текст: 'Это тестовое предложение для проверки нового токенизатора.'

Input IDs (пользовательский токенизатор): tensor([[101, 62, 38, 50, 24, 35, 27, 80, 27, 26, 32, 35, 52, 27,

73, 80, 35, 24, 110, 31, 29, 53, 24, 35, 54, 163, 6]])

Attention Mask (пользовательский токенизатор): tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1]])

Декодированный текст (пользовательский токенизатор): 'Этотестовоепредлоениедляпроверкиновоготокенизатора.'

Таким образом, HuggingFace Tokenizers – это мощный и незаменимый инструмент в арсенале любого специалиста по НЛП. Его скорость, гибкость и интеграция с библиотекой Transformers делают его идеальным выбором для подготовки текстовых данных для современных языковых моделей. Понимание различных методов токенизации, процесса токенизации и нюансов использования библиотеки позволяет эффективно работать с текстом и добиваться высоких результатов в задачах НЛП. Освоив HuggingFace Tokenizers, вы заложите прочный фундамент для своих проектов в области обработки естественного языка.

## **§5.** Стеминг

**Что такое стемминг?**

**Стемминг** — это процесс приведения словоформ к их «корню» или «основе» путем отсечения суффиксов. Цель стемминга состоит в том, чтобы свести морфологически связанные слова к одной общей форме, даже если эта форма не является настоящим словом. Например, стеммер может привести слова "running", "ran", "runs" к общему корню "run", а слова "beautiful", "beautifully" — к "beauti".

Основное преимущество стемминга заключается в его простоте и высокой скорости. Однако, он может быть слишком агрессивным (отсекать слишком много, например, "universal" -> "univers") или слишком консервативным (не отсекать достаточно, например, "data" -> "data" вместо "dat"). Лемматизация, в свою очередь, использует словарь и морфологические правила для нахождения настоящей словарной формы слова (**леммы**), что делает её более точной, но и более медленной и ресурсоемкой.

Далее мы подробно рассмотрим пять основных методов стемминга.

**🌱** 1. Porter Stemmer

Porter Stemmer, разработанный Мартином Портером в 1980 году, является одним из самых известных и широко используемых алгоритмов стемминга для английского языка. Его алгоритм состоит из пяти последовательных шагов, каждый из которых представляет собой набор правил для отсечения суффиксов. Правила применяются по порядку, и на каждом шаге проверяется, соответствует ли слово определенным критериям, таким как длина слова или наличие гласных. Алгоритм не использует словарь, что делает его очень быстрым. Он работает по принципу "наибольший суффикс отсекается первым", что обеспечивает консистентность. Несмотря на свою простоту, он эффективно справляется со многими задачами. Однако, его основной недостаток — это агрессивность, которая часто приводит к созданию основ, не являющихся настоящими словами.

Пример алгоритма:

Рассмотрим слово "caresses":

**Шаг 1:** Удаление суффикса -es (т.к. слово заканчивается на -es и его длина больше 3). Слово становится "caress".

**Шаг 2:** Проверка наличия суффиксов вроде -ed, -ing, -ly. Ничего не найдено.

**Шаг 3:** Проверка суффиксов вроде -ational. Ничего не найдено.

**Шаг 4:** Проверка суффиксов вроде -ous. Ничего не найдено.

Шаг 5: Завершающие правила. Слово "caress" остается неизменным.  
Итоговая основа: "caress".

Реализация на Python:

Для реализации Porter Stemmer в Python мы будем использовать библиотеку NLTK, которая предоставляет готовый класс PorterStemmer.

|  |
| --- |
| import nltk  from nltk.stem import PorterStemmer  # Загрузка Porter Stemmer  porter\_stemmer = PorterStemmer()  # Список слов для стемминга  words = ["caresses", "flies", "running", "crying", "agreed", "classification"]  # Применение Porter Stemmer к каждому слову  print("Стемминг с помощью Porter Stemmer:")  for word in words:      stem = porter\_stemmer.stem(word)      print(f"  Исходное слово: {word:<15}  ->  Основа: {stem}") |

Результат

Стемминг с помощью Porter Stemmer:

Исходное слово: caresses -> Основа: caress

Исходное слово: flies -> Основа: fli

Исходное слово: running -> Основа: run

Исходное слово: crying -> Основа: cri

Исходное слово: agreed -> Основа: agre

Исходное слово: classification -> Основа: classif

**🌱** 2. Lovins Stemmer

Lovins Stemmer — один из первых алгоритмов стемминга, разработанный Джулией Ловинс в 1968 году. Он отличается от Porter Stemmer тем, что использует подход "одного прохода". В его основе лежит большой список из более чем 290 суффиксов, упорядоченных по длине и приоритету. Алгоритм ищет самый длинный суффикс из этого списка, который соответствует окончанию слова, и отсекает его. После отсечения суффикса применяются правила для исправления корня слова (например, "slowing" -> "slow", но "slowly" -> "slow"). Lovins Stemmer более консервативен, чем Porter, и реже создает нереальные слова. Однако его алгоритм менее гибок и может не справиться с некоторыми случаями, которые Porter обрабатывает успешно.

Пример алгоритма:

Рассмотрим слово "generously":

Алгоритм находит в своем списке суффиксов "-ously".

Он отсекает этот суффикс, получая "gener".

Затем применяются правила для исправления "gener" на "generous".  
Итоговая основа: "generous".

Реализация на Python:

В NLTK также есть реализация Lovins Stemmer, которую можно использовать.

|  |
| --- |
| class LovinsStemmer:      def \_\_init\_\_(self):          # Список суффиксов (пример, не полный)          self.suffixes = {              'ization': '',              'ational': '',              'fulness': '',              'ousness': '',              'iveness': '',              'tional': '',              'biliti': '',              'lessli': '',              'entli': '',              'ation': '',              'alism': '',              'aliti': '',              'ousli': '',              'iviti': '',              'fulli': '',              'enci': '',              'anci': '',              'abli': '',              'izer': '',              'ator': '',              'alli': '',              'bli': '',              'ogi': 'og',              'li': '',              'ing': '',              'ed': '',              'ly': '',              'es': '',              's': '',          }      def stem(self, word):          word = word.lower()          for suffix in sorted(self.suffixes.keys(), key=len, reverse=True):              if word.endswith(suffix):                  stemmed = word[:-len(suffix)] + self.suffixes[suffix]                  return stemmed          return word  # Тестирование  lovins = LovinsStemmer()  words = ["generously", "doing", "beautiful", "rejection"]  print("Пример стемминга Lovins Stemmer:")  for w in words:      print(f"{w:<15} -> {lovins.stem(w)}") |

Результат:

Пример стемминга Lovins Stemmer:

generously -> generous

doing -> do

beautiful -> beautiful

rejection -> rejection

**🌱** 3. Paice-Husk Stemmer

Paice-Husk Stemmer (также известный как Lancaster Stemmer) — это итеративный, высокоагрессивный алгоритм, разработанный Кристофером Пайсом и Полом Хаском. Его ключевая особенность — использование списка из 120 правил, которые применяются к слову циклически до тех пор, пока не перестанут срабатывать. Каждое правило состоит из суффикса, который нужно найти, замены, которую нужно сделать (включая отсечение), и условия, при котором правило может быть применено. Например, правило a:b\* может означать "если слово заканчивается на 'a', замени его на 'b', если слово не заканчивается на 'c'". Этот алгоритм известен своей агрессивностью и часто создает очень короткие, нечитаемые основы, что может быть как преимуществом, так и недостатком, в зависимости от задачи.

Пример алгоритма: Рассмотрим слово "declarations":

Сначала может быть найдено правило, отсекающее "-ations" и заменяющее на "-ate", получая "declarate".

На следующем цикле другое правило может отсечь "-ate", получая "declar".

Процесс продолжается до тех пор, пока ни одно правило больше не сработает.  
Итоговая основа может быть "declar".

Реализация на Python:

NLTK предоставляет реализацию Paice-Husk Stemmer.

|  |
| --- |
| import nltk  from nltk.stem import LancasterStemmer  # Загрузка Paice-Husk Stemmer (в NLTK он называется LancasterStemmer)  lancaster\_stemmer = LancasterStemmer()  # Список слов для стемминга  words = ["presumably", "rejection", "protestor", "community"]  # Применение Paice-Husk Stemmer к каждому слову  print("Стемминг с помощью Paice-Husk Stemmer (Lancaster):")  for word in words:      stem = lancaster\_stemmer.stem(word)      print(f"  Исходное слово: {word:<15}  ->  Основа: {stem}") |

Результат:

Стемминг с помощью Paice-Husk Stemmer (Lancaster):

Исходное слово: presumably -> Основа: presum

Исходное слово: rejection -> Основа: reject

Исходное слово: protestor -> Основа: protest

Исходное слово: community -> Основа: commun

**🌱** 4. Snowball (Porter2) Stemmer

Snowball, или Porter2, — это улучшенная и более современная версия оригинального Porter Stemmer, разработанная тем же Мартином Портером. Она была создана для устранения некоторых недостатков и ошибок Porter Stemmer, делая его правила более точными и менее агрессивными. Ключевым преимуществом Snowball является его **многоязычность**: он имеет отдельные алгоритмы стемминга для множества языков, включая русский, испанский, французский, немецкий и др. Это делает его очень гибким и мощным инструментом для проектов, работающих с текстами на разных языках. В отличие от Porter, который часто отсекает слишком много, Snowball более консервативен в спорных случаях, что приводит к более качественным результатам.

Пример алгоритма: Рассмотрим английское слово "generously":

Porter может дать "gener" (слишком агрессивно).

Snowball, с его уточненными правилами, скорее всего даст "generous", что является более приемлемой основой.

Реализация на Python:

SnowballStemmer в NLTK поддерживает несколько языков. Мы покажем примеры для английского и русского.

|  |
| --- |
| import nltk  from nltk.stem import SnowballStemmer  # Загрузка Snowball Stemmer для английского языка  english\_stemmer = SnowballStemmer("english")  # Загрузка Snowball Stemmer для русского языка  russian\_stemmer = SnowballStemmer("russian")  # Список слов для стемминга на английском  words\_en = ["caresses", "flies", "running", "crying", "agreed", "community"]  # Применение Snowball Stemmer к английским словам  print("Стемминг с помощью Snowball Stemmer (English):")  for word in words\_en:      stem = english\_stemmer.stem(word)      print(f"  Исходное слово: {word:<15}  ->  Основа: {stem}")  # Список слов для стемминга на русском  words\_ru = ["кошки", "бежали", "читают", "универсальные", "прекрасный"]  # Применение Snowball Stemmer к русским словам  print("\nСтемминг с помощью Snowball Stemmer (Russian):")  for word in words\_ru:      stem = russian\_stemmer.stem(word)      print(f"  Исходное слово: {word:<15}  ->  Основа: {stem}") |

Результат:

Стемминг с помощью Snowball Stemmer (English):

Исходное слово: caresses -> Основа: caress

Исходное слово: flies -> Основа: fli

Исходное слово: running -> Основа: run

Исходное слово: crying -> Основа: cri

Исходное слово: agreed -> Основа: agre

Исходное слово: community -> Основа: communiti

Стемминг с помощью Snowball Stemmer (Russian):

Исходное слово: кошки -> Основа: кошк

Исходное слово: бежали -> Основа: бежа

Исходное слово: читают -> Основа: чита

Исходное слово: универсальные -> Основа: универсальн

Исходное слово: прекрасный -> Основа: прекрасн

**🌱** 5. ML-based Stemming

Стемминг на основе машинного обучения — это современный подход, который отходит от традиционных правил и словарей. Вместо этого он использует модели, обученные на больших наборах данных, чтобы научиться предсказывать основу слова. Этот метод рассматривает стемминг как задачу преобразования последовательности символов (исходное слово) в другую последовательность (основу). Такие модели, как **последовательность-к-последовательности (seq2seq)** с использованием рекуррентных нейронных сетей (RNN) или трансформеров, могут быть обучены на данных, содержащих пары (слово, основа). Модель учится паттернам, по которым суффиксы отсекаются или изменяются, и может даже обрабатывать нерегулярные случаи и внесловарные слова, которых нет в обучающей выборке.

**Пример алгоритма:**

* **Обучение:** Модель получает на вход слова, например "running", и на выходе должна сгенерировать "run". После тысяч таких примеров она учится, что суффикс "-ing" обычно отсекается.
* **Инференс:** Когда на вход подается новое слово, например "jumping", модель применяет свои выученные паттерны для генерации "jump".

Реализация на Python:

Напрямую "из коробки" готовой и простой для использования ML-based стеммер-библиотеки, как в случае с NLTK, не существует. Обычно такие модели создаются с помощью фреймворков для глубокого обучения (например, PyTorch или TensorFlow). Мы покажем концептуальный пример, как это могло бы выглядеть.

|  |
| --- |
| # Для реализации ML-based стемминга вам понадобятся библиотеки  # для глубокого обучения, например, PyTorch или TensorFlow.  # Здесь мы покажем концептуальный код, не требующий установки этих библиотек.  class MLStemmerConceptual:      """      Концептуальный класс для демонстрации стемминга на основе машинного обучения.      В реальном проекте здесь будет код, использующий обученную модель.      """      def \_\_init\_\_(self, model\_path="my\_trained\_stemming\_model.pt"):          # В реальной реализации здесь происходит загрузка предварительно обученной модели.          # self.model = load\_model(model\_path)          # self.tokenizer = load\_tokenizer(model\_path)          print(f"Концептуальный стеммер на основе ML инициализирован.")          print(f"Была бы загружена модель из файла '{model\_path}'.")      def stem(self, word):          """          Предсказывает основу слова, используя обученную модель.          """          # В реальной реализации здесь происходит:          # 1. Токенизация слова.          # 2. Подача токенизированного слова на вход модели.          # 3. Генерация основы слова моделью.          # 4. Декодирование результата в строку.            # Для демонстрации мы используем простой маппинг.          simple\_mapping = {              "running": "run",              "jumped": "jump",              "unbelievable": "unbeliev",              "better": "bet"          }            # Здесь мы имитируем "предсказание" модели          stem = simple\_mapping.get(word.lower(), word.lower())          return stem  # Использование концептуального класса  ml\_stemmer = MLStemmerConceptual()  words = ["running", "jumped", "unbelievable", "better", "unknown\_word"]  print("\nКонцептуальный стемминг на основе машинного обучения:")  for word in words:      stem = ml\_stemmer.stem(word)      print(f"  Исходное слово: {word:<15}  ->  Основа: {stem}") |

Результат:

Концептуальный стеммер на основе ML инициализирован.

Была бы загружена модель из файла 'my\_trained\_stemming\_model.pt'.

Концептуальный стемминг на основе машинного обучения:

Исходное слово: running -> Основа: run

Исходное слово: jumped -> Основа: jump

Исходное слово: unbelievable -> Основа: unbeliev

Исходное слово: better -> Основа: bet

Исходное слово: unknown\_word -> Основа: unknown\_word

**🌱** Заключение и Сравнительная Таблица

Выбор метода стемминга зависит от конкретной задачи и требований к точности и скорости.

* **Porter Stemmer** — классический и быстрый, подходит для общего английского текста.
* **Lovins Stemmer** — более старый и консервативный, используется реже, но интересен с исторической точки зрения.
* **Paice-Husk Stemmer** — очень агрессивный и настраиваемый, может быть полезен, когда нужно получить очень короткие основы.
* **Snowball (Porter2) Stemmer** — современный стандарт, лучший выбор для большинства задач, так как он более точный, чем Porter, и поддерживает множество языков.
* **ML-based Stemming** — самый точный, но и самый сложный и ресурсоемкий метод, идеален для задач, где требуется максимальное качество, и есть доступ к большим объемам данных для обучения.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Метод** | **Подход** | **Языки** | **Преимущества** | **Недостатки** |
| **Porter** | Правила, 5 шагов | Английский | Простой, быстрый, широко используется | Агрессивный, создает ненастоящие слова |
| **Lovins** | Правила, 1 проход | Английский | Более консервативен | Менее точен, устарел |
| **Paice-Husk** | Итеративные правила | Английский | Настраиваемый, агрессивный | Сложный, может быть слишком агрессивным |
| **Snowball** | Улучшенные правила | Множество | Более точный, многоязычный | Всё ещё на основе правил, может ошибаться |
| **ML-based** | Обучение на данных | Множество | Высокая точность, контекст | Сложный, требует данных и ресурсов |

## **§6.** Лематизация

Этот раздел посвящен **лемматизации**, фундаментальной технике в области обработки естественного языка (NLP). Лемматизация — это процесс приведения различных флективных форм слова к одной базовой или словарной форме, называемой **леммой**. Например, слова "бежать", "бежит", "бежал" и "бегущий" имеют общую лемму "бежать". В отличие от **стемминга**, который часто просто отсекает суффиксы и может привести к нереальным словам (например, "красивый" -> "красив"), лемматизация использует словарный запас и морфологический анализ для получения корректной словарной формы. Это крайне важно для задач, где необходимо понимать истинное значение слова, таких как информационный поиск, машинный перевод, анализ настроений и кластеризация текстов, поскольку позволяет унифицировать различные формы одного и того же слова, значительно улучшая точность последующего анализа.

**1. Лемматизация на основе словаря (Dictionary-based Lemmatization)**

**Лемматизация на основе словаря** является одним из самых простых и прямолинейных подходов к приведению слов к их базовой форме. Суть этого метода заключается в сопоставлении каждого слова в тексте с заранее определенным словарем. Этот словарь представляет собой набор пар, где каждая пара состоит из флективной формы слова и ее соответствующей леммы.

Принцип работы этого метода довольно прост: сначала создается или используется уже существующий словарь, в котором ключами являются различные флективные формы слов (например, "бежит", "кошки", "были"), а значениями — их леммы ("бежать", "кошка", "быть"). Затем каждое слово из входного текста последовательно ищется в этом словаре. Если слово найдено, оно немедленно заменяется своей соответствующей леммой. В случае, если слово отсутствует в словаре (так называемое **внесловарное слово**, или OOV - Out-Of-Vocabulary), оно либо остается неизменным, либо для его обработки применяется некий резервный механизм, например, стемминг, или оно просто игнорируется. Этот подход обеспечивает высокую точность для слов, присутствующих в словаре, и относительно прост в реализации, поскольку не требует сложного морфологического анализа или разработки лингвистических правил. Однако его основным недостатком является неспособность обрабатывать внесловарные слова, а также необходимость постоянного обновления словаря для новых слов, сленга или редких форм. Кроме того, этот метод обычно не учитывает контекст слова, что может привести к неверной лемматизации для омографов (слов, которые пишутся одинаково, но имеют разное значение и, возможно, разную лемму в зависимости от контекста, например, английское "read" в прошедшем и настоящем времени).

**Пример реализации на Python**

|  |
| --- |
| # Пример словаря для лемматизации  lemma\_dictionary = {      "бежит": "бежать",      "бежал": "бежать",      "бегущий": "бежать",      "кошки": "кошка",      "собаки": "собака",      "лучший": "хороший",      "были": "быть",      "есть": "быть",      "читает": "читать",      "читал": "читать"  }  def dictionary\_lemmatize(word, dictionary):      """      Лемматизирует слово на основе заданного словаря.      Если слово не найдено в словаре, возвращает исходное слово.      """      return dictionary.get(word.lower(), word) # Используем .lower() для регистронезависимого поиска  # Исходный текст для лемматизации  text = "Кошки бежит по улице, а собаки читает книгу. Он был лучший."  # Разделяем текст на слова, удаляя знаки препинания для простоты  words = text.replace('.', '').replace(',', '').split()  # Применяем словарную лемматизацию к каждому слову  lemmatized\_words = [dictionary\_lemmatize(word, lemma\_dictionary) for word in words]  lemmatized\_text = " ".join(lemmatized\_words)  print(f"Исходный текст: {text}")  print(f"Лемматизированный текст: {lemmatized\_text}")  # Пример с неизвестным словом (внесловарное слово)  unknown\_word\_text = "Он быстро прыгает и смеется."  unknown\_words = unknown\_word\_text.replace('.', '').replace(',', '').split()  lemmatized\_unknown\_words = [dictionary\_lemmatize(word, lemma\_dictionary) for word in unknown\_words]  print(f"\nИсходный текст (с неизвестными словами): {unknown\_word\_text}")  print(f"Лемматизированный текст (с неизвестными словами): {' '.join(lemmatized\_unknown\_words)}") |

**Вывод:**

Исходный текст: Кошки бежит по улице, а собаки читает книгу. Он был лучший.

Лемматизированный текст: кошка бежать по улице а собака читать книгу Он был хороший

Исходный текст (с неизвестными словами): Он быстро прыгает и смеется.

Лемматизированный текст (с неизвестными словами): Он быстро прыгает и смеется

**2. Лемматизация на основе правил (Rule-based Lemmatization)**

**Лемматизация на основе правил** представляет собой подход, который использует заранее определенный набор лингвистических правил для определения леммы слова. Эти правила обычно формулируются на основе глубокого понимания морфологических особенностей конкретного языка, таких как суффиксы, префиксы, изменения в корне слова и другие грамматические признаки. Например, одно из правил для русского языка может гласить: "если глагол заканчивается на '-ает', отбрось '-ет' и добавь '-ать'", чтобы преобразовать "читает" в "читать".

Процесс работы начинается с тщательной разработки этих правил, которая требует значительных усилий со стороны лингвистов и экспертов в области языка. Правила обычно применяются в определенном, строго заданном порядке, часто от более специфичных к более общим, чтобы избежать конфликтов и обеспечить максимальную точность. Для каждого слова в тексте система проверяет, соответствует ли оно какому-либо из заданных правил. Как только правило срабатывает, слово преобразуется в соответствии с этим правилом. Основное преимущество этого подхода заключается в его способности обрабатывать **внесловарные слова**, если их морфологическая структура соответствует существующим правилам, а также в том, что он не требует огромных словарей. Однако разработка и поддержка такого набора правил чрезвычайно трудоемки, особенно для языков с богатой и нерегулярной морфологией (например, русский язык с его множеством исключений и сложными склонениями/спряжениями). Существует высокая вероятность возникновения ошибок, если слово не соответствует ни одному правилу или, наоборот, соответствует нескольким противоречивым правилам. Как и словарный подход, лемматизация на основе правил обычно не учитывает контекст слова, что ограничивает ее способность разрешать лексическую неоднозначность.

**Пример реализации на Python**

|  |
| --- |
| def rule\_based\_lemmatize(word):      """      Лемматизирует слово на основе простых правил для русского языка.      Этот пример является очень упрощенным и не охватывает всю сложность      морфологии русского языка, но демонстрирует принцип.      """      word\_lower = word.lower()      # Правила для глаголов (очень базовые, для демонстрации)      if word\_lower.endswith("ает"):          return word\_lower[:-3] + "ать"      if word\_lower.endswith("ует"):          return word\_lower[:-3] + "овать"      if word\_lower.endswith("ит"):          return word\_lower[:-2] + "ить"      if word\_lower.endswith("ел"): # Например, "хотел" -> "хотеть"          return word\_lower[:-2] + "еть"      if word\_lower.endswith("тся"): # Например, "смеются" -> "смеяться"          return word\_lower[:-3] + "ться"      if word\_lower.endswith("лись"): # Например, "смеялись" -> "смеяться"          return word\_lower[:-4] + "ться"      # Правила для существительных (очень базовые)      if word\_lower.endswith("и"): # Например, "столы" -> "стол", но избегаем "дни" -> "дн"          if len(word\_lower) > 2 and word\_lower[-2] not in "аеёиоуыэюя":               return word\_lower[:-1]      if word\_lower.endswith("ы"): # Например, "книги" -> "книга"          return word\_lower[:-1] + "а"      if word\_lower.endswith("ами"): # Например, "руками" -> "рука"          return word\_lower[:-3] + "а"      if word\_lower.endswith("ом"): # Например, "столом" -> "стол"          return word\_lower[:-2]      # Правила для прилагательных (очень базовые)      if word\_lower.endswith("ый"):          return word\_lower[:-2] + "ой"      if word\_lower.endswith("ая"):          return word\_lower[:-2] + "ая" # Или просто "ой" для мужского рода      if word\_lower.endswith("ие"): # Например, "красивые" -> "красивый"          return word\_lower[:-2] + "ый"      if word\_lower.endswith("ого"): # Например, "красивого" -> "красивый"          return word\_lower[:-3] + "ый"      return word # Возвращаем исходное слово, если ни одно правило не сработало  # Исходный текст  text = "Мальчик читает книги и думает о красивых днях. Он хотел бежать и смеялся."  words = text.replace('.', '').replace(',', '').split()  # Применяем лемматизацию на основе правил  lemmatized\_words = [rule\_based\_lemmatize(word) for word in words]  lemmatized\_text = " ".join(lemmatized\_words)  print(f"Исходный текст: {text}")  print(f"Лемматизированный текст: {lemmatized\_text}") |

**Вывод:**

Исходный текст: Мальчик читает книги и думает о красивых днях. Он хотел бежать и смеялся.

Лемматизированный текст: Мальчик читать книг и думать о красивых днях Он хотеть бежать и смеялся

**Примечание:** Этот пример является *очень* упрощенным. Реальные системы на основе правил гораздо сложнее и содержат тысячи правил и исключений.

**3. WordNet Lemmatizer (NLTK)**

**WordNet Lemmatizer** — это специализированный лемматизатор, являющийся частью популярной библиотеки Natural Language Toolkit (NLTK) для Python. Он разработан для работы с английским языком и использует обширную лексическую базу данных под названием **WordNet**. WordNet — это не просто словарь, а структурированная база данных, которая организует английские слова в наборы синонимов, называемые **синсетами** (synsets). Каждый синсет представляет собой отдельное лексическое понятие, и между этими синсетами записаны различные семантические отношения (например, гиперонимия/гипонимия, меронимия и т.д.).

Лемматизатор WordNet использует эту богатую структуру для определения леммы слова. Его принцип работы основан на поиске слова в базе данных WordNet и применении морфологических правил, которые определяют, как различные флективные формы связаны с их базовыми формами. Ключевой особенностью этого лемматизатора является возможность принимать аргумент pos (part-of-speech, часть речи). Указание части речи значительно повышает точность лемматизации, поскольку многие слова имеют одинаковую форму, но разные леммы в зависимости от их грамматической роли (например, "run" может быть глаголом или существительным). Без явного указания части речи лемматизатор по умолчанию считает слово существительным, что может привести к неверным результатам для глаголов, прилагательных или наречий. Например, если слово "running" и указана часть речи "глагол", WordNet Lemmatizer найдет лемму "run". Если часть речи не указана, он может вернуть "running" (как существительное) или "run" (если это единственная форма, но менее надежно). Этот метод обеспечивает высокую точность для английского языка, особенно при использовании информации о части речи, но его главный недостаток заключается в том, что он **предназначен исключительно для английского языка** и не может быть использован для других языков. Кроме того, для максимальной точности требуется предварительный этап определения части речи (POS-теггинг), что добавляет дополнительный шаг в пайплайн обработки.

**Пример реализации на Python**

|  |
| --- |
| import nltk  from nltk.stem import WordNetLemmatizer  from nltk.corpus import wordnet  from nltk.tag import pos\_tag  from nltk.tokenize import word\_tokenize  # Загрузка всех необходимых ресурсов NLTK  def download\_nltk\_resources():      resource\_paths = {          'punkt': 'tokenizers/punkt',          'averaged\_perceptron\_tagger\_eng': 'taggers/averaged\_perceptron\_tagger\_eng',          'wordnet': 'corpora/wordnet',          'omw-1.4': 'corpora/omw-1.4'      }      for name, path in resource\_paths.items():          try:              nltk.data.find(path)          except LookupError:              print(f"Загрузка ресурса {name}...")              nltk.download(name)    # Вызываем функцию загрузки ресурсов  download\_nltk\_resources()  # Инициализация лемматизатора WordNet  lemmatizer = WordNetLemmatizer()  def get\_wordnet\_pos(treebank\_tag):      """      Преобразует теги частей речи из формата NLTK Treebank      в формат, понятный WordNet Lemmatizer.      """      if treebank\_tag.startswith('J'):          return wordnet.ADJ      elif treebank\_tag.startswith('V'):          return wordnet.VERB      elif treebank\_tag.startswith('N'):          return wordnet.NOUN      elif treebank\_tag.startswith('R'):          return wordnet.ADV      else:          return wordnet.NOUN  # По умолчанию — существительное  # Исходный текст на английском языке  text\_en = "The cats were running quickly. He has better ideas."  # 1. Токенизация  tokens = word\_tokenize(text\_en)  print(f"Токены: {tokens}")  # 2. POS-теггинг  tagged\_tokens = pos\_tag(tokens)  print(f"Тегированные токены: {tagged\_tokens}")  # 3. Лемматизация с учетом POS  lemmatized\_words = [      lemmatizer.lemmatize(word, get\_wordnet\_pos(tag))      for word, tag in tagged\_tokens  ]  lemmatized\_text = " ".join(lemmatized\_words)  print(f"\nИсходный текст: {text\_en}")  print(f"Лемматизированный текст (с учетом POS): {lemmatized\_text}")  # Для сравнения: без учета POS  lemmatized\_words\_no\_pos = [lemmatizer.lemmatize(word) for word in tokens]  lemmatized\_text\_no\_pos = " ".join(lemmatized\_words\_no\_pos)  print(f"Лемматизированный текст (без учета POS): {lemmatized\_text\_no\_pos}") |

**Вывод:**

Загрузка ресурса omw-1.4...

Токены: ['The', 'cats', 'were', 'running', 'quickly', '.', 'He', 'has', 'better', 'ideas', '.']

Тегированные токены: [('The', 'DT'), ('cats', 'NNS'), ('were', 'VBD'), ('running', 'VBG'), ('quickly', 'RB'), ('.', '.'), ('He', 'PRP'), ('has', 'VBZ'), ('better', 'JJR'), ('ideas', 'NNS'), ('.', '.')]

Исходный текст: The cats were running quickly. He has better ideas.

Лемматизированный текст (с учетом POS): The cat be run quickly . He have good idea .

Лемматизированный текст (без учета POS): The cat were running quickly . He ha better idea .

Как видно из вывода, при использовании POS-теггинга were лемматизируется в be, running в run, а better в good, что является более точными и желаемыми леммами. Без учета части речи were и running остаются неизменными (так как по умолчанию рассматриваются как существительные), а has становится ha, что не является корректной леммой.

**4. spaCy Lemmatizer**

**spaCy** — это высокопроизводительная библиотека NLP для Python, разработанная с акцентом на использование в производственной среде. Её лемматизатор является неотъемлемой частью более широкого и мощного **пайплайна обработки текста**, который включает в себя такие этапы, как токенизация (разделение текста на слова и знаки препинания), POS-теггинг (определение части речи), синтаксический анализ зависимостей (выявление грамматических связей между словами) и распознавание именованных сущностей (NER). Лемматизатор spaCy использует комбинацию статистических моделей и правил, которые были обучены на больших корпусах текстовых данных. Это позволяет ему достигать очень высокой точности и эффективно обрабатывать даже **внесловарные слова** (слова, которые не встречались в обучающих данных).

Принцип работы лемматизатора spaCy заключается в том, что после загрузки предварительно обученной модели для конкретного языка (например, en\_core\_web\_sm для английского или ru\_core\_news\_sm для русского), текст пропускается через весь NLP-пайплайн. В процессе этого пайплайна лемматизатор получает информацию от других компонентов, в частности от POS-теггера. Используя эту информацию о части речи, а также контекст слова в предложении, лемматизатор определяет наиболее вероятную и корректную лемму для каждого токена. Модели spaCy специально разработаны для обработки нерегулярных форм и разрешения лексической неоднозначности, что делает их очень мощным инструментом. Основные преимущества spaCy включают высокую точность и производительность, поддержку множества языков, автоматический учет части речи и контекста, а также способность обрабатывать внесловарные слова. Однако, для работы с spaCy требуется загрузка довольно больших языковых моделей, что может быть недостатком для систем с ограниченными ресурсами или для очень простых задач, где не требуются другие функции NLP.

**Пример реализации на Python**

|  |
| --- |
| import spacy  # Загрузка языковой модели spaCy.  # Если модель не установлена, ее можно загрузить из командной строки:  # Для русского языка: python -m spacy download ru\_core\_news\_sm  # Для английского языка: python -m spacy download en\_core\_web\_sm  # Попытка загрузить русскую модель. Если не удается, загружаем английскую.  try:      nlp = spacy.load("ru\_core\_news\_sm")      print("Загружена русская модель spaCy.")  except OSError:      print("Русская модель spaCy не найдена. Пожалуйста, установите ее, выполнив 'python -m spacy download ru\_core\_news\_sm'.")      print("Загружаю английскую модель для демонстрации.")      nlp = spacy.load("en\_core\_web\_sm")      print("Загружена английская модель spaCy.")  # Примеры текстов для разных языков  text\_ru = "Кошки бежали по улице, а собаки читают книги. Он был лучшим."  text\_en = "The cats were running quickly. He has better ideas."  # Выбираем текст для обработки в зависимости от загруженной модели  if nlp.lang == 'ru':      text\_to\_process = text\_ru  else:      text\_to\_process = text\_en  # Обработка текста с помощью spaCy  doc = nlp(text\_to\_process)  print(f"\nИсходный текст: {text\_to\_process}")  print("Лемматизация с spaCy:")  # Итерируем по токенам в обработанном документе  for token in doc:      # Выводим исходное слово, его часть речи и лемму      print(f"  Слово: {token.text}, Часть речи: {token.pos\_}, Лемма: {token.lemma\_}")  # Пример с неоднозначностью для английского языка (если загружена английская модель)  if nlp.lang == 'en':      text\_ambiguous = "I read a book yesterday. I will read a book tomorrow."      doc\_ambiguous = nlp(text\_ambiguous)      print(f"\nИсходный текст: {text\_ambiguous}")      print("Лемматизация неоднозначных слов с spaCy:")      for token in doc\_ambiguous:          print(f"  Слово: {token.text}, Часть речи: {token.pos\_}, Лемма: {token.lemma\_}") |

Результат:

Русская модель spaCy не найдена. Пожалуйста, установите ее, выполнив 'python -m spacy download ru\_core\_news\_sm'.

Загружаю английскую модель для демонстрации.

Загружена английская модель spaCy.

Исходный текст: The cats were running quickly. He has better ideas.

Лемматизация с spaCy:

Слово: The, Часть речи: DET, Лемма: the

Слово: cats, Часть речи: NOUN, Лемма: cat

Слово: were, Часть речи: AUX, Лемма: be

Слово: running, Часть речи: VERB, Лемма: run

Слово: quickly, Часть речи: ADV, Лемма: quickly

Слово: ., Часть речи: PUNCT, Лемма: .

Слово: He, Часть речи: PRON, Лемма: he

Слово: has, Часть речи: VERB, Лемма: have

Слово: better, Часть речи: ADJ, Лемма: well

Слово: ideas, Часть речи: NOUN, Лемма: idea

Слово: ., Часть речи: PUNCT, Лемма: .

Исходный текст: I read a book yesterday. I will read a book tomorrow.

Лемматизация неоднозначных слов с spaCy:

Слово: I, Часть речи: PRON, Лемма: I

Слово: read, Часть речи: VERB, Лемма: read

Слово: a, Часть речи: DET, Лемма: a

Слово: book, Часть речи: NOUN, Лемма: book

Слово: yesterday, Часть речи: NOUN, Лемма: yesterday

Слово: ., Часть речи: PUNCT, Лемма: .

Слово: I, Часть речи: PRON, Лемма: I

Слово: will, Часть речи: AUX, Лемма: will

Слово: read, Часть речи: VERB, Лемма: read

Слово: a, Часть речи: DET, Лемма: a

Слово: book, Часть речи: NOUN, Лемма: book

Слово: tomorrow, Часть речи: NOUN, Лемма: tomorrow

Слово: ., Часть речи: PUNCT, Лемма:

spaCy демонстрирует высокую точность, правильно лемматизируя read как read в обоих контекстах, несмотря на одинаковую форму слова для разных времен в английском языке.

**5. Stanza Lemmatizer**

**Stanza** (ранее известная как StanfordNLP) — это передовая библиотека NLP от Стэнфордского университета, предоставляющая полный и унифицированный пайплайн обработки текста для множества языков. Она построена на основе фреймворка **PyTorch** и использует современные архитектуры **нейронных сетей** для всех своих компонентов. Пайплайн Stanza включает в себя токенизацию, многоязычное встраивание слов, POS-теггинг (определение части речи), морфологический анализ, лемматизацию, синтаксический анализ зависимостей и распознавание именованных сущностей (NER). Лемматизатор Stanza, подобно spaCy, работает в контексте этого комплексного пайплайна, используя глубокие нейронные сети, которые были обучены на обширных корпусах **Universal Dependencies**. Universal Dependencies — это проект, направленный на создание универсальных грамматических аннотаций для различных языков, что обеспечивает консистентность и сопоставимость результатов между языками.

Принцип работы Stanza Lemmatizer начинается с загрузки предварительно обученной модели для конкретного языка. Затем входной текст проходит через весь пайплайн Stanza. В процессе этого лемматизатор использует богатую информацию, полученную от других компонентов, такую как часть речи слова, его морфологические признаки (например, род, число, падеж для существительных; время, лицо, число для глаголов) и, что крайне важно, **контекст слова в предложении**. Благодаря использованию нейросетевых моделей, Stanza очень хорошо справляется с **внесловарными словами** и сложной, нерегулярной морфологией, что является значительным преимуществом по сравнению с более простыми методами. Основные преимущества Stanza включают высокую точность для большого числа языков, поддержку Universal Dependencies, эффективную обработку внесловарных слов и нерегулярных форм, а также предоставление полного и интегрированного пайплайна NLP. Однако, как и в случае с spaCy, Stanza требует загрузки довольно больших языковых моделей и может быть медленнее, чем более простые методы, из-за вычислительной сложности нейросетевых моделей.

**Пример реализации на Python**

|  |
| --- |
| import stanza  # === Установка моделей (если еще не установлены) ===  for lang\_code in ['ru', 'en']:      try:          stanza.download(lang\_code, verbose=False)      except Exception as e:          print(f"⚠️ Ошибка при загрузке модели для {lang\_code}: {e}")  # === Создание пайплайнов ===  pipelines = {}  try:      pipelines['ru'] = stanza.Pipeline(lang='ru', processors='tokenize,pos,lemma', verbose=False)      print("✅ Русская модель загружена.")  except Exception as e:      print(f"❌ Ошибка загрузки русской модели: {e}")  try:      pipelines['en'] = stanza.Pipeline(lang='en', processors='tokenize,mwt,pos,lemma', verbose=False)      print("✅ Английская модель загружена.")  except Exception as e:      print(f"❌ Ошибка загрузки английской модели: {e}")  # === Тексты для обработки ===  texts = {      'ru': "Кошки бежали по улице, а собаки читают книги. Он был лучшим.",      'en': "The cats were running quickly. He has better ideas."  }  # === Обработка всех доступных языков ===  for lang, text in texts.items():      if lang not in pipelines:          print(f"\n⚠️ Пропускаем {lang}: модель не загружена.")          continue      print(f"\n🌍 Язык: {lang.upper()}")      print(f"Исходный текст: {text}")      doc = pipelines[lang](text)      print("🧠 Лемматизация:")      for sentence in doc.sentences:          for word in sentence.words:              print(f"  Слово: {word.text}, Часть речи: {word.upos}, Лемма: {word.lemma}")  # === Пример с неоднозначными словами для английского ===  if 'en' in pipelines:      ambiguous\_text = "I read a book yesterday. I will read a book tomorrow."      doc2 = pipelines['en'](ambiguous\_text)      print(f"\n🔍 Неоднозначный английский текст: {ambiguous\_text}")      for sentence in doc2.sentences:          for word in sentence.words:              print(f"  Слово: {word.text}, Часть речи: {word.upos}, Лемма: {word.lemma}") |

Результат:

✅ Русская модель загружена.

✅ Английская модель загружена.

🌍 Язык: RU

Исходный текст: Кошки бежали по улице, а собаки читают книги. Он был лучшим.

🧠 Лемматизация:

Слово: Кошки, Часть речи: NOUN, Лемма: кошка

Слово: бежали, Часть речи: VERB, Лемма: бежать

Слово: по, Часть речи: ADP, Лемма: по

Слово: улице, Часть речи: NOUN, Лемма: улица

Слово: ,, Часть речи: PUNCT, Лемма: ,

Слово: а, Часть речи: CCONJ, Лемма: а

Слово: собаки, Часть речи: NOUN, Лемма: собака

Слово: читают, Часть речи: VERB, Лемма: читать

Слово: книги, Часть речи: NOUN, Лемма: книга

Слово: ., Часть речи: PUNCT, Лемма: .

Слово: Он, Часть речи: PRON, Лемма: он

Слово: был, Часть речи: AUX, Лемма: быть

Слово: лучшим, Часть речи: ADJ, Лемма: лучший

Слово: ., Часть речи: PUNCT, Лемма: .

🌍 Язык: EN

Исходный текст: The cats were running quickly. He has better ideas.

🧠 Лемматизация:

Слово: The, Часть речи: DET, Лемма: the

Слово: cats, Часть речи: NOUN, Лемма: cat

Слово: were, Часть речи: AUX, Лемма: be

Слово: running, Часть речи: VERB, Лемма: run

Слово: quickly, Часть речи: ADV, Лемма: quickly

Слово: ., Часть речи: PUNCT, Лемма: .

Слово: He, Часть речи: PRON, Лемма: he

Слово: has, Часть речи: VERB, Лемма: have

Слово: better, Часть речи: ADJ, Лемма: good

Слово: ideas, Часть речи: NOUN, Лемма: idea

Слово: ., Часть речи: PUNCT, Лемма: .

🔍 Неоднозначный английский текст: I read a book yesterday. I will read a book tomorrow.

Слово: I, Часть речи: PRON, Лемма: I

Слово: read, Часть речи: VERB, Лемма: read

Слово: a, Часть речи: DET, Лемма: a

Слово: book, Часть речи: NOUN, Лемма: book

Слово: yesterday, Часть речи: NOUN, Лемма: yesterday

Слово: ., Часть речи: PUNCT, Лемма: .

Слово: I, Часть речи: PRON, Лемма: I

Слово: will, Часть речи: AUX, Лемма: will

Слово: read, Часть речи: VERB, Лемма: read

Слово: a, Часть речи: DET, Лемма: a

Слово: book, Часть речи: NOUN, Лемма: book

Слово: tomorrow, Часть речи: NOUN, Лемма: tomorrow

Слово: ., Часть речи: PUNCT, Лемма: .

Stanza, как и spaCy, демонстрирует высокую точность, правильно определяя леммы даже в сложных случаях и для разных языков.

**6. Transformers-based Lemmatization (BERT, XLM-R и др.)**

**Лемматизация на основе трансформеров** представляет собой наиболее современный и мощный подход в NLP, использующий архитектуры глубоких нейронных сетей, такие как **BERT, XLM-R, RoBERTa, T5** и другие. Эти модели обучаются на огромных объемах текстовых данных (миллиарды слов) и благодаря своей архитектуре способны улавливать сложные лингвистические зависимости и глубокое контекстное значение слова в предложении. В отличие от традиционных методов, которые полагаются на явные правила или фиксированные словари, трансформеры учатся предсказывать лемму, основываясь на паттернах и семантических отношениях, которые они извлекают из обучающих данных.

Принцип работы трансформерных моделей для лемматизации заключается в следующем: входной текст подается на вход предварительно обученной модели трансформера. Модель обрабатывает текст, создавая **контекстуализированные встраивания (embeddings)** для каждого слова. Эти встраивания уникальны для каждого вхождения слова и отражают его значение в конкретном контексте предложения. Затем поверх этих встраиваний добавляется специализированный слой, который обучается предсказывать лемму для каждого слова. Существует несколько подходов к реализации лемматизации с использованием трансформеров:

* **Последовательная маркировка (Sequence Tagging):** Модель предсказывает лемму для каждого токена как отдельную метку, например, "бежали" -> "бежать". Это похоже на POS-теггинг, но с более сложным набором выходных меток.
* **Последовательность-к-последовательности (Sequence-to-Sequence):** Модель генерирует лемму как новую последовательность символов. Например, для входного слова "running" модель может генерировать "run". Это особенно полезно для языков с очень сложной морфологией или для обработки внесловарных слов, где лемма может быть значительно отличающейся от исходной формы.
* **Преобразование символов (Character-level Transformation):** Модель учится применять изменения к символам слова для получения леммы. Это может быть реализовано путем предсказания операций редактирования (например, удаление, вставка, замена символов).

Главными преимуществами трансформерных моделей являются их **высочайшая точность**, которая превосходит традиционные методы, особенно для внесловарных слов и нерегулярной морфологии, благодаря глубокому контекстному пониманию. Они также обладают **контекстно-зависимой лемматизацией**, что позволяет им различать леммы для омографов на основе окружающих слов. Модели, такие как XLM-R, обучены на множестве языков, что обеспечивает **многоязычность** и позволяет выполнять лемматизацию для разных языков без необходимости переобучения. Кроме того, они обладают **гибкостью** и могут быть дообучены (fine-tuned) на специфичных для задачи данных для еще большей точности. Однако, эти модели требуют значительных **вычислительных ресурсов** (GPU/TPU) для обучения и инференса, имеют очень **большой размер**, что затрудняет их развертывание на устройствах с ограниченными ресурсами, и их **сложность** требует глубоких знаний в области глубокого обучения для эффективной имплементации и понимания. Для оптимальной производительности на конкретных данных часто требуется дополнительное дообучение.

**Пример реализации на Python**

Использование моделей на основе трансформеров для лемматизации обычно осуществляется через библиотеку Hugging Face Transformers. Для этой задачи часто используются модели, которые были дообучены на задачах POS-теггинга или лемматизации, или же модели, способные к генерации текста (sequence-to-sequence).

Важно отметить, что библиотеки, такие как Stanza и spaCy, *уже* используют архитектуры, вдохновленные трансформерами или схожие с ними (например, Bi-LSTM с вниманием), для своих компонентов, включая лемматизацию. Таким образом, предыдущие примеры с spaCy и Stanza уже демонстрируют "трансформер-подобную" лемматизацию в действии, но с более удобным высокоуровневым API.

Для иллюстрации "чисто" трансформерной лемматизации, мы покажем, как можно было бы использовать модель для генерации последовательностей (например, T5), чтобы она "переводила" слово в его лемму. Это более сложный сценарий, требующий специализированного дообучения, так как общие модели T5 не обучены специально для лемматизации "из коробки".

|  |
| --- |
| from transformers import pipeline, AutoTokenizer, AutoModelForSeq2SeqLM  import torch  # --- Концептуальный пример: Лемматизация с использованием модели Sequence-to-Sequence (T5) ---  # ВНИМАНИЕ: Для реального использования, эта модель должна быть ДООБУЧЕНА  # на парах (флективная форма слова, лемма).  # Здесь мы используем общую T5 модель, которая не обучена специально для лемматизации,  # поэтому результаты будут неточными БЕЗ fine-tuning.  # Загрузка токенизатора и модели T5 (используем маленькую версию для примера)  # Это может занять некоторое время при первом запуске, так как модель довольно большая.  try:      tokenizer = AutoTokenizer.from\_pretrained("google/flan-t5-small")      model = AutoModelForSeq2SeqLM.from\_pretrained("google/flan-t5-small")      print("Загружена модель Flan-T5-small.")  except Exception as e:      print(f"Ошибка при загрузке модели Flan-T5-small: {e}")      print("Пропустим пример с трансформерами, так как он требует загрузки большой модели.")      tokenizer = None      model = None  if tokenizer and model:      # Создаем пайплайн для генерации текста.      # В реальном сценарии, промпт должен быть очень специфичным для задачи лемматизации,      # и модель должна быть дообучена на таких промптах.      generator = pipeline("text2text-generation", model=model, tokenizer=tokenizer)      def lemmatize\_with\_t5(word):          """          Концептуальная функция лемматизации с использованием T5.          Требует ДООБУЧЕНИЯ модели для получения точных результатов лемматизации.          """          # Формируем промпт для модели.          # Например, "lemmatize: running" -> "run"          prompt = f"lemmatize the word: {word}"          # Генерируем текст. max\_new\_tokens ограничивает длину сгенерированной леммы.          result = generator(prompt, max\_new\_tokens=10, num\_return\_sequences=1)          return result[0]['generated\_text']      words\_to\_lemmatize = ["running", "cats", "better", "was", "going", "бежали", "кошки", "читают"]      print("\nЛемматизация с использованием Flan-T5-small (концептуально, без дообучения):")      for word in words\_to\_lemmatize:          lemma = lemmatize\_with\_t5(word)          print(f"  Слово: {word}, Предсказанная лемма: {lemma}")  # --- Альтернативный подход: Использование трансформеров для POS-теггинга,  # --- который является ОСНОВОЙ для лемматизации в некоторых системах.  # --- (Этот подход уже реализован в spaCy и Stanza на более низком уровне)  # Пример использования пайплайна для POS-теггинга с BERT-подобной моделью  # (для русского языка, модель DeepPavlov/rubert-base-cased-sentence)  try:      # Загружаем пайплайн для классификации токенов (POS-теггинг)      pos\_tagger\_ru = pipeline("token-classification", model="DeepPavlov/rubert-base-cased-sentence", aggregation\_strategy="simple")      print("\nЗагружена модель DeepPavlov/rubert-base-cased-sentence для POS-теггинга.")      text\_ru\_pos = "Кошки бежали по улице."      tokens\_with\_pos = pos\_tagger\_ru(text\_ru\_pos)      print(f"Исходный текст для POS-теггинга: {text\_ru\_pos}")      print("POS-теггинг с DeepPavlov/rubert-base-cased-sentence:")      for token\_info in tokens\_with\_pos:          print(f"  Слово: {token\_info['word']}, Тег: {token\_info['entity\_group']}")  except Exception as e:      print(f"\nОшибка при загрузке модели DeepPavlov/rubert-base-cased-sentence: {e}")      print("Пропустим пример с POS-теггингом на основе трансформеров.") |

Результат:

Device set to use cpu

Загружена модель Flan-T5-small.

Лемматизация с использованием Flan-T5-small (концептуально, без дообучения):

Слово: running, Предсказанная лемма: lemmatize the word: running

Слово: cats, Предсказанная лемма: cat lemmatize

Слово: better, Предсказанная лемма: lemmatize the word better: better

Слово: was, Предсказанная лемма: lemmatize the word was

Слово: going, Предсказанная лемма: going to a party

Слово: бежали, Предсказанная лемма: lemmatize the word: е

Слово: кошки, Предсказанная лемма: lemmatize the word коки

Слово: читают, Предсказанная лемма: lemmatize the word: и

Some weights of BertForTokenClassification were not initialized from the model checkpoint at DeepPavlov/rubert-base-cased-sentence and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Device set to use cpu

Asking to truncate to max\_length but no maximum length is provided and the model has no predefined maximum length. Default to no truncation.

Загружена модель DeepPavlov/rubert-base-cased-sentence для POS-теггинга.

Исходный текст для POS-теггинга: Кошки бежали по улице.

POS-теггинг с DeepPavlov/rubert-base-cased-sentence:

Слово: Кошки бежали по улице., Тег: LABEL\_1

Важное примечание к выводу:

Модель Flan-T5-small не обучена специально для лемматизации, поэтому её "предсказания" лемм будут неточными и могут выглядеть как продолжение предложения, а не как чистая лемма. Чтобы T5 или другая seq2seq модель выполняла лемматизацию точно, её необходимо дообучить на датасете пар (слово, лемма). Пример с DeepPavlov показывает, как трансформеры могут выполнять POS-теггинг, который является важным шагом для контекстно-зависимой лемматизации.

Этот пример с T5 показывает *архитектурную возможность*, а не готовое решение для лемматизации без дообучения. Реальные трансформерные лемматизаторы, как упоминалось, часто интегрированы в более крупные NLP-пайплайны (как в Stanza) или являются специализированными моделями, дообученными именно для этой задачи.

**Заключение**

**Лемматизация** является критически важным шагом в предобработке текста для многих задач NLP, поскольку она обеспечивает унификацию слов и значительно улучшает качество последующего анализа. Выбор наиболее подходящего метода лемматизации зависит от нескольких ключевых факторов:

* **Язык:** Для английского языка **WordNet Lemmatizer (NLTK)** может быть хорошим выбором, особенно при использовании информации о части речи. Однако для других языков, особенно с богатой морфологией, требуются более универсальные и мощные решения, такие как **spaCy** или **Stanza**.
* **Требования к точности:** Для достижения максимальной точности, особенно при работе с внесловарными словами, нерегулярной морфологией и необходимостью учета контекста, предпочтительны современные библиотеки, такие как **spaCy** и **Stanza**, или специализированные **трансформерные модели**.
* **Вычислительные ресурсы:** Простые методы, такие как словарные или основанные на правилах, менее требовательны к вычислительным ресурсам. В то же время, трансформерные модели требуют значительных мощностей (GPU/TPU) для обучения и инференса из-за своего большого размера и сложности.
* **Сложность реализации:** Использование готовых библиотек, таких как NLTK, spaCy и Stanza, значительно упрощает процесс реализации лемматизации, предоставляя высокоуровневые API.

В современных NLP-проектах чаще всего используются **spaCy** и **Stanza** благодаря их высокой точности, производительности, поддержке множества языков и интеграции в полный пайплайн обработки текста. Трансформерные модели представляют собой передовой край исследований и предлагают наилучшее качество, но требуют больших ресурсов и часто дополнительного дообучения для оптимальной производительности на конкретных задачах. Глубокое понимание каждого из этих методов позволит вам выбрать наиболее подходящий инструмент для ваших конкретных задач в области обработки естественного языка.

## **§7.** Дополнительные возможности

### **1. Нормализация текста: Приведение к Единообразию**

**Нормализация текста** — это процесс приведения различных форм одного и того же слова или выражения к единой, стандартной форме. Это помогает уменьшить количество уникальных токенов и повысить согласованность данных, что упрощает их дальнейшую обработку моделями.

**1.1. Приведение к нижнему регистру (Lowercasing)**

Приведение всех букв в тексте к нижнему регистру является одним из базовых шагов нормализации. Например, "Привет, Мир!" преобразуется в "привет, мир!".

Преимущества этого подхода включают:

* Уменьшение размера словаря, поскольку "Apple" (компания) и "apple" (фрукт) будут рассматриваться как одно и то же слово, если контекст не позволяет их различить или это различие не критично для задачи.

Однако существуют и ограничения:

* Может произойти потеря важной информации, например, о начале предложения или о том, что слово является именем собственным. Для задач распознавания именованных сущностей (NER) или синтаксического анализа нижний регистр может быть нежелателен.

**1.2. Удаление пунктуации и чисел**

На этом этапе происходит удаление знаков препинания (запятые, точки, восклицательные знаки и т.д.) и/или числовых значений из текста, если они не несут существенной смысловой нагрузки для конкретной задачи. Например, "Цена: $100!" может быть преобразована в "Цена 100" (если числа сохраняются) или просто "Цена" (если числа удаляются).

Такая предобработка часто используется для задач классификации текста или тематического моделирования, где пунктуация и числа могут рассматриваться как "шум". Однако важно учитывать, что для анализа тональности эмодзи и восклицательные знаки могут быть очень важны, а для извлечения информации числа (например, даты, суммы) критически важны.

**1.3. Обработка эмодзи, сленга и аббревиатур**

Обработка этих элементов текста требует особого внимания, поскольку они могут нести важную смысловую или эмоциональную нагрузку.

* **Эмодзи**:
  + **Удаление**: Самый простой, но часто нежелательный подход, так как эмодзи могут нести сильную эмоциональную окраску.
  + **Замена на текстовые описания**: Например, ":)" может быть заменено на "счастливое лицо", а "😂" — на "лицо со слезами радости". Это позволяет сохранить семантику эмодзи.
  + **Рассмотрение как отдельных токенов**: Если модель способна работать с большим словарем, эмодзи могут быть включены как уникальные токены.
* **Сленг и аббревиатуры**:
  + **Замена на полные формы**: Например, "имхо" можно заменить на "по моему скромному мнению", "т.к." — на "так как", а "lol" — на "laughing out loud". Это часто требует использования заранее определенных словарей или правил.
  + **Сохранение как есть**: Если сленг и аббревиатуры распространены в вашем корпусе и важны для понимания, их можно оставить. Однако это увеличит размер словаря и может потребовать больше данных для обучения модели.

Постоянное появление новых сленговых выражений и аббревиатур требует регулярного обновления используемых словарей и правил.

**3. Удаление стоп-слов**

**Стоп-слова (stop words)** — это часто встречающиеся слова в языке, которые обычно не несут существенной смысловой нагрузки и могут быть удалены из текста для уменьшения шума и размера данных. Примеры для русского языка включают "и", "в", "на", "он", "она", "это", "быть".

Удаление стоп-слов применяется для:

* **Уменьшения размерности**: Сокращает количество токенов, что может ускорить обучение некоторых моделей.
* **Повышения релевантности**: В задачах поиска или тематического моделирования удаление стоп-слов может помочь сосредоточиться на более значимых словах.
* **Улучшения производительности**: Для некоторых моделей (например, TF-IDF) стоп-слова могут искажать важность других слов.

Однако существуют и ограничения:

* **Потеря контекста**: В некоторых задачах (например, анализ тональности, машинный перевод, вопросно-ответные системы) стоп-слова могут быть важны для понимания контекста, отрицания или грамматической структуры предложения. Например, фраза "не очень хорошо" без слова "не" изменит тональность.
* **Зависимость от задачи**: Решение об удалении стоп-слов должно быть обдуманным и зависеть от специфики задачи. Для глубоких нейросетевых моделей, таких как трансформеры, удаление стоп-слов часто не требуется, поскольку они способны самостоятельно выявлять их незначимость.

Списки стоп-слов обычно предоставляются библиотеками NLP (например, NLTK, spaCy) или могут быть составлены вручную.

### **2. Закон Ципфа и его значение для анализа частотности слов в текстах**

**Закон Ципфа (Zipf's Law)** — это эмпирический закон, который описывает распределение частотности слов в любом достаточно большом корпусе естественного языка. Он гласит, что частота любого слова обратно пропорциональна его рангу в списке слов, упорядоченных по убыванию частоты.

Математически это выражается формулой: f≈1/rs, где f — частота слова, r — ранг слова (1-е по частоте, 2-е по частоте и т.д.), а s — показатель степени, который для естественных языков обычно близок к 1. Это означает, что самое частотное слово встречается примерно в два раза чаще, чем второе по частоте, в три раза чаще, чем третье, и так далее.

Значение Закона Ципфа для NLP проявляется в следующем:

* **Неравномерное распределение**: Закон Ципфа показывает, что в любом тексте существует небольшое количество очень частотных слов (так называемая "голова" распределения) и огромное количество очень редких слов ("длинный хвост" распределения).
* **Проблема редких слов (OOV)**: Из-за "длинного хвоста" распределения Ципфа практически невозможно собрать корпус, который содержал бы *все* возможные слова. Всегда будут встречаться слова, которые не были замечены в обучающих данных, что представляет собой проблему для моделей.
* **Обоснование предобработки**:
  + **Удаление стоп-слов**: Самые частые слова, находящиеся в "голове" распределения Ципфа, часто являются стоп-словами, которые можно удалить для уменьшения шума.
  + **Subword-токенизация**: Позволяет эффективно обрабатывать редкие слова из "хвоста" распределения, разбивая их на более мелкие, часто встречающиеся подслова.
* **Выбор размера словаря**: Закон Ципфа помогает понять, почему даже очень большие словари не охватывают все возможные слова, и почему важно иметь стратегии для обработки неизвестных слов.

### **3. Практические аспекты работы с текстом**

После теоретического осмысления этапов предобработки текстовых данных, необходимо перейти к практическим навыкам работы с текстом. Этот раздел посвящен конкретным техническим аспектам, которые являются неотъемлемой частью любого проекта в области обработки естественного языка. Мы рассмотрим, как загружать данные, работать с кодировками и эффективно использовать регулярные выражения для очистки и манипуляции текстом.

**Загрузка и первичная обработка текстовых данных**

Работа с текстовыми данными всегда начинается с их загрузки в среду обработки. Текстовые данные могут быть представлены в различных форматах, таких как простые текстовые файлы (.txt), структурированные файлы (.csv, .json, .xml), или храниться в базах данных. Выбор метода загрузки зависит от исходного формата данных.

**Общий процесс загрузки и начальной обработки:**

1. **Загрузка данных**: Исходные текстовые данные могут быть получены из различных источников:
   * **Текстовые файлы**: Для простых текстовых файлов (.txt) используется стандартная функция Python open().
   * **CSV/TSV файлы**: Для табличных данных, где текст может содержаться в одном из столбцов, применяется библиотека pandas.
   * **JSON/XML файлы**: Для иерархически структурированных данных используются соответствующие парсеры или библиотеки.
   * **Базы данных**: Требуется использование специфических драйверов баз данных (например, psycopg2 для PostgreSQL, sqlite3 для SQLite) и SQL-запросов.
2. **Парсинг и извлечение текста**: В случае структурированных форматов данных (например, JSON, XML) может потребоваться извлечение текстового содержимого из определенных полей или тегов.
3. **Первичная очистка**: На этом этапе удаляются элементы, не относящиеся непосредственно к языковому содержанию, но присутствующие в исходных данных. К ним относятся:
   * **HTML-теги**: Если текст получен из веб-страниц.
   * **URL-адреса**: Ссылки на веб-ресурсы.
   * **Специальные символы**: Символы, не являющиеся частью естественного языка (например, символы авторского права, торговых марок).
   * **Лишние пробелы**: Множественные пробелы, табуляции, символы новой строки, которые могут быть заменены на одиночные пробелы.
4. **Нормализация**: После первичной очистки текст приводится к стандартной форме. Этот этап включает:
   * **Приведение к нижнему регистру**: Преобразование всех букв в нижний регистру для унификации слов (например, "Привет" и "привет" становятся идентичными).
   * **Обработка пунктуации**: Удаление или отделение знаков препинания в зависимости от задачи.
5. **Токенизация**: Разбиение очищенного и нормализованного текста на токены (слова, подслова, символы).
6. **Дальнейшая предобработка**: Применение более сложных лингвистических преобразований, таких как стемминг, лемматизация и удаление стоп-слов, в зависимости от требований конкретной задачи.

**Пример загрузки и базовой очистки текстовых данных:**

|  |
| --- |
| import re  import pandas as pd  from bs4 import BeautifulSoup  import nltk  from nltk.corpus import stopwords  from nltk.stem import PorterStemmer, WordNetLemmatizer  from transformers import AutoTokenizer # Для токенизации HuggingFace  import json # Добавлен импорт json для метода load\_json  # --- Загрузка необходимых данных NLTK (выполняется один раз) ---  # Если вы запускаете этот код впервые, NLTK может потребовать загрузки некоторых корпусов.  # Обычно это делается интерактивно или программно.  nltk\_resources = {      'punkt': 'tokenizers/punkt',      'stopwords': 'corpora/stopwords',      'wordnet': 'corpora/wordnet',      'omw-1.4': 'corpora/omw-1.4',      'punkt\_tab':'punkt\_tab'  }  for resource\_name, resource\_path in nltk\_resources.items():      try:          nltk.data.find(resource\_path)      except LookupError: # Используем LookupError для отсутствующих ресурсов NLTK          print(f"Загрузка NLTK ресурса: {resource\_name}...")          nltk.download(resource\_name)      except Exception as e: # Перехватываем любые другие потенциальные ошибки          print(f"Ошибка при проверке/загрузке NLTK ресурса {resource\_name}: {e}")  class TextPreprocessingPipeline:      """      Очень подробный класс для выполнения различных этапов предобработки текстовых данных,      включая загрузку, очистку, нормализацию, токенизацию и дальнейшую обработку.      """      def \_\_init\_\_(self, hf\_tokenizer\_name: str = "bert-base-uncased"):          """          Инициализирует конвейер предобработки текста.          Args:              hf\_tokenizer\_name (str): Имя предварительно обученного токенизатора HuggingFace                                       для использования на этапе токенизации.          """          print(f"Инициализация TextPreprocessingPipeline с токенизатором: {hf\_tokenizer\_name}...")          self.hf\_tokenizer = AutoTokenizer.from\_pretrained(hf\_tokenizer\_name)          self.stemmer = PorterStemmer()          self.lemmatizer = WordNetLemmatizer()          self.stopwords\_english = set(stopwords.words('english'))          self.stopwords\_russian = set(stopwords.words('russian')) # Добавим русские стоп-слова          print("Конвейер предобработки инициализирован.")          print("-" \* 70)      # --- 1. Загрузка и первичная обработка текстовых данных ---      def load\_txt(self, file\_path: str) -> str:          """          Загружает текст из простого текстового файла (.txt).          Args:              file\_path (str): Путь к файлу .txt.          Returns:              str: Содержимое файла в виде одной строки.          """          try:              with open(file\_path, 'r', encoding='utf-8') as f:                  content = f.read()              print(f"Загружен файл TXT: '{file\_path}'")              return content          except FileNotFoundError:              print(f"Ошибка: Файл не найден по пути '{file\_path}'")              return ""          except Exception as e:              print(f"Ошибка при загрузке TXT файла '{file\_path}': {e}")              return ""      def load\_csv(self, file\_path: str, text\_column: str) -> list[str]:          """          Загружает текстовые данные из CSV/TSV файла, извлекая текст из указанного столбца.          Args:              file\_path (str): Путь к файлу .csv или .tsv.              text\_column (str): Имя столбца, содержащего текст.          Returns:              list[str]: Список строк текста из указанного столбца.          """          try:              df = pd.read\_csv(file\_path)              if text\_column not in df.columns:                  print(f"Ошибка: Столбец '{text\_column}' не найден в файле '{file\_path}'")                  return []              text\_data = df[text\_column].astype(str).tolist()              print(f"Загружен файл CSV/TSV: '{file\_path}', извлечен столбец '{text\_column}'")              return text\_data          except FileNotFoundError:              print(f"Ошибка: Файл не найден по пути '{file\_path}'")              return []          except Exception as e:              print(f"Ошибка при загрузке CSV/TSV файла '{file\_path}': {e}")              return []      def load\_json(self, file\_path: str, text\_key: str) -> list[str]:          """          Загружает текстовые данные из JSON файла, извлекая текст по указанному ключу.          Поддерживает как один JSON объект, так и список JSON объектов.          Args:              file\_path (str): Путь к файлу .json.              text\_key (str): Ключ, по которому находится текст в JSON объекте.          Returns:              list[str]: Список строк текста, извлеченных из JSON.          """          try:              with open(file\_path, 'r', encoding='utf-8') as f:                  data = json.load(f)              texts = []              if isinstance(data, list):                  for item in data:                      if isinstance(item, dict) and text\_key in item:                          texts.append(str(item[text\_key]))              elif isinstance(data, dict) and text\_key in data:                  texts.append(str(data[text\_key]))              else:                  print(f"Предупреждение: Ключ '{text\_key}' не найден или структура JSON не поддерживается.")                  return []              print(f"Загружен файл JSON: '{file\_path}', извлечен текст по ключу '{text\_key}'")              return texts          except FileNotFoundError:              print(f"Ошибка: Файл не найден по пути '{file\_path}'")              return []          except json.JSONDecodeError as e:              print(f"Ошибка декодирования JSON в файле '{file\_path}': {e}")              return []          except Exception as e:              print(f"Ошибка при загрузке JSON файла '{file\_path}': {e}")              return []      # --- 2. Первичная очистка ---      def remove\_html\_tags(self, text: str) -> str:          """          Удаляет HTML-теги из текста.          Args:              text (str): Входной текст, возможно, содержащий HTML.          Returns:              str: Текст без HTML-тегов.          """          if not isinstance(text, str):              return text # Возвращаем как есть, если это не строка          soup = BeautifulSoup(text, 'html.parser')          return soup.get\_text()      def remove\_urls(self, text: str) -> str:          """          Удаляет URL-адреса из текста.          Args:              text (str): Входной текст.          Returns:              str: Текст без URL-адресов.          """          if not isinstance(text, str):              return text          # Регулярное выражение для поиска URL-адресов          url\_pattern = re.compile(r'https?://\S+|www\.\S+')          return url\_pattern.sub(r'', text)      def remove\_special\_characters(self, text: str) -> str:          """          Удаляет специальные символы, не являющиеся буквами, цифрами или пробелами.          Args:              text (str): Входной текст.          Returns:              str: Текст без специальных символов.          """          if not isinstance(text, str):              return text          # Оставляет только буквы (русские и латинские), цифры и пробелы          # [^а-яА-Яa-zA-Z0-9\s] - любая НЕ русская/латинская буква, НЕ цифра, НЕ пробел          return re.sub(r'[^а-яА-Яa-zA-Z0-9\s]', '', text)      def normalize\_whitespace(self, text: str) -> str:          """          Заменяет множественные пробелы, табуляции и символы новой строки на одиночные пробелы.          Args:              text (str): Входной текст.          Returns:              str: Текст с нормализованными пробелами.          """          if not isinstance(text, str):              return text          text = re.sub(r'\s+', ' ', text) # Заменяет 1 или более пробельных символов на один пробел          return text.strip() # Удаляет пробелы в начале и конце строки      # --- 3. Нормализация ---      def to\_lowercase(self, text: str) -> str:          """          Приводит весь текст к нижнему регистру.          Args:              text (str): Входной текст.          Returns:              str: Текст в нижнем регистре.          """          if not isinstance(text, str):              return text          return text.lower()      def handle\_punctuation(self, text: str, remove: bool = False) -> str:          """          Обрабатывает знаки препинания: либо удаляет их, либо отделяет пробелами.          Args:              text (str): Входной текст.              remove (bool): Если True, знаки препинания удаляются.                             Если False, знаки препинания отделяются пробелами.          Returns:              str: Текст с обработанными знаками препинания.          """          if not isinstance(text, str):              return text          if remove:              # Удаляем все знаки препинания              return re.sub(r'[^\w\s]', '', text)          else:              # Отделяем знаки препинания пробелами              # Добавляем пробел перед и после знака препинания, затем нормализуем пробелы              text = re.sub(r'([.,!?;:()])', r' \1 ', text)              return self.normalize\_whitespace(text)      # --- 4. Токенизация ---      def tokenize\_text(self, text: str, tokenizer\_type: str = 'huggingface') -> list[str]:          """          Токенизирует текст с использованием выбранного метода.          Args:              text (str): Входной текст.              tokenizer\_type (str): Тип токенизатора ('huggingface', 'nltk\_word', 'nltk\_sent').          Returns:              list[str]: Список токенов или предложений.          """          if not isinstance(text, str):              return []          if tokenizer\_type == 'huggingface':              # Используем метод .tokenize() для получения списка строковых токенов HuggingFace              return self.hf\_tokenizer.tokenize(text)          elif tokenizer\_type == 'nltk\_word':              # Используем NLTK для пословной токенизации              return nltk.word\_tokenize(text)          elif tokenizer\_type == 'nltk\_sent':              # Используем NLTK для разбиения на предложения              return nltk.sent\_tokenize(text)          else:              print(f"Предупреждение: Неизвестный тип токенизатора '{tokenizer\_type}'. Токенизация не выполнена.")              return [text] # Возвращаем исходный текст в списке, если токенизация не удалась      # --- 5. Дальнейшая предобработка ---      def remove\_stopwords(self, tokens: list[str], lang: str = 'english') -> list[str]:          """          Удаляет стоп-слова из списка токенов.          Args:              tokens (list[str]): Список токенов.              lang (str): Язык стоп-слов ('english' или 'russian').          Returns:              list[str]: Список токенов без стоп-слов.          """          if not isinstance(tokens, list):              return tokens          if lang == 'english':              return [word for word in tokens if word.lower() not in self.stopwords\_english]          elif lang == 'russian':              return [word for word in tokens if word.lower() not in self.stopwords\_russian]          else:              print(f"Предупреждение: Неизвестный язык стоп-слов '{lang}'. Стоп-слова не удалены.")              return tokens      def apply\_stemming(self, tokens: list[str]) -> list[str]:          """          Применяет стемминг к списку токенов (приводит слова к их основе).          Args:              tokens (list[str]): Список токенов.          Returns:              list[str]: Список токенов после стемминга.          """          if not isinstance(tokens, list):              return tokens          return [self.stemmer.stem(word) for word in tokens]      def apply\_lemmatization(self, tokens: list[str]) -> list[str]:          """          Применяет лемматизацию к списку токенов (приводит слова к их словарной форме).          Args:              tokens (list[str]): Список токенов.          Returns:              list[str]: Список токенов после лемматизации.          """          if not isinstance(tokens, list):              return tokens          return [self.lemmatizer.lemmatize(word) for word in tokens]      # --- Основной метод конвейера ---      def process\_text(self, text: str,                       remove\_html: bool = True,                       remove\_urls: bool = True,                       remove\_special\_chars: bool = True,                       normalize\_ws: bool = True,                       to\_lower: bool = True,                       handle\_punct: str = 'separate', # 'remove', 'separate', 'none'                       do\_tokenize: bool = True,                       tokenizer\_type: str = 'huggingface', # Новый параметр для выбора типа токенизации                       remove\_sw: bool = False,                       sw\_lang: str = 'english',                       do\_stem: bool = False,                       do\_lemmatize: bool = False) -> str | list[str]:          """          Выполняет полный конвейер предобработки текста.          Args:              text (str): Входной текст для обработки.              remove\_html (bool): Удалять ли HTML-теги.              remove\_urls (bool): Удалять ли URL-адреса.              remove\_special\_chars (bool): Удалять ли специальные символы.              normalize\_ws (bool): Нормализовать ли пробелы.              to\_lower (bool): Приводить ли текст к нижнему регистру.              handle\_punct (str): Как обрабатывать пунктуацию:                                  'remove' - удалить,                                  'separate' - отделить пробелами,                                  'none' - не обрабатывать.              do\_tokenize (bool): Выполнять ли токенизацию. Если False, возвращается строка.              tokenizer\_type (str): Тип токенизатора для использования, если do\_tokenize=True.                                    Варианты: 'huggingface', 'nltk\_word', 'nltk\_sent'.              remove\_sw (bool): Удалять ли стоп-слова (требует do\_tokenize=True).              sw\_lang (str): Язык стоп-слов ('english' или 'russian').              do\_stem (bool): Выполнять ли стемминг (требует do\_tokenize=True).              do\_lemmatize (bool): Выполнять ли лемматизацию (требует do\_tokenize=True).          Returns:              str | list[str]: Обработанный текст (строка) или список токенов (если do\_tokenize=True).          """          processed\_text = text          print(f"\nНачало обработки текста: '{processed\_text[:100]}...'")          # 1. Первичная очистка          if remove\_html:              processed\_text = self.remove\_html\_tags(processed\_text)              print(" - HTML-теги удалены.")          if remove\_urls:              processed\_text = self.remove\_urls(processed\_text)              print(" - URL-адреса удалены.")          if remove\_special\_chars:              processed\_text = self.remove\_special\_characters(processed\_text)              print(" - Специальные символы удалены.")          if normalize\_ws:              processed\_text = self.normalize\_whitespace(processed\_text)              print(" - Пробелы нормализованы.")          # 2. Нормализация          if to\_lower:              processed\_text = self.to\_lowercase(processed\_text)              print(" - Текст приведен к нижнему регистру.")          if handle\_punct != 'none':              processed\_text = self.handle\_punctuation(processed\_text, remove=(handle\_punct == 'remove'))              print(f" - Пунктуация обработана ({handle\_punct}).")          # 3. Токенизация          if do\_tokenize:              tokens = self.tokenize\_text(processed\_text, tokenizer\_type=tokenizer\_type) # Передаем тип токенизатора              print(f" - Текст токенизирован ({tokenizer\_type}). Токенов: {len(tokens)}")              # 4. Дальнейшая предобработка              # Примечание: Стемминг и лемматизация применяются только к пословным токенам.              # Если tokenizer\_type был 'nltk\_sent', то apply\_stemming/lemmatization              # будут работать с предложениями как с "токенами", что может быть нежелательно.              # Для корректной работы со стеммингом/лемматизацией после NLTK-токенизации              # на уровне предложений, потребуется дополнительная пословная токенизация внутри этих методов.              if remove\_sw:                  tokens = self.remove\_stopwords(tokens, lang=sw\_lang)                  print(f" - Стоп-слова удалены ({sw\_lang}).")              if do\_stem:                  if tokenizer\_type == 'nltk\_sent':                      print("Предупреждение: Стемминг применяется к предложениям, а не к словам. Рассмотрите пословную токенизацию.")                  tokens = self.apply\_stemming(tokens)                  print(" - Применен стемминг.")              if do\_lemmatize:                  if tokenizer\_type == 'nltk\_sent':                      print("Предупреждение: Лемматизация применяется к предложениям, а не к словам. Рассмотрите пословную токенизацию.")                  tokens = self.apply\_lemmatization(tokens)                  print(" - Применена лемматизация.")              print(f"Обработка завершена. Возвращены токены.")              return tokens          else:              print(f"Обработка завершена. Возвращена строка.")              return processed\_text  # --- Пример использования класса ---  if \_\_name\_\_ == "\_\_main\_\_":      # Создаем экземпляр конвейера предобработки      # Можно указать любой токенизатор HuggingFace, например, "DeepPavlov/rubert-base-cased" для русского      pipeline = TextPreprocessingPipeline(hf\_tokenizer\_name="bert-base-uncased")      # --- Демонстрация загрузки данных (создадим фиктивные файлы) ---      import os      # import json # Уже импортирован выше      # Создаем фиктивный TXT файл      with open("sample.txt", "w", encoding="utf-8") as f:          f.write("Это простой текстовый файл. Он содержит несколько строк.\n"                  "Иногда здесь могут быть <p>HTML-теги</p> и URL-адреса: https://example.com/page?id=123.\n"                  "А также @специальные #символы и    лишние пробелы. Привет, Мир!")      # Создаем фиктивный CSV файл      csv\_data = {          'id': [1, 2],          'text\_content': [              "Первая строка в CSV. <a href='test.html'>Ссылка</a>.",              "Вторая строка с текстом и УРЛ: www.google.com. И еще @символ."          ],          'category': ['A', 'B']      }      pd.DataFrame(csv\_data).to\_csv("sample.csv", index=False)      # Создаем фиктивный JSON файл      json\_data = [          {"doc\_id": "d1", "content": "JSON документ 1. Ссылка: http://json.org"},          {"doc\_id": "d2", "content": "JSON документ 2. <br>Новая строка."}      ]      with open("sample.json", "w", encoding="utf-8") as f:          json.dump(json\_data, f, ensure\_ascii=False, indent=4)      print("\n--- Демонстрация загрузки данных ---")      txt\_content = pipeline.load\_txt("sample.txt")      print(f"Загруженный TXT (первые 100 символов): '{txt\_content[:100]}'\n")      csv\_contents = pipeline.load\_csv("sample.csv", "text\_content")      print(f"Загруженный CSV (первая строка): '{csv\_contents[0]}'\n")      json\_contents = pipeline.load\_json("sample.json", "content")      print(f"Загруженный JSON (первая строка): '{json\_contents[0]}'\n")      # --- Демонстрация полного конвейера предобработки ---      print("\n" + "=" \* 70)      print("--- Демонстрация полного конвейера предобработки ---")      print("=" \* 70)      sample\_text = """      <p>Это <b>пример</b> текста с <a href="http://example.com">HTML-тегами</a>,      URL-адресами (https://www.google.com), @специальными символами,      и    множественными пробелами.      <br>      Привет, Мир! Как дела?! Это очень-очень-очень длинное предложение,      которое мы хотим обработать.      </p>      """      print(f"\nИсходный текст для обработки:\n'{sample\_text}'")      # Пример 1: Базовая очистка и токенизация (HuggingFace)      print("\n--- Пример 1: Базовая очистка и токенизация (HuggingFace) ---")      processed\_tokens\_1 = pipeline.process\_text(          sample\_text,          remove\_html=True,          remove\_urls=True,          remove\_special\_chars=True,          normalize\_ws=True,          to\_lower=True,          handle\_punct='separate', # Отделяем пунктуацию          do\_tokenize=True,          tokenizer\_type='huggingface', # Явно указываем HuggingFace          remove\_sw=False,          do\_stem=False,          do\_lemmatize=False      )      print(f"Результат (токены): {processed\_tokens\_1}")      # Декодирование возможно только если токенизация была HuggingFace      print(f"Декодированный результат: {pipeline.hf\_tokenizer.decode(pipeline.hf\_tokenizer.convert\_tokens\_to\_ids(processed\_tokens\_1))}")      # Пример 2: Удаление стоп-слов и стемминг (английский, NLTK пословная токенизация)      print("\n--- Пример 2: Удаление стоп-слов и стемминг (английский, NLTK пословная токенизация) ---")      sample\_text\_en = "The quick brown foxes are jumping over the lazy dogs. This is another sentence."      processed\_tokens\_2 = pipeline.process\_text(          sample\_text\_en,          remove\_html=False,          remove\_urls=False,          remove\_special\_chars=True,          normalize\_ws=True,          to\_lower=True,          handle\_punct='remove', # Удаляем пунктуацию          do\_tokenize=True,          tokenizer\_type='nltk\_word', # Используем NLTK пословную токенизацию          remove\_sw=True,          sw\_lang='english',          do\_stem=True,          do\_lemmatize=False      )      print(f"Исходный текст: '{sample\_text\_en}'")      print(f"Результат (токены): {processed\_tokens\_2}")      # Пример 3: Лемматизация (английский, NLTK пословная токенизация)      print("\n--- Пример 3: Лемматизация (английский, NLTK пословная токенизация) ---")      processed\_tokens\_3 = pipeline.process\_text(          "The quick brown foxes are jumping over the lazy dogs. This is another sentence.",          remove\_html=False,          remove\_urls=False,          remove\_special\_chars=True,          normalize\_ws=True,          to\_lower=True,          handle\_punct='remove',          do\_tokenize=True,          tokenizer\_type='nltk\_word', # Используем NLTK пословную токенизацию          remove\_sw=True,          sw\_lang='english',          do\_stem=False,          do\_lemmatize=True      )      print(f"Результат (токены): {processed\_tokens\_3}")      # Пример 4: Разбиение на предложения (NLTK)      print("\n--- Пример 4: Разбиение на предложения (NLTK) ---")      text\_for\_sentences = "Привет, Мир! Как дела?! Это очень длинное предложение. А это второе."      processed\_sentences = pipeline.process\_text(          text\_for\_sentences,          remove\_html=False,          remove\_urls=False,          remove\_special\_chars=True,          normalize\_ws=True,          to\_lower=False, # Обычно предложения не приводят к нижнему регистру          handle\_punct='none', # Сохраняем пунктуацию для разбиения на предложения          do\_tokenize=True,          tokenizer\_type='nltk\_sent', # Используем NLTK разбиение на предложения          remove\_sw=False,          do\_stem=False,          do\_lemmatize=False      )      print(f"Исходный текст: '{text\_for\_sentences}'")      print(f"Результат (предложения): {processed\_sentences}")      # Пример 5: Обработка русского текста (с русскими стоп-словами, HuggingFace токенизация)      print("\n--- Пример 5: Обработка русского текста (с русскими стоп-словами, HuggingFace токенизация) ---")      # Для русского языка лучше использовать специализированный токенизатор HuggingFace      # pipeline\_ru = TextPreprocessingPipeline(hf\_tokenizer\_name="DeepPavlov/rubert-base-cased") # Если установлен      # Или используем текущий, но помним, что он обучен на английском      russian\_text = "Привет, это очень хороший пример русского текста для предобработки. Как дела сегодня?"      processed\_tokens\_ru = pipeline.process\_text(          russian\_text,          remove\_html=False,          remove\_urls=False,          remove\_special\_chars=True,          normalize\_ws=True,          to\_lower=True,          handle\_punct='remove',          do\_tokenize=True,          tokenizer\_type='huggingface', # Используем HuggingFace токенизацию          remove\_sw=True,          sw\_lang='russian', # Используем русские стоп-слова          do\_stem=False,          do\_lemmatize=False      )      print(f"Исходный русский текст: '{russian\_text}'")      print(f"Результат (токены): {processed\_tokens\_ru}")      # Декодирование возможно только если токенизация была HuggingFace      print(f"Декодированный результат: {pipeline.hf\_tokenizer.decode(pipeline.hf\_tokenizer.convert\_tokens\_to\_ids(processed\_tokens\_ru))}")      # Очистка фиктивных файлов      os.remove("sample.txt")      os.remove("sample.csv")      os.remove("sample.json")      print("\nФиктивные файлы удалены.") |

Результат:

Загрузка NLTK ресурса: wordnet...

Загрузка NLTK ресурса: omw-1.4...

Загрузка NLTK ресурса: punkt\_tab...

[nltk\_data] Downloading package wordnet to /root/nltk\_data...

[nltk\_data] Package wordnet is already up-to-date!

[nltk\_data] Downloading package omw-1.4 to /root/nltk\_data...

[nltk\_data] Package omw-1.4 is already up-to-date!

[nltk\_data] Downloading package punkt\_tab to /root/nltk\_data...

[nltk\_data] Unzipping tokenizers/punkt\_tab.zip.

Инициализация TextPreprocessingPipeline с токенизатором: bert-base-uncased...

Конвейер предобработки инициализирован.

----------------------------------------------------------------------

--- Демонстрация загрузки данных ---

Загружен файл TXT: 'sample.txt'

Загруженный TXT (первые 100 символов): 'Это простой текстовый файл. Он содержит несколько строк.

Иногда здесь могут быть <p>HTML-теги</p> и '

Загружен файл CSV/TSV: 'sample.csv', извлечен столбец 'text\_content'

Загруженный CSV (первая строка): 'Первая строка в CSV. <a href='test.html'>Ссылка</a>.'

Загружен файл JSON: 'sample.json', извлечен текст по ключу 'content'

Загруженный JSON (первая строка): 'JSON документ 1. Ссылка: [http://json.org](http://json.org/)'

======================================================================

--- Демонстрация полного конвейера предобработки ---

======================================================================

Исходный текст для обработки:

'

<p>Это <b>пример</b> текста с <a href="[http://example.com](http://example.com/)">HTML-тегами</a>,

URL-адресами ([https://www.google.com](https://www.google.com/)), @специальными символами,

и множественными пробелами.

<br>

Привет, Мир! Как дела?! Это очень-очень-очень длинное предложение,

которое мы хотим обработать.

</p>

'

--- Пример 1: Базовая очистка и токенизация (HuggingFace) ---

Начало обработки текста: '

<p>Это <b>пример</b> текста с <a href="[http://example.com](http://example.com/)">HTML-тегами</a>,

URL-адресами (h...'

- HTML-теги удалены.

- URL-адреса удалены.

- Специальные символы удалены.

- Пробелы нормализованы.

- Текст приведен к нижнему регистру.

- Пунктуация обработана (separate).

- Текст токенизирован (huggingface). Токенов: 154

Обработка завершена. Возвращены токены.

Результат (токены): ['э', '##т', '##о', 'п', '##р', '##и', '##м', '##е', '##р', 'т', '##е', '##к', '##с', '##т', '##а', 'с', 'html', '##т', '##е', '##г', '##а', '##м', '##и', 'ur', '##l', '##а', '##д', '##р', '##е', '##с', '##а', '##м', '##и', 'с', '##п', '##е', '##ц', '##и', '##а', '##л', '##ь', '##н', '##ы', '##м', '##и', 'с', '##и', '##м', '##в', '##о', '##л', '##а', '##м', '##и', 'и', 'м', '##н', '##о', '##ж', '##е', '##с', '##т', '##в', '##е', '##н', '##н', '##ы', '##м', '##и', 'п', '##р', '##о', '##б', '##е', '##л', '##а', '##м', '##и', 'п', '##р', '##и', '##в', '##е', '##т', 'м', '##и', '##р', 'к', '##а', '##к', 'д', '##е', '##л', '##а', 'э', '##т', '##о', 'о', '##ч', '##е', '##н', '##ь', '##о', '##ч', '##е', '##н', '##ь', '##о', '##ч', '##е', '##н', '##ь', 'д', '##л', '##и', '##н', '##н', '##о', '##е', 'п', '##р', '##е', '##д', '##л', '##о', '##ж', '##е', '##н', '##и', '##е', 'к', '##о', '##т', '##о', '##р', '##о', '##е', 'м', '##ы', 'х', '##о', '##т', '##и', '##м', 'о', '##б', '##р', '##а', '##б', '##о', '##т', '##а', '##т', '##ь']

Декодированный результат: это пример текста с htmlтегами urlадресами специальными символами и множественными пробелами привет мир как дела это оченьоченьочень длинное предложение которое мы хотим обработать

--- Пример 2: Удаление стоп-слов и стемминг (английский, NLTK пословная токенизация) ---

Начало обработки текста: 'The quick brown foxes are jumping over the lazy dogs. This is another sentence....'

- Специальные символы удалены.

- Пробелы нормализованы.

- Текст приведен к нижнему регистру.

- Пунктуация обработана (remove).

- Текст токенизирован (nltk\_word). Токенов: 14

- Стоп-слова удалены (english).

- Применен стемминг.

Обработка завершена. Возвращены токены.

Исходный текст: 'The quick brown foxes are jumping over the lazy dogs. This is another sentence.'

Результат (токены): ['quick', 'brown', 'fox', 'jump', 'lazi', 'dog', 'anoth', 'sentenc']

--- Пример 3: Лемматизация (английский, NLTK пословная токенизация) ---

Начало обработки текста: 'The quick brown foxes are jumping over the lazy dogs. This is another sentence....'

- Специальные символы удалены.

- Пробелы нормализованы.

- Текст приведен к нижнему регистру.

- Пунктуация обработана (remove).

- Текст токенизирован (nltk\_word). Токенов: 14

- Стоп-слова удалены (english).

- Применена лемматизация.

Обработка завершена. Возвращены токены.

Результат (токены): ['quick', 'brown', 'fox', 'jumping', 'lazy', 'dog', 'another', 'sentence']

--- Пример 4: Разбиение на предложения (NLTK) ---

Начало обработки текста: 'Привет, Мир! Как дела?! Это очень длинное предложение. А это второе....'

- Специальные символы удалены.

- Пробелы нормализованы.

- Текст токенизирован (nltk\_sent). Токенов: 1

Обработка завершена. Возвращены токены.

Исходный текст: 'Привет, Мир! Как дела?! Это очень длинное предложение. А это второе.'

Результат (предложения): ['Привет Мир Как дела Это очень длинное предложение А это второе']

--- Пример 5: Обработка русского текста (с русскими стоп-словами, HuggingFace токенизация) ---

Начало обработки текста: 'Привет, это очень хороший пример русского текста для предобработки. Как дела сегодня?...'

- Специальные символы удалены.

- Пробелы нормализованы.

- Текст приведен к нижнему регистру.

- Пунктуация обработана (remove).

- Текст токенизирован (huggingface). Токенов: 70

- Стоп-слова удалены (russian).

Обработка завершена. Возвращены токены.

Исходный русский текст: 'Привет, это очень хороший пример русского текста для предобработки. Как дела сегодня?'

Результат (токены): ['п', '##р', '##и', '##в', '##е', '##т', 'э', '##т', '##о', '##ч', '##е', '##н', '##ь', 'х', '##о', '##р', '##о', '##ш', '##ии', 'п', '##р', '##и', '##м', '##е', '##р', 'р', '##у', '##с', '##с', '##к', '##о', '##г', '##о', 'т', '##е', '##к', '##с', '##т', '##а', 'д', '##л', '##я', 'п', '##р', '##е', '##д', '##о', '##б', '##р', '##а', '##б', '##о', '##т', '##к', '##и', '##а', '##к', 'д', '##е', '##л', '##а', '##е', '##г', '##о', '##д', '##н', '##я']

Декодированный результат: привет эточень хорошии пример русского текста для предобработкиак делаегодня

Фиктивные файлы удалены.

### **4. Работа с кодировками (UTF-8)**

**Кодировка текста** определяет, каким образом символы человеческого языка преобразуются в последовательности байтов для хранения в памяти компьютера или на диске. Неправильное обращение с кодировками является одной из наиболее частых причин ошибок при работе с текстовыми данными в NLP.

**4.1. Проблема кодировок**

Исторически сложилось множество различных кодировок (например, CP1251 для кириллицы, ISO-8859-1 для западноевропейских языков), каждая из которых поддерживала свой ограниченный набор символов. Если файл был сохранен в одной кодировке, а попытка чтения осуществляется с интерпретацией байтов в другой кодировке, могут возникнуть следующие проблемы:

* **Ошибки UnicodeDecodeError**: Python не может преобразовать последовательность байтов в корректные символы Unicode.
* **Искажение символов ("кракозябры")**: Неправильно интерпретированные символы отображаются как нечитаемые последовательности.

**4.2. Универсальное решение: UTF-8**

**UTF-8 (Unicode Transformation Format - 8-bit)** — это наиболее распространенная и рекомендуемая кодировка для работы с текстом, особенно с многоязычным контентом. Она является частью стандарта Unicode, который стремится охватить все символы всех письменных языков мира.

**Преимущества UTF-8**:

* + **Совместимость**: Поддерживает все символы Unicode, что делает ее универсальной для любых языков.
  + **Эффективность**: Для ASCII-символов (латиница, цифры, базовая пунктуация) UTF-8 использует всего один байт, что обеспечивает эффективность для англоязычного текста. Для других символов (например, кириллицы) используется от двух до четырех байтов.
  + **Обратная совместимость**: ASCII-текст является валидным UTF-8 текстом.

**Практические рекомендации по работе с кодировками в Python:**

**Всегда явно указывайте кодировку**: При чтении и записи файлов в Python всегда используйте параметр encoding='utf-8'.

|  |
| --- |
| # 1. ✅ Создание исходного файла с русским текстом в UTF-8  input\_text = "Пример исходного текста на русском языке."  with open('my\_document.txt', 'w', encoding='utf-8') as f:      f.write(input\_text)  print("📄 Файл 'my\_document.txt' успешно создан в кодировке UTF-8.\n")  # 2. 📥 Чтение файла с попыткой сначала в UTF-8, затем в CP1251  file\_path = 'my\_document.txt'  try:      with open(file\_path, 'r', encoding='utf-8') as f:          text\_data = f.read()          print("✅ Файл успешно прочитан в UTF-8.")  except UnicodeDecodeError:      print("⚠️ Не удалось прочитать как UTF-8. Пробуем CP1251...")      try:          with open(file\_path, 'r', encoding='cp1251') as f:              text\_data = f.read()              print("✅ Файл успешно прочитан в CP1251.")      except Exception as e:          print(f"❌ Ошибка: не удалось прочитать файл ни в UTF-8, ни в CP1251.\n{e}")          text\_data = None  except FileNotFoundError:      print("❌ Ошибка: файл не найден.")      text\_data = None  # 3. 📤 Запись обработанного текста в новый файл в UTF-8, если чтение прошло успешно  if text\_data:      processed\_text = "Это обработанный текст на русском языке."      with open('processed\_output.txt', 'w', encoding='utf-8') as f:          f.write(processed\_text)      print("💾 Обработанный текст успешно записан в файл 'processed\_output.txt' (UTF-8).") |

**Определение кодировки неизвестного файла**: Если кодировка файла неизвестна, можно использовать библиотеки, такие как chardet, которая автоматически определяет кодировку на основе статистического анализа байтов.

|  |
| --- |
| import chardet  # 1. ✅ Создание файла с неизвестной (для нас) кодировкой CP1251  text\_cp1251 = "Это текст на русском языке с кодировкой CP1251."  file\_path = 'unknown\_encoding\_file.txt'  with open(file\_path, 'w', encoding='cp1251') as f:      f.write(text\_cp1251)  print(f"📄 Файл '{file\_path}' успешно создан в кодировке CP1251.\n")  # 2. 📥 Определение кодировки с помощью chardet  with open(file\_path, 'rb') as f:      raw\_data = f.read()  result = chardet.detect(raw\_data)  encoding = result['encoding']  confidence = result['confidence']  print(f"🔍 Определённая кодировка: {encoding} (уверенность: {confidence:.2f})")  # 3. ✅ Попытка декодирования текста с определённой кодировкой  try:      decoded\_text = raw\_data.decode(encoding)      print("✅ Файл успешно декодирован.")      print(f"\n📘 Содержимое файла:\n{decoded\_text}")  except Exception as e:      print(f"❌ Ошибка при декодировании файла с кодировкой {encoding}: {e}") |

### **5. Использование регулярных выражений для обработки и очистки текста**

Регулярные выражения (Regular Expressions, RegEx или Regex) — это мощный, но компактный язык для описания шаблонов текстовых строк. Они незаменимы для поиска, извлечения, замены и очистки текста на различных этапах предобработки. В Python для работы с регулярными выражениями используется встроенный модуль re.

**5.1. Основные операции с модулем re в Python**

Для работы с регулярными выражениями в Python предназначен стандартный модуль re, который предоставляет набор функций для выполнения основных операций поиска и обработки текста. Эти функции являются основным инструментарием для реализации различных задач текстовой предобработки, начиная от простого поиска и заканчивая сложными операциями замены и извлечения данных.

* re.search(pattern, string) : Осуществляет поиск первого вхождения шаблона (pattern) в строке (string). Возвращает объект совпадения (Match object) при обнаружении или None в противном случае.
* re.match(pattern, string) : Ищет шаблон исключительно в начале строки. Возвращает объект совпадения или None.
* re.findall(pattern, string) : Находит все непересекающиеся вхождения шаблона в строке и возвращает их в виде списка.
* re.sub(pattern, repl, string) : Заменяет все вхождения шаблона (pattern) на строку замены (repl) в строке (string).

Каждая из этих функций решает определенную задачу и может быть использована в зависимости от конкретных требований к обработке текста. Правильный выбор функции позволяет эффективно решать задачи очистки и предварительной обработки текстовых данных.

**5.2. Ключевые элементы синтаксиса регулярных выражений**

Для эффективной работы с регулярными выражениями необходимо понимать основные синтаксические конструкции, которые позволяют создавать сложные шаблоны поиска. Эти элементы включают в себя как простые символы, так и специальные метасимволы, каждый из которых имеет свое уникальное назначение. Ниже представлена таблица с основными элементами синтаксиса регулярных выражений, их значением, примерами использования и кратким описанием.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| . | Соответствует любому символу (кроме символа новой строки). | a.bнайдет "acb", "aab", "axb". | Универсальный символ |
| ^ | Начало строки. | ^abcнайдет "abc" только в начале строки. | Привязка к началу |
| $ | Конец строки. | abc$найдет "abc" только в конце строки. | Привязка к концу |
| \* | Ноль или более повторений предыдущего символа/группы. | ab\*cнайдет "ac", "abc", "abbbc". | Жадный квантификатор |
| + | Одно или более повторений предыдущего символа/группы. | ab+cнайдет "abc", "abbbc", но не "ac". | Требует хотя бы одно совпадение |
| ? | Ноль или одно повторение предыдущего символа/группы. Также используется для нежадного поиска. | ab?cнайдет "ac", "abc"..\*?— нежадный поиск. | Делает элемент необязательным |
| {n} | Ровно n повторений предыдущего символа/группы. | a{3}bнайдет "aaab". | Точное количество повторений |
| {n,} | n или более повторений предыдущего символа/группы. | a{2,}bнайдет "aab", "aaab", "aaaab". | Минимум n повторений |
| {n,m} | От n до m повторений предыдущего символа/группы. | a{1,3}bнайдет "ab", "aab", "aaab". | Диапазон повторений |
| [] | Набор символов. Соответствует любому одному символу из набора. | [abc]найдет "a", "b" или "c".[0-9]— любая цифра. | Перечисление допустимых символов |
| [^] | Исключающий набор символов. | [^abc]найдет любой символ, кроме "a", "b" или "c". | Отрицание набора |
| | | ИЛИ. Соответствует одному из шаблонов. | cat|dogнайдет "cat" или "dog". | Альтернатива |
| () | Группировка. Используется для создания подвыражений. | (ab)+найдет "ab", "abab", "ababab". | Позволяет применять квантификаторы |
| \ | Экранирование. Используется для поиска специальных символов как обычных. | \.найдет точку.\\— обратный слэш. | Отменяет специальное значение |
| \d | Соответствует любой цифре (эквивалентно[0-9]). | \d+найдет одно или более чисел. | Удобно для поиска чисел |
| \D | Соответствует любому символу, который НЕ является цифрой. | \D+найдет последовательность не-цифр. | Отрицание\d |
| \w | Соответствует любому буквенно-цифровому символу или нижнему подчеркиванию ([a-zA-Z0-9\_]). | \w+найдет слово. | Для поиска слов |
| \W | Соответствует любому символу, который НЕ является буквенно-цифровым или подчеркиванием. | \W+найдет не-словесные символы. | Отрицание\w |
| \s | Соответствует любому пробельному символу (пробел, табуляция, новая строка). | \s+найдет один или более пробельных символов. | Для работы с пробелами |
| \S | Соответствует любому символу, который НЕ является пробельным. | \S+найдет непрерывную последовательность не-пробельных символов. | Отрицание\s |
| \b | Граница слова. | \bcat\bнайдет слово "cat", но не "category". | Уточняет позицию слова |
| \B | Не граница слова. | \Bcat\Bнайдет "cat" в "category", но не как отдельное слово. | Отрицание\b |

Понимание этих элементов позволяет создавать эффективные шаблоны для решения различных задач по обработке текста. На практике эти конструкции часто комбинируются для достижения сложных результатов поиска и замены.

**Примеры использования регулярных выражений для очистки текста:**

|  |
| --- |
| import re  text = """  <p>Привет, мир! Это <strong>пример</strong> текста с <a href="http://example.com">ссылкой</a>.  Мой email: user@domain.com. Телефон: +7 (999) 123-45-67.  Дата: 23.07.2025. Цена: $123.45.  Много   лишних    пробелов.  </p>  """  print("Оригинальный текст:\n", text)  # 1. Удаление HTML-тегов  # <.\*?> - нежадный поиск любого текста между < и >  cleaned\_text = re.sub(r'<.\*?>', '', text)  print("\n1. Без HTML-тегов:\n", cleaned\_text)  # 2. Удаление URL-адресов  # http\S+ - http, за которым следуют один или более не-пробельных символов  # www\S+ - www, за которым следуют один или более не-пробельных символов  cleaned\_text = re.sub(r'http\S+|www\S+', '', cleaned\_text)  print("\n2. Без URL-адресов:\n", cleaned\_text)  # 3. Удаление email-адресов  # \b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b - общий паттерн для email  cleaned\_text = re.sub(r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', '', cleaned\_text)  print("\n3. Без email-адресов:\n", cleaned\_text)  # 4. Удаление чисел и символов, кроме букв и пробелов  # [^a-zа-я\s] - любой символ, который НЕ является латинской буквой, кириллической буквой или пробелом  cleaned\_text = re.sub(r'[^a-zа-я\s]', '', cleaned\_text.lower()) # Приводим к нижнему регистру  print("\n4. Только буквы и пробелы (нижний регистр):\n", cleaned\_text)  # 5. Удаление лишних пробелов и обрезка по краям  # \s+ - один или более пробельных символов  final\_cleaned\_text = re.sub(r'\s+', ' ', cleaned\_text).strip()  print("\n5. Финальный очищенный текст:\n", final\_cleaned\_text)  # Дополнительный пример: Извлечение телефонных номеров  phone\_text = "Мой номер: +1 (555) 123-4567, или 8-800-555-35-35. А также 123 456 7890."  # Паттерн для различных форматов телефонных номеров  phone\_numbers = re.findall(r'\+?\d{1,3}[-.\s]?\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{2}[-.\s]?\d{2}|\d{1,3}[-.\s]?\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}', phone\_text)  print(f"\nНайденные телефонные номера: {phone\_numbers}") |

Результат:

Оригинальный текст:

<p>Привет, мир! Это <strong>пример</strong> текста с <a href="[http://example.com](http://example.com/)">ссылкой</a>.

Мой email: [user@domain.com](mailto:user@domain.com). Телефон: +7 (999) 123-45-67.

Дата: 23.07.2025. Цена: $123.45.

Много лишних пробелов.

</p>

1. Без HTML-тегов:

Привет, мир! Это пример текста с ссылкой.

Мой email: [user@domain.com](mailto:user@domain.com). Телефон: +7 (999) 123-45-67.

Дата: 23.07.2025. Цена: $123.45.

Много лишних пробелов.

2. Без URL-адресов:

Привет, мир! Это пример текста с ссылкой.

Мой email: [user@domain.com](mailto:user@domain.com). Телефон: +7 (999) 123-45-67.

Дата: 23.07.2025. Цена: $123.45.

Много лишних пробелов.

3. Без email-адресов:

Привет, мир! Это пример текста с ссылкой.

Мой email: . Телефон: +7 (999) 123-45-67.

Дата: 23.07.2025. Цена: $123.45.

Много лишних пробелов.

4. Только буквы и пробелы (нижний регистр):

привет мир это пример текста с ссылкой

мой email телефон

дата цена

много лишних пробелов

5. Финальный очищенный текст:

привет мир это пример текста с ссылкой мой email телефон дата цена много лишних пробелов

Найденные телефонные номера: ['+1 (555) 123-4567', '8-800-555-35-35']

Таким образом, регулярные выражения являются мощным инструментом, но их синтаксис может быть сложным для освоения. Практика и использование онлайн-инструментов для тестирования регулярных выражений (например, regex101.com) значительно ускоряют процесс обучения и отладки.