МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

АВТОМАТИЧЕСКОЕ ФОРМИРОВАНИЕ ЗАДАНИЙ ПО ГРАММАТИКЕ АНГЛИЙСКОГО ЯЗЫКА

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
А. А. Мезги

Научный руководитель:
кандидат физико-математических наук, доцент В. А. Нестеренко

Ростов-на-Дону
2020

Содержание

Перечень условных обозначений

NLP – Natural Language Processing
SVM – Support Vector Machine
POS – Part Of Speech
NER – Named Entity Recognition
SRL – Semantic Role Labeling
OHE – One-Hot Encoding
BOW – Bag-Of-Words
CBOW – Continuous Bag-Of-Words
SVD – Singular Value Decomposition
CNN – Convolutional Neural Network

Введение

Принято считать, что история развития обработки естественного языка берёт своё начало в 1950-х годах, когда Алан Тьюринг опубликовал свою работу «Вычислительные машины и разум» [1], где был представлен тест Тьюринга.

В настоящий момент в рамках обработки естественного языка (англ. Natural Language Processing, NLP), стоящей на пересечении таких компьютерных наук, как машинное обучение и компьютерная лингвистика, решаются проблемы анализа, понимания и извлечения смысла из естественной человеческой речи.

Актуальность темы дипломной работы обусловлена стремительным ростом рынка онлайн-образования и тем, что изучение иностранных языков является приоритетным направлением для потребителей. Создатели онлайн-платформ часто называют персонализацию обучения отличительной особенностью их курсов. Однако несмотря на большую гибкость по сравнению с классическим методом изучения иностранного языка, некоторые материалы могут быть устаревшими и неактуальными. Это связано с тем, что процесс формирования упражнений является трудозатратным ввиду задействования человеческого труда и отсутствия программных средств для решения этой задачи.

Рассмотрим основные этапы изучения грамматических явлений:

- Presentation
  Демонстрация примеров и структуры грамматической конструкции.
- Pattern Recognition
  Самостоятельный поиск грамматических конструкций в исходном тексте.

- Controlled Practice
  Построение изучаемых грамматических конструкций в ограниченной форме.
- Semi-Controlled Practice
  Персонализация языка с упором на изучаемые грамматические явления.
- Free Practice
  Свободное использование языка.

Цель исследования - проектирование и реализация системы автоматического формирования заданий видов *Controlled Practice*, которая позволит сократить время подготовки методических материалов и дать возможность большей персонализации траектории обучения.

Для достижения результата были поставлены следующие задачи:

- изучить методы Data Mining и Machine Learning в области обработки естественного языка;
- изучить методы векторного представления слов;
- разработать метод предварительной очистки текста;
- изучить модели машинного обучения для извлечения необходимых признаков из текста;
- разработать алгоритм поиска грамматических конструкций;
- разработать алгоритм формирования заданий на основе найденных конструкций;
- реализовать развертывание сервера;
- разработать web-платформу для взаимодействия с пользователями;

- проанализировать возможности повышения производительности системы;
- проанализировать возможности повышения точности обнаружения грамматических конструкций;
- реализовать базу данных результатов пользователей для ведения статистики.

Объектом исследования дипломной работы является применение компьютерной лингвистики в области онлайн-образования.

Предметом исследования дипломной работы является построение алгоритмов поиска грамматических конструкций и формирования на их основе упражнений по грамматике английского языка с использованием методов компьютерной лингвистики и Machine Learning.

1. Теоретические основы задачи обработки естественного языка

Задача обработки естественного языка это целый ряд теоретико-мотивированных вычислительных методов, которые позволяют производить анализ человеческого языка. В рамках NLP решается большое количество задач, затрагивающих различные уровни- начиная от определения частей речи, заканчивая созданием диалоговых систем.

Долгое время в задачах NLP применялись модели поверхностного обучения, такие как SVM (англ. Support Vector Machine), обученные на разреженных данных, представленных в пространстве высокой размерности, что не позволяло достичь достаточной точности и требовало значительных вычислительных ресурсов. Однако в последние годы были разработаны новые методы векторного представления слов, что позволило избежать экспоненциального роста размерности. Это помогло сократить время обучения сетей и перейти к методам, основанным на глубоком обучении, которые обеспечивают автоматическое обучение признакам. Это существенно отличает новые архитектуры от тех, что применялись раньше, так как для них больше не требуется ручное конструирование признаков.

Одна из первых архитектур глубокого обучения в области NLP была продемонстрирована Ронаном Коллобертом и Джейсоном Уэстоном [2]. На тот момент она превосходила большую часть уже существующих моделей в таких задачах, как поиск именованных сущностей и др. С того времени появилось большое количество алгоритмов и моделей, решающих сложные задачи обработки естественного языка.

1.1. Стандартные задачи NLP

В области обработки естественного языка выделяют 4 базовые задачи: *POS tagging (POS), Chunking, Named Entity Recognition (NER), Semantic Role Labeling (SRL)*. Они лежат в основе более высокоуровневых задач, в том числе в задаче автоматического создания заданий по грамматике. Рассмотрим каждую из них в отдельности.

1.1.1. POS tagging

POS tagging — задача маркировки каждого слова в соответствии с принадлежностью его к той или иной части речи: имя существительное, имя прилагательное, глагол и т.д.

POS классификаторы часто базируются на классификаторах, которые обучены на текстовых "окнах". После этого результат передается алгоритму двунаправленного декодирования [3]. В качестве признаков может браться контекст слова и некоторые созданные вручную признаки. Для проверки качества работы модели в задаче POS tagging зачастую применяется размеченный корпус текстов Wall Street Journal (WSJ).

1.1.2. Chunking

Chunking — задача поверхностно-синтаксического анализа. В ней производится анализ предложения, который сначала индексирует составные части предложений (существительные, глаголы и т.д.), а затем связывает их в единицы более высокого порядка, такие как именные или глагольные группы и т.д.

Традиционные алгоритмы поверхностно-синтаксического анализа используют поиск по шаблонам (например, регулярные выражения). Алгоритмы с использованием машинного обучения могут учитывать контекстную информацию, что дает возможность улучшить точность восстановления семантических связей.

1.1.3. Named Entity Recognition

Named Entity Recognition — задача распознавания именованных сущностей. Является подзадачей Data Mining, в рамках которой необходимо найти и классифицировать именованные сущности в неструктурированном тексте по заранее заданным категориям, таким как имена людей, названия организаций, количество и т.д.

Распознавание именованных сущностей часто разбивается на две подзадачи: выявление имен и их классификация по типу сущности. Первая фаза рассматривается как проблема сегментации: зачастую именованные сущности рассматриваются как непрерывные промежутки токенов без вложенностей. Для второй фазы требуется определение онтологии, благодаря которой формируются категории вещей.

Существует несколько иерархий именованных сущностей. Так, категории BNN используются в вопросно-ответных системах и включают в себя 29 типов и 64 подтипа. В свою очередь иерархия Sekine является расширенной и состоит из 200 подтипов, а в 2011 Риттер представил иерархию, которая основана на общих типах объектов Freebase.

Системы NER могут быть основаны как на лингвистических методах грамматики, так и на статистических моделях. Несмотря на то, что системы, созданные вручную, имеют большую точность, они требуют месяцев работы профессиональных лингвистов. Подход с использованием методов машинного обучения сейчас является приоритетным, однако является менее устойчивым и требует большого объема аннотированных данных на этапе обучения моделей.

1.1.4. Semantic Role Labeling

Semantic Role Labeling — задача маркировки семантических ролей. Представляет собой процесс присваивания меток словам или

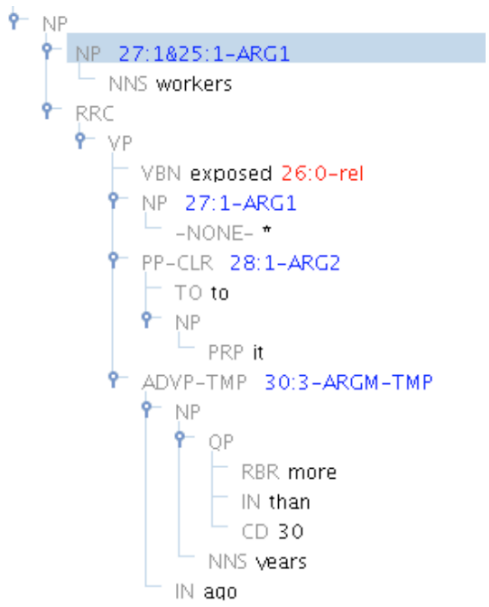

Рисунок 1 — Пример размеченных данных *PropBank*

фразам, которые характеризуют их семантические роли в предложении. Результатом работы является порождение поверхностной интерпретации, основанной на теории семантических ролей.

Процесс часто делят на несколько этапов: обнаружение и определение значения целевых предикатов, обнаружение и дальнейшую классификацию актантов.

Для реализации автоматической маркировки используются статистические модели, обученные на размеченных корпусах текстов, содержащих информацию о семантических ролях и предикатах. Примером такого корпуса выступает *PropBank*, который был получен добавлением вручную созданных семантических аннотаций в корпус *Penn Treebank*. Пример такой разметки можно увидеть на Рис. 1

Однако при использовании статистических моделей существует недостаток, который проявляется в зависимости от выбранного аннотированного ресурса. Кроме того, создание семантической разметки является трудозатратной и плохо формализуемой задачей. Из-за этого может проявляться некорректная работа на новых данных. Этот аспект принято называть проблемой доменной специфичности SRL.

1.2. Word Embeddings

Статистические модели стали основным инструментом в задачах NLP, однако в начале они страдали от т.н. проклятия размерности [4]. Это повлекло развитие алгоритмов, которые позволили бы представлять слова в низкоразмерном пространстве.

Стоит сказать, что на данный момент нет общепризнанного перевода термина embedding (от англ. ‘вложение’), поэтому будет использоваться англицизм.

Embedding представляет собой преобразование некой сущности в числовой вектор. Далее будут рассмотрены основные подходы, применяющиеся для решения данной задачи.

1.2.1. One-hot encoding

Одним из первых решений задачи векторного представления слов был так называемый унитарный код. Он представлял собой вектор длины n, которая определяется количеством слов некоторого словаря, содержащий $n - 1$ нулей и 1 единицу. Индекс значащей единицы соответствовал расположению слова в данном словаре —см. таблицу 1.

Несмотря на то, что такая архитектура позволяет решить проблему кодирования слов, она обладает рядом существенных недостатков:

- При добавлении нового слова в середину существующего словаря есть необходимость заново проводить нумерацию его элементов.
- Происходит быстрый рост размерности представления текстов. Например для текста из 9 уникальных слов требуется матрица 9x9
- Данный метод не предоставляет информации о семантической близости слов. Данный аспект связан с тем, что написание слов

Таблица 1 — One-Hot Encoded векторы

Vocabulary	1	2	3	4	5	6	7	8	9	10
bag	1	0	0	0	0	0	0	0	0	0
words	0	1	0	0	0	0	0	0	0	0
model	0	0	1	0	0	0	0	0	0	0
way	0	0	0	1	0	0	0	0	0	0
representing	0	0	0	0	1	0	0	0	0	0
text	0	0	0	0	0	1	0	0	0	0
data	0	0	0	0	0	0	1	0	0	0
simple	0	0	0	0	0	0	0	1	0	0
understand	0	0	0	0	0	0	0	0	1	0
implement	0	0	0	0	0	0	0	0	0	1

не имеет непосредственной связи с объектами, которые они описывают [5].

1.2.2. Bag of Words

На основе кодирования слов унитарным кодом, рассмотренным в пункте 1.1.1, был предложен более экономичный вариант представления текстов, называемый “мешком слов” (от англ. Bag of words).

Алгоритм построения такого представления содержит следующие основные этапы:

- Предварительное создание словаря методом OHE.
- Кодирование слов, содержащихся в тексте.
- Сложение всех полученных one-hot векторов.

На выходе получим числовой вектор, который описывает информацию о количестве различных слов в исходном тексте. Такая модель не сохраняет структуру входных данных, в связи с чем теряется информация о взаимном расположении слов. Тем не менее, применяя

Таблица 2 — Терм-документная таблица

Vocabulary	Documents	
	C1	C2
bag	1	1
words	1	1
model	1	1
way	1	0
representing	1	0
text	1	0
data	1	0
simple	0	1
understand	0	1
implement	0	1

данный подход можно сравнивать тексты путем сравнения выходных векторов. Примером такой метрики является косинусная мера:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}, \tag{1}$$

A_i, B_i – компоненты векторов $\mathbf{A}$ и $\mathbf{B}$ соответственно.

Рассмотрим этот процесс имплементации алгоритма.

1. Зададим экземпляры текста.
   C1 = *“The Bag of Words model is a way of representing text data.”*
   C2 = *“The Bag of Words model is simple to understand and implement.”*

2. Очистим тексты от пунктуационных символов и ‘стоп-слов’[1], которые не несут смысловой нагрузки.

3. Сформируем словарь и закодируем его методом ОНЕ (см. таблицу 1).

[1] the, of, is, a, to, and

4. Для каждого предложения сложим One-Hot векторы слов, которые входят в их составы (см. таблицу 2).

5. Применим косинусную меру (1) к полученным векторам, которые количественно описывают исходные тексты.

$$\text{similarity} = \cos(\theta) = \frac{3}{\sqrt{7} \cdot \sqrt{6}} = 0.46291$$

Кроме того, полученную Таблицу 2 с зависимостью “слово-документ” можно представить в виде произведения матриц “слово-тема” и “тема-документ”. Для этого можно использовать SVD-разложение:

$$(\mathbf{t}_i^T) \rightarrow \begin{matrix} & (\mathbf{d}_j) \\ & \downarrow \end{matrix} \begin{bmatrix} x_{1,1} & \dots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,n} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \\ \mathbf{u}_1 \\ \\ \end{bmatrix} & \dots & \begin{bmatrix} \\ \mathbf{u}_l \\ \\ \end{bmatrix} \end{bmatrix} \cdot \begin{bmatrix} \sigma_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_l \end{bmatrix} \cdot \begin{bmatrix} \begin{bmatrix} & \mathbf{v}_1 & \end{bmatrix} \\ \vdots \\ \begin{bmatrix} & \mathbf{v}_l & \end{bmatrix} \end{bmatrix} \tag{2}$$

$\mathbf{t}_i$ – слово, $\mathbf{d}_i$ – документ.

Применим такое разложение для текстов из примера и визуализируем (см. рис. 2). Как можно видеть, получаемые результаты имеют сильную зависимость от корпуса, к которому применяется разложение.

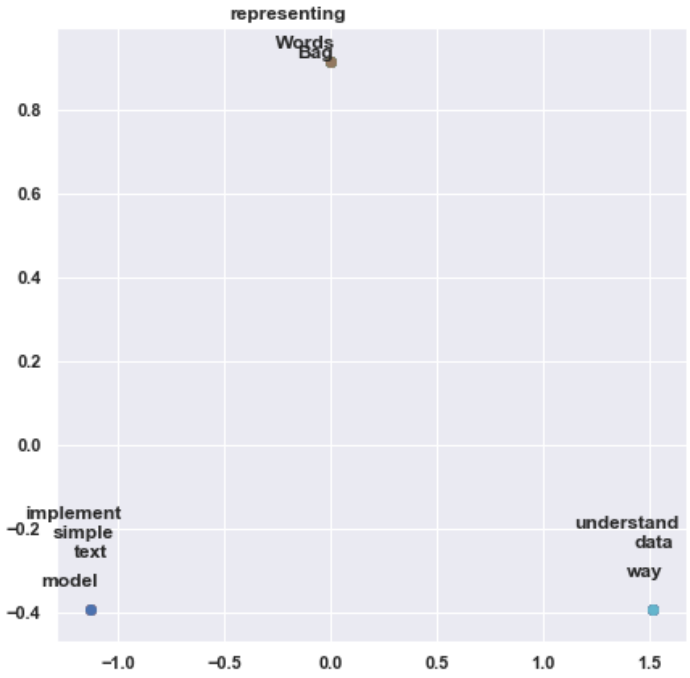

Рисунок 2 — SVD разложение для C1 и C2

1.2.3. Word2vec

Из-за ряда недостатков, которыми обладали традиционные алгоритмы, исследования в области представлений слов продолжились. В 2013 году Томаш Миколов представил подход [6] к кодированию слов, который решал целый ряд проблем, присущих другим моделям, в том числе рост размерности векторного пространства и невозможность сохранять семантическую близость.

Им было представлено 2 подхода, которые он назвал continuous bag-of-words (CBOW) и skip-gram. Они позволяют строить высококачественные распределенные векторные представлений.

Между представленными моделями есть существенное отличие. CBOW вычисляет условную вероятность появления целевого слова исходя из его контекста, который лежит в окне размера k. В свою очередь skip-gram предсказывает слова контекста по центральному сло-

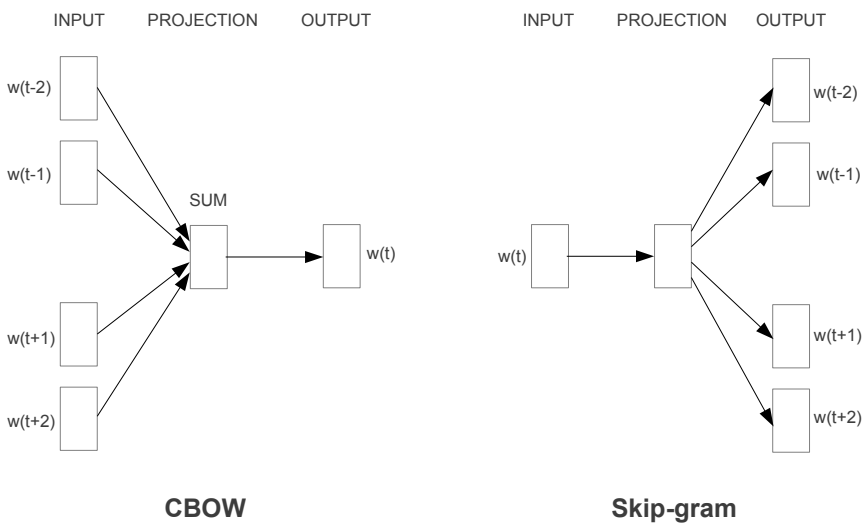

Рисунок 3 — Модели CBOW и Skip-gram (Источник рисунка: Mikolov [6])

ву. Предполагается, что контекстные слова расположены симметрично целевым словам на расстоянии, равном размеру окна в обоих направлениях. Схематичное представление данных методов показано на рис. 3

1.2.3.1. Модель CBOW Рассмотрим упрощенную модель CBOW (см. рис. 4) с контекстом из одного слова более детально. Она представляет собой полносвязную нейронную сеть со скрытым слоем. Входной слой, принимающий one-hot вектор, состоит из V нейронов, а скрытый слой из N нейронов. На выходном слое применяется операция Softmax, давая на выходе распределение вероятностей по всем словам в словаре. Слои связаны матрицами весов $\mathbf{W} \in \mathcal{R}^{V \times N}$ и $\mathbf{W}^{'} \in \mathcal{R}^{N \times V}$ соответственно.

Каждая строка матрицы $\mathbf{W}$ это N-мерное векторное представление $\mathbf{v}_w$ связанного слова входного слоя. Формально можно записать строку i матрицы $\mathbf{W}$ как $\mathbf{v}_w^T$.

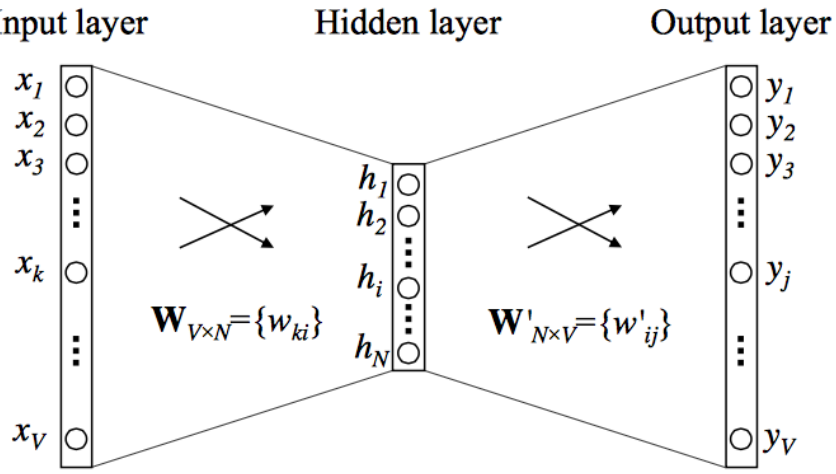

Рисунок 4 — Модель CBOW (Источник рисунка: Rong [7])

Таким образом,

$$\mathbf{h} = \mathbf{W}^T\mathbf{x} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I}, \tag{3}$$

где $\mathbf{v}_{w_I}$ - векторное представление входного слова $\mathbf{w_I}$.

Скрытый и выходной слой связаны матрицей $\mathbf{W}' = \{w'_{ij}\}$. Используя данную матрицу весов можно посчитать оценку u_j для каждого слова из словаря

$$u_j = {\mathbf{v}'_{w_j}}^T\mathbf{h}, \tag{4}$$

где $\mathbf{v}'_{w_j}$ это j-я колонка матрицы W'. После этого используется нелинейная функция Softmax (5) для получения апостериорного распределения слов.

$$p(w_j|w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V}\exp(u_{j'})}, \tag{5}$$

где y_j является выходом j-го нейрона на выходном слое. Используя формулы (3) и (4) можно записать (5) в следующем виде:

$$p(w_j|w_I) = \frac{\exp\left({\mathbf{v}'_{w_j}}^T\mathbf{v}_{w_I}\right)}{\sum_{j'=1}^{V}\exp\left({\mathbf{v}'_{w_{j'}}}^T\mathbf{v}_{w_I}\right)} \tag{6}$$

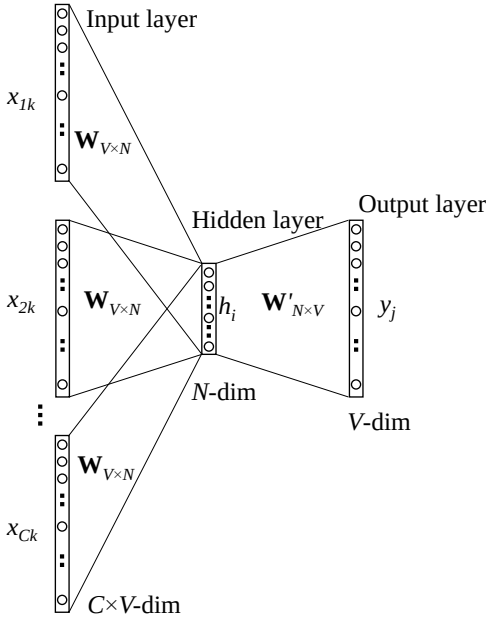

Рисунок 5 — Модель CBOW в общем случае

Учитывая, что $\mathbf{v}_w$ и $\mathbf{v}'_w$ - представления слова w через строки матрицы $\mathbf{W}$ и колонки матрицы $\mathbf{W}'$ соответственно. Будем называть $\mathbf{v}_w$ «входным вектором», а $\mathbf{v}'_w$ - «выходным вектором» слова w.

В качестве функции потерь, которую следует минимизировать, используется логарифмическая функция правдоподобия

$$E = -\log p(w_O|w_I) \tag{7}$$

где w_I - входное слово, а w_O- предсказанное.

В общем случае модели CBOW см. рис. 5, когда контекст превышает одно слово, при вычислении выходных значений скрытого слоя подсчитывается среднее значение векторов контекста:

$$\mathbf{h} = \frac{1}{C}\mathbf{W}^T(\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_C) \tag{8}$$

$$= \frac{1}{C}(\mathbf{v}_{w_1} + \mathbf{v}_{w_2} + \cdots + \mathbf{v}_{w_C})^T \tag{9}$$

где C- количество слов в контексте, $w_1, \cdots, w_C$ - контекстные слова, $\mathbf{v}_w$ - входное слово w.

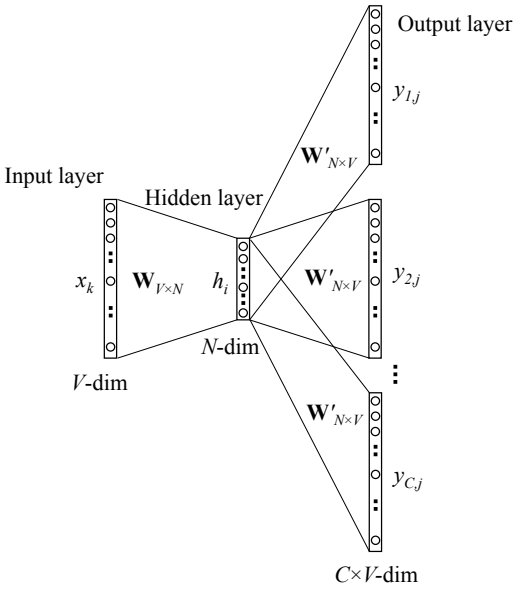

Рисунок 6 — Модель Skip-gram в общем случае

1.2.3.2. Модель Skip-gram Рассматриваемая общая модель (см. рис. 6) является обратной к CBOW. Теперь целевое слово подается на входной слой, а предсказание контекстных слов является результатом работы модели.

Будем использовать обозначение $\mathbf{v}_{w_I}$ для входного вектора. Через $\mathbf{h}$ обозначим выходные данные скрытого слоя, которые останутся прежними:

$$\mathbf{h} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I}, \tag{10}$$

Будем использовать обозначение $\mathbf{v}_{w_I}$ для входного вектора. Через $\mathbf{h}$ обозначим выходные данные скрытого слоя:

$$\mathbf{h} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I}, \tag{11}$$

Выходной же слой теперь формирует C полиномиальных распределений вместо одного. Каждый элемент выходного слоя считается с использование матриц скрытого и выходного слоя.

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{\exp(u_{c,j})}{\sum_{j'=1}^{V} \exp(u_{j'})} \tag{12}$$

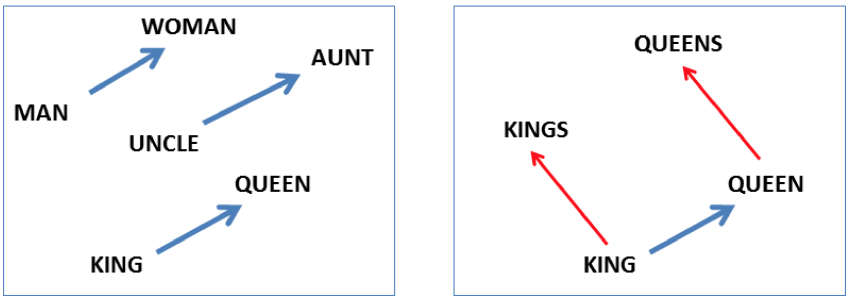

Рисунок 7 — Иллюстрация отношений векторного представления слов (Источник рисунка: Mikolov [8])

$w_{c,j}$ – j-е слово в c-м векторе выходного слоя;
$w_{O,c}$ – действительное c-е выходное контекстное слово;
w_I – единственное входное слово;
$y_{c,j}$ – предсказание j-го элемента в c-м векторе выходного слоя;
$u_{c,j}$ – входные данные j-го элемента в c-м векторе выходного слоя;

Функция потерь также меняется и принимает следующий вид:

$$E = -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C}|w_I) \tag{13}$$

$$= -\log \prod_{c=1}^{C} \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} \exp(u_{j'})} \tag{14}$$

$$= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} \exp(u_{j'}) \tag{15}$$

где j_c^* – индекс верного c-го выходного слова в словаре.

Таким образом, преимущество использования модели Skip-gram перед CBOW в том, что при том же корпусе текстов получается больше экземпляров обучающей выборки, из-за чего повышается точность предсказаний модели. Резюмируя, модель word2vec позволяет решить основную проблему векторного представления слов- размерность выходных векторов. Кроме того необходимо отметить главную особенность таких архитектур, которая заключается в том, что

несмотря на отсутствие сохранения порядка слов в контексте, так как входные векторы суммируются, они сохраняют некоторые семантические характеристики корпусов текста, на которых проводится обучение. В связи с этим схожие по смыслу слова имеют схожие векторные представления. Наиболее популярный пример сохранения семантической близости представлен на рис. 7.

1.3. Применение CNN в задачах NLP

Рассмотренный в предыдущем пункте метод векторного представления слов word2vec и схожие с ним (GloVe и др.) дали большой толчок в разработке архитектур глубоких нейронных сетей для задач NLP. Данные представления применяются как первый слой обработки данных в deep learning моделях.

Одним из методов извлечения признаков из текстовых данных является применение сверточных нейронных сетей (англ. Convolutional Neural Network, CNN). В задачах NLP наиболее часто встречаются варианты мультизадачных моделей, в которых результаты работы низкоуровневых слоев служат входными данными (признаками) для более высокоуровневых. Так, например, корректные POS тэги слов помогают улучшить показатель точности на задачах построения дерева зависимости.

Пример такой архитектуры был представлен авторами Коллобертом и Уэстоном [2]. В своей работе они использовать мультизадачную нейронную сеть, которая решает задачи определения частей речи, восстановления семантических ролей, поиск именованных сущностей и др.

Как видно из Рис. 8, в этой архитектуре была использована таблица поиска (англ. lookup table) для векторного представления слов, благодаря которой последовательность слов $\{s_1, s_2, ...s_n\}$ была преобразована в некоторую последовательность векторов $\{w_{s_1}, w_{s_2}, ...w_{s_n}\}$.

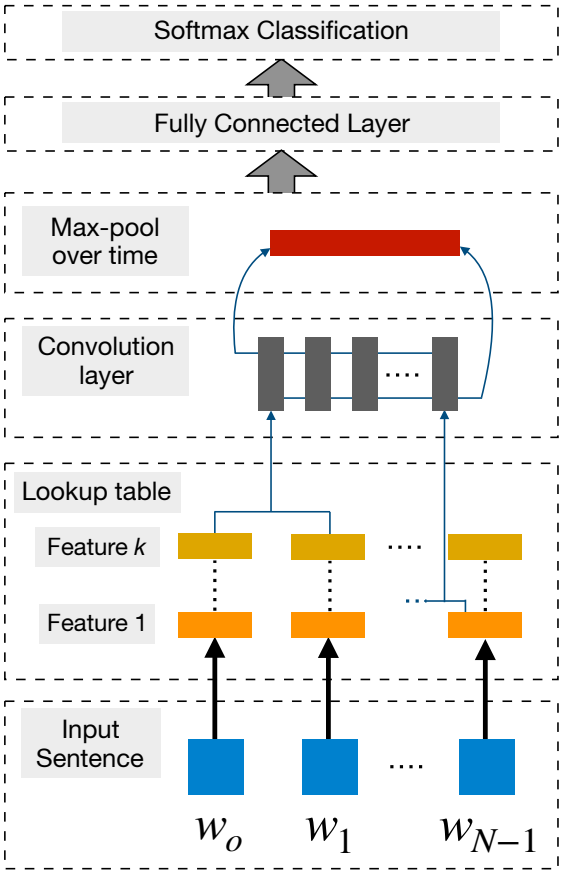

Рисунок 8 — Пример CNN для задачи классификации на уровне слов (Источник рисунка: Collobert [9])

Сверточные нейронные сети способны извлекать информацию из n-грамм входного предложения, что дает возможность создания информативного семантического представления для последующих задач. Рассмотрим несколько архитектур и применений CNN для обработки естественного языка.

1.3.1. Предсказание на уровне предложений

Положим $\mathbf{w}_i \in \mathcal{R}^d$ - векторное представление i-го слова входного предложения, где d - размерность векторов вложений. Тогда предложение, состоящее из n слов, может быть представлено матрицей вложений слов $\mathbf{W} \in \mathcal{R}^{n \times d}$. Пример такого представления можно увидеть на Рис. 9.

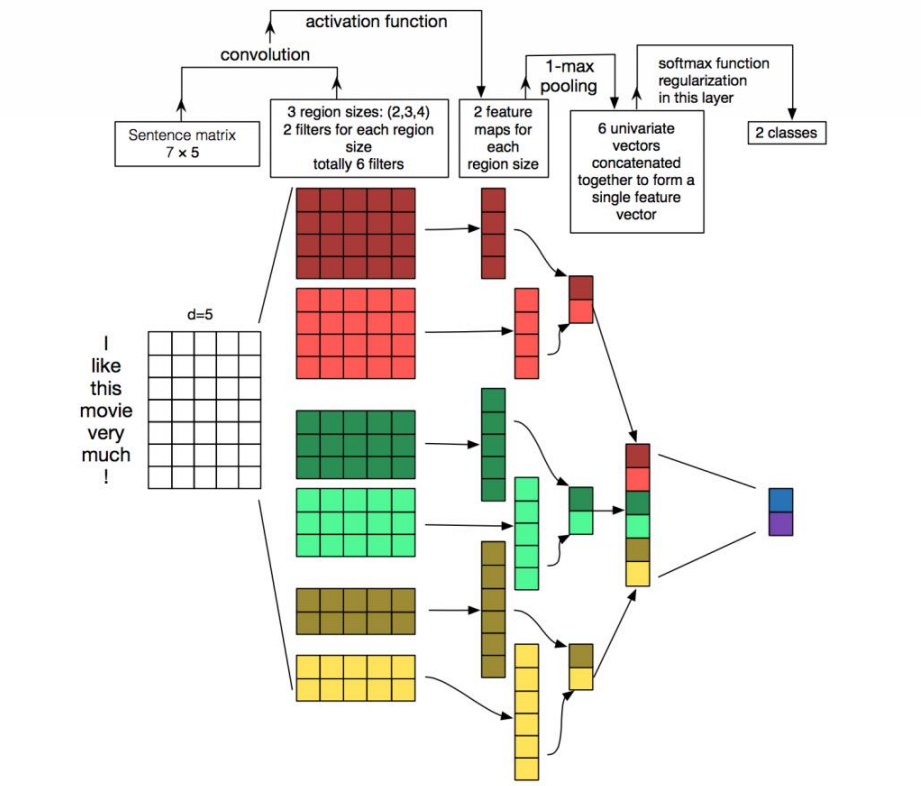

Рисунок 9 — Модель CNN

Обозначим $\mathbf{w}_{i:i+j}$ как конкатенацию векторов вложений слов $\mathbf{w}_i, \mathbf{w}_{i+1}, ...\mathbf{w}_{i+j}$. Начальная свертка применяется к этому входному слою и представляет собой фильтр $\mathbf{k} \in \mathcal{R}^{h \times d}$, который применяется к h словам, создавая новые признаки. Например, признак c_i получается путем применения окна свертки к словам $\mathbf{w}_{i:i+h-1}$

$$c_i = f(\mathbf{w}_{i:i+h-1}.\mathbf{k}^T + b) \tag{16}$$

$b \in \mathcal{R}$ - смещение, f - нелинейная функция активации (например, сигмоида).

Фильтр k применяется ко всем возможным окнам текста и формирует карту признаков

$$c = [c_1, c_2, ..., c_{n-h+1}] \tag{17}$$

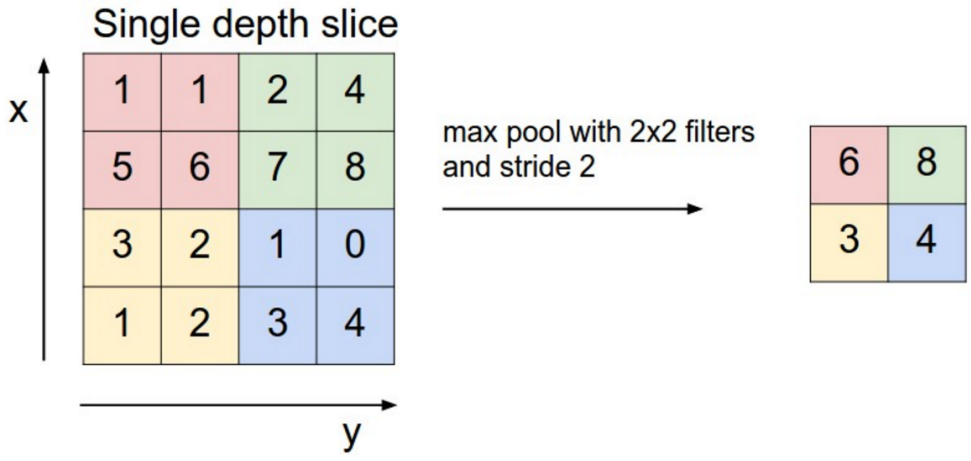

Рисунок 10 — Операция *max-pooling*

Стоит отметить, что представление текстовых данных отличается от представления изображений, к которым изначально применялись CNN. В отличие от них, текстовые данные представлены в двух, а не трех измерениях: ширине и количестве каналов. Ширину формирует количество слов в последовательности, а количество каналов - длина векторного представления слов. Учитывая то, что embedding слов сохраняет смысл только полностью, в задачах NLP применяются одномерные свертки (1D convolution), двигаясь только по оси, которая отвечает за количество слов.

В CNN применяется множество фильтров различной ширины, что позволяет извлекать специфичную информацию из различных n-грамм. Чаще всего после сверточного слоя идет слой с операцией *max-pooling*, которую обозначим $\hat{c} = max\{c\}$ (см. рис. 10). Эта операция выбирает максимальное значение из некоторого окна, проходясь по карте признаков. У применения такой операции есть несколько предпосылок. Первая - *max-pooling* обеспечивает контроль размерности выходных данных, что необходимо для задач классификации. Вторая причина заключается в том, что операция *max-pooling* позволяет более высокоуровневым слоям работать с информацией, которая захва-

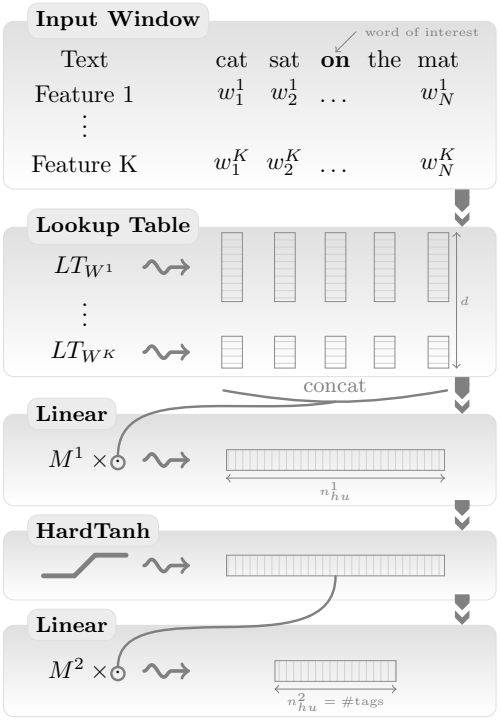

Рисунок 11 — Модель Window Approach (Источник рисунка: Collobert [2])

тывает больший участок данных, при этом такая операция передает наиболее значимые данные, полученные на всем предложении.

Комбинации слоев свертки и *max-pooling* выступают основными составляющими сверточной нейронной сети. Их последовательное применение совместно с большим количеством ядер свертки помогают улучшить качество анализа предложения и создавать абстрактные представления, содержащие семантическую информацию. При этом операции свертки на более глубоких слоях захватывают большую часть предложения. Такие операции продолжаются до тех пор, пока не захватят все предложение и не создадут обобщение его характеристик.

1.3.2. Предсказание на уровне слов

Рассмотренная архитектура позволяет делать предсказания на уровне предложений, однако такой подход не решает ряд задач, в

том числе описанных в пункте 1.1. Для них необходимо производить предсказания на уровне слов. Адаптация CNN под такие задачи заключается в применении т.н. "оконного подхода"(англ. Window Approach). Он основывается на гипотезе, что классификация слова зависит в первую очередь от слов, которые его окружают.

Таким образом, к каждому слову применяется окно фиксированного размера $2n + 1$, где целевое слово располагается в центре. Выделенный фрагмент теста рассматривается как под-предложение. Далее к нему применяется многослойная нейронная сеть и производится предсказание некоторых признаков целевого слова. Общая архитектура такой сети представлена на Рис. 11. Следует отметить, что она рассматривает слова как последовательности, что особенно актуально для таких задач, как SRL.

2. Задача автоматического формирования заданий по грамматике

Прикладная часть исследования заключается в создании системы, способной автоматически создавать задания по грамматике английского языка, используя в качестве входных данных произвольный текст. В качестве приоритетных направлений были выбраны задания вида Controlled Practice в разделах Active и Passive Voice с учетом всех возможных времен.

Цель создания такой системы состоит в том, чтобы предоставить альтернативу стандартным учебникам и тренажерам по грамматике, которые используют заранее сформированные материалы и не дают достаточной гибкости в выборе методических материалов, что сказывается на заинтересованности студента в их изучении.

В отличие от такого подхода, формирование заданий из пользовательских материалов может повысить вовлеченность и учесть специфическую лексику, которая используется в сферах, интересных конкретному пользователю.

Система представляет собой клиент-серверное приложение и состоит из 3х основных блоков:

- получение и предобработка текстовых данных;
- поиск грамматических конструкций;
- визуализация результатов и обработка вводимых пользователем данных.

2.1. Механика взаимодействия с пользователем

Для взаимодействия с пользователем была выбрана концепция web-приложения, которое встраивается в браузер Chrome. Таким образом, для пользователя приложение представлено двумя элемента-

ми: расширением для браузера и web-страницей, на которой происходит визуализация результатов.

В расширении предлагается выбрать основной раздел с заданиями, после чего конкретизировать свой запрос (см. рис. 12).

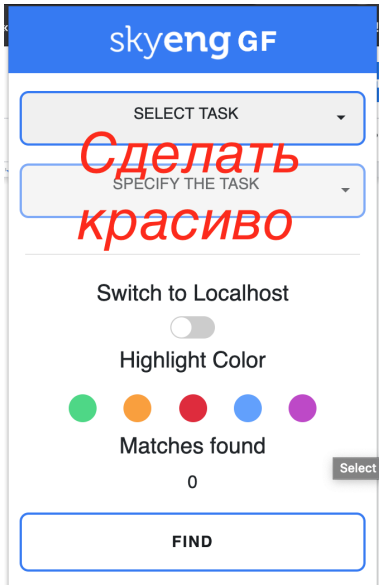

Рисунок 12 — Расширение для браузера Chrome

На странице с заданиями, которые соответствуют запросу и формируются на базе текстовой информации, находящейся на странице, где было вызвано расширение, пользователь может видеть свои результаты в интерактивном режиме (см. рис. 13), при этом вся статистика ответов записывается и хранится как локально, так и на сервере в базе данных.

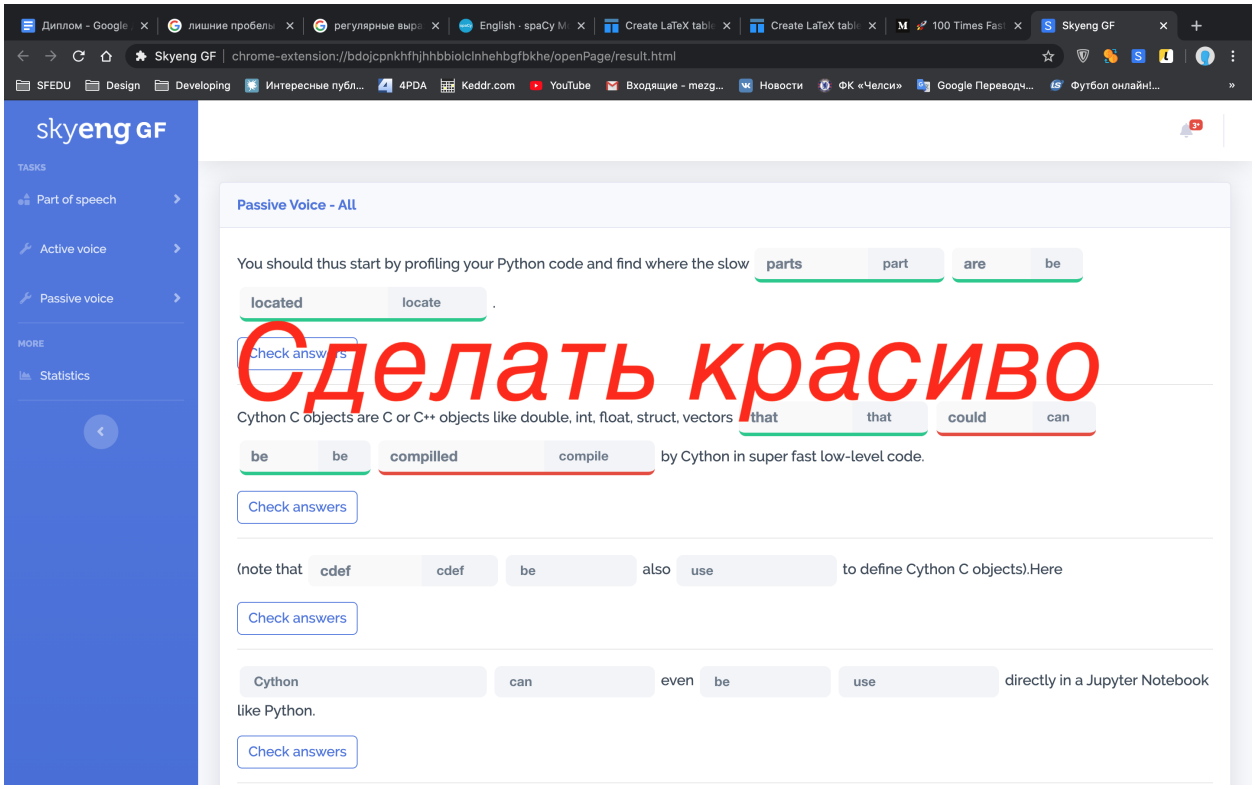

Рисунок 13 — Страница со сформированными заданиями

Таблица 3 — Сравенение выстродействия NLP библиотек

SYSTEM	TOKENIZE	TAG	PARSE
spaCy	0.2ms	1ms	19ms
CoreNLP	0.18ms	10ms	49ms
ZPar	1ms	8ms	850ms
NLTK	4ms	443ms	n/a

2.2. Используемые инструменты и оборудование

Для обработки текста и поиска необходимых грамматических конструкций была использована связка модели машинного обучения и Rule-Based алгоритма, который применялся к полученным моделью атрибутам.

В выборе инструментов обработки текстовых данных скорость работы была первоочередным приоритетом ввиду того, что система является клиенто-ориентированной и требует достижения минимального времени на обработку и передачу данных. В Таблице 3 приведено сравнение времени работы различных библиотек для ряда задач.

Исходя из результатов было принято решение остановиться на *spaCy*. Кроме того, данная библиотека обладает обширной документацией, что облегчает работу с ее компонентами.

Для развертывания сервера был выбран микрофреймворк *Flask*, который использует набор инструментов Werkzeug и шаблонизатор Jinja2. При этом он обладает достаточной производительностью и совместим с *Python 3.x*, что позволяет использовать его вместе с *spaCy*.

Характеристики сервера следующие:

- Memory: 16GB
- Processor: Intel Core i5-8400 CPU @ 2.80GHz x6
- Graphics: GeForce GTX 1070/PCIe/SSE2
- OS: Ubuntu 18.04.4 LTS

2.3. Получение текстовых данных

Текстовые данные получаются путем их извлечения из html-файла. Для этого производятся следующие операции:

- удаление информации за пределами тега <body> (см. листинг 2.1);
- очистка от тегов;
- очистка от скриптов
- очистка от номеров с общими суффиксами;
- очистка от изображений.

Для работы с подстроками используются правила, основанные на регулярных выражениях.

Листинг 2.1. Удаление информации за пределами тега <body>

```
getStrippedBody: function (html) {
  let body = html.match(/<body[^>]*>(?:([^]*)<\/body>([^]*)
     |([^]*))/i);
  if (body && body.length > 1) {
  if (body[2] && body[2].length > this.minBodyTailLength()) {
        body = body[1] + ' ' + body[2];
      } else if (body[1] === undefined) {
                body = body[3];
            } else {
                body = body[1];
            }
        } else {
            body = html;
        }

        return body.replace(/<script\b[^>]*(?:>[^]*?<\/script
           >|\/>)/ig, '<blink/>');
    },
```

Таблица 4 — Сравнение скорости работы NLP библиотек

MODEL	VERSION	LAS	UAS	NER F	POS	SIZE
en_core_web_sm	2.2.5	89.71	91.62	85.55	97.05	35MB
en_core_web_lg	2.2.5	90.17	92.01	86.55	97.22	812MB

2.4. Модель машинного обучения

Для извлечения атрибутов, которые необходимы для применения Rule-Based алгоритма, было принято решение использовать сверточную нейронную сеть, принципы работы которой были разобраны в пункте 2.1.

В ходе выбора архитектуры были проанализированы предобученные многозадачные нейронные сети *en_core_web_sm* и *en_core_web_lg*, обученные на OntoNotes 5. После их применения элементам текста присваиваются метки, обозначающие части речи (TAGGER), синтаксические отношения между словами (PARSER) и именованные сущности (NER).

Основное отличие состоит в применении метода GloVe [10] для векторного представления слов в *en_core_web_lg*. Данный метод комбинирует в себе принципы word2vec (см. пункт 1.2.3) и SVD-разложения. В то же время архитектура *en_core_web_sm* использует Bloom Embeddings [11], что позволяет существенно сократить вес модели, однако влечет за собой значительное снижение сохранения семантических связей.

Сравнение основных показателей до применения методов Transfer Learning можно видеть в таблице 4.

Основную роль для задачи поиска грамматических конструкций играет правильное построение дерева зависимостей, поэтому главным показателем в приведенной таблице является LAS (анг. Labeled Attachment Score).

Также были проведены тесты для измерения скорости их работы. Для теста были взяты тексты средней длины (порядка 20 000 сим-

Таблица 5 — Сравнение скорости работы моделей

MODEL	Loading	Preparation	Text Processing	Full Time
en_core_web_sm	0.59028	2.4e-5	0.51975	1.11307
en_core_web_lg	8.51049	2.3e-5	0.47446	8.98792

волов) и проведены замеры скорости работы на 50 итерациях. После этого результаты были округлены. Результаты представлены на таблице 5.

Как мы видим, наиболее затратным по времени является этап загрузки модели, а не момент ее применения. Таким образом, в случае предварительной загрузки на сервере, выбор большей модели не скажется на времени отклика работы программы.

При этом применение модели *en_core_web_lg* показывает рост точности определения семантических и синтаксических связей между членами предложения, что позволяет более точно находить грамматические конструкции в предложениях со сложной структурой.

Следовательно, окончательным выбором стала модель *en_core_web_lg*.

2.5. Предварительная обработка текста и настройка модели

Базовая очистка текста включает в себя удаление нестандартных символов, в том числе различных дефисов (см. листинг 2.2), смайлов, излишних пробелов. Удаление стоп-слов (предлоги, суффиксы, причастия и пр.) не производится ввиду необходимости сохранения начальной структуры предложения для последующего формирования упражнений.

Этап настройки модели состоит в применение нестандартных правил для компонентов *Tokenizer* и *Matcher*. Данные меры необхо-

Листинг 2.2. Предварительная очистка текста

```
def replace_canadian_period(text):
    text = text.replace(u"\u1427", ".")
    return text


def replace_fancy_hyphens(text):
    hlist = [u"\u002d", u"\u058a", u"\u058b", u"\u2010",
             u"\u2011", u"\u2012", u"\u2013", u"\u2014",
             u"\u2015", u"\u2e3a", u"\u2e3b", u"\ufe58",
             u"\ufe63", u"\uff0d"]
    for h in hlist:
        text = text.replace(h, "-")
    return text


def lexical_processor(text):
    text = os.linesep.join([s for s in text.splitlines() if s])
    text = replace_canadian_period(text)
    text = replace_fancy_hyphens(text)
    return text
```

димы для корректного отображения токенов при формировании заданий.

В случае *Tokenizer* был изъят дефис из набора символов, которые разделяют слово на части: например, слово “semi-controlled” при изначальных настройках делится на токены ”semi”, ”-”, ”controlled”.

Для компонента *Matcher* были добавлены правила, которые позволяют собрать токены, обозначающие сокращенное написание слов ”am”, ”have” в один: например, конструкции ”I’m” и ”I’ve” при изначальных настройках делится на токены ”I”, ” ’ ”, ”m” и ”I”, ” ’ ”, ”ve” соответственно.

После применения дополнительных правил разбиения и форматирования (см. листинг. 2.3) текст становится приемлемым для визуализации пользователю.

Листинг 2.3. Модификация компонентов *Tokenizer* и *Matcher*

```
# Change tokenizer of the NLP Model.
infixes = (
        LIST_ELLIPSES
        + LIST_ICONS
        + [
            r"(?<=[0-9])[+\-\*^](?=[0-9-])",
            r"(?<=[{al}{q}])\.(?=[{au}{q}])".format(
                al=ALPHA_LOWER, au=ALPHA_UPPER,
                q=CONCAT_QUOTES
            ),
            r"(?<=[{a}]),(?=[{a}])".format(a=ALPHA),
            r"(?<=[{a}0-9])[:<>=/](?=[{a}])".format(a=ALPHA),
        ]
)
infix_re = compile_infix_regex(infixes)

# Change matcher of the NLP Model
matcher = Matcher(nlp.vocab)
pattern = [{'ORTH': "'"},
           {'ORTH': 've'}]
pattern_2 = [{'ORTH': "'"},
             {'ORTH': 'm'}]

matcher.add('QUOTED', None, pattern, pattern_2)
```

2.6. Построение алгоритма поиска грамматических конструкций

Для поиска грамматических конструкций используется алгоритм, основанный на навигации по дереву зависимостей. Оно может быть построено на основе наборов признаков, которые формируются для каждого члена предложения в ходе применения модели. К таким признакам относятся:

- Text: оригинальный текст слова.
- Lemma: начальная форма слова.
- POS: простая метка части речи по модели UPOS.
- Tag: подробная метка части речи.

Таблица 6 — Признаки, полученные моделью *en_core_web_lg*

TEXT	LEMMA	POS	TAG	DEP	SHAPE	ALPHA	STOP
Apple	apple	PROPN	NNP	nsubj	Xxxxx	True	False
is	be	VERB	VBZ	aux	xx	True	True
looking	look	VERB	VBG	ROOT	xxxx	True	False
at	at	ADP	IN	prep	xx	True	True
buying	buy	VERB	VBG	pcomp	xxxx	True	False
U.K.	u.k.	PROPN	NNP	compound	X.X.	False	False
startup	startup	NOUN	NN	dobj	xxxx	True	False

- Dep: синтаксическая зависимость, связь между токенами.
- Shape: форма слова - заглавные буквы, знаки препинания, цифры.
- Is alpha: состоит ли токен из буквенных символов?
- Is stop: является ли токен частью стоп-листа, то есть наиболее распространенных слов языка?

Рассмотрим пример построения признаков для следующего предложения: “Apple is looking at buying U.K. startup”. Таблица 6 демонстрирует результат применения модели *en_core_web_lg*.

На базе полученных данных может быть построено дерево зависимостей, пример приведен на рисунке 14.

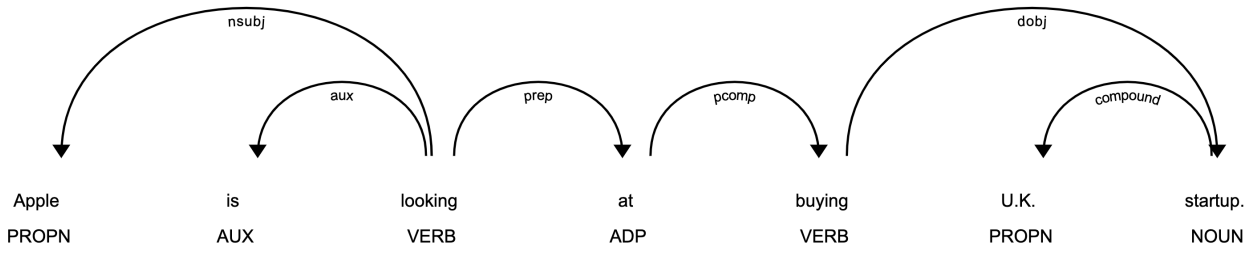

Рисунок 14 — Пример построения дерева зависимостей

Далее были построены наборы правил, которые задают необходимые шаблоны, ориентированные на распознание различных грам-

Таблица 7 — Шаблоны для детекции конструкций *Passive Voice*

TASK	AUX	AUXPASS	NUMBER OF AUX
Present Simple	n/a	am, is, are	0
Present Continuous	am, is, are	being	1
Present Perfect	have, has	been	1
Future In The Past Perfect	should, would, have, has	been	2

матических конструкций. Для детекции *Active* и *Passive Voice* основными атрибутами при построении правил стали соответствия *auxiliary verb (aux)*, *passive auxiliary of a clause (auxpass)*, *verb tag*, а также соответствие набору зависимостей.

Паттерны были сформированы на основе формальной грамматики английского языка с учетом особенностей применяемых статистических моделей. Пример таких правил представлен в таблице 7.

Следующий этап – навигация по дереву зависимостей. Результатом работы алгоритма должны стать массивы найденных фраз, их индексов, восстановленных начальных форм, POS тегов и тегов зависимостей слов, входящих в найденные фразы, а также предложений, к которым они относятся.

Обобщенный алгоритм поиска состоит из следующих этапов:

1. Создание массивов, которые соответствуют ожидаемым результатам.
2. Навигация по предложениям в тексте.
3. Поиск корневого слова в предложении.
4. Создание локальных массивов, относящихся к данному предложению.

5. Навигация по словам, зависящим от корневого.
6. Проверка соответствия зависимых слов набору правил.
7. Добавление подходящих слов в локальные массивы.
8. Контрольная проверка найденных соответствий общему шаблону.
9. Переход к действиям в зависимости от результата:
    9.1. При соответствии общему шаблону добавить найденную информацию в общие массивы.
    9.2. При несоответствии общему шаблону обнулить массивы и прервать навигацию по зависимым словам.
10. Продолжение навигации по следующим предложениям.

Момент обработки *auxiliary verb (aux)* для задания *Passive Voice* приведен на листинге 2.6.

Полученные данные отправляются на сторону клиента для дальнейшей обработки и формирования упражнений.

Листинг 2.4. Обработка *'aux'* при поиске конструкций *Passive Voice*

```
# 3: An auxiliary of a clause
if child.dep_ == 'aux':
    if child_lower in tense_rule.get('aux'):
        num_of_aux += 1
        passive_match.append(child.text)
        passive_match_lexemes.append(child.lemma_)
        passive_match_indices.append([
                child.idx - sent.start_char,
                child.idx + len(child) - sent.start_char])
        passive_match_pos.append(child.pos_)
        passive_match_dep.append(child.dep_)

        # Exception handling with Modals
        if tense == 'MODALS' and child_lower == 'have':
            num_of_aux_rule += 1
    else:
     # If we find a match but from another tense => skip
     passive_match = []
     passive_match_lexemes = []
     passive_match_indices = []
     passive_match_pos = []
     passive_match_dep = []
     num_of_aux = 0
     num_of_aux_rule = tense_rule.get('num_of_aux')
     break
```

2.7. Формирование упражнений

Кратко рассмотрим механизм формирования и визуализации упражнений.

Полученная информация с найденными фразами, лексемами, индексами и их предложениями передается в функцию, которая создает блоки с упражнениями. Каждый такой блок представляет собой элемент, который хранит информацию о типе задания, порядковом номере упражнения, а также содержит поля ввода, которые вставляются на места слов, входящих в состав найденных фраз. Для помощи пользователю вставляются восстановленные начальные формы вырезанных слов.

Работа над формированием блоков ведется путем редактирования строк, которые представляют собой блоки html кода. После окончательного формирования элемента производится его интеграция на открытую страницу. Для такой работы с html используется класс htmlString, который входит в состав Jquery.

Пример сформированного блока можно увидеть на листинге 2.5.

Листинг 2.5. Пример упрощенной структуры элемента с заданием

```
<div class="task-p">
<p>You should thus start by profiling your Python code and find
 where the slow
    <label class="input__label" for="task-0-0">
        <span class="input__label-content">part</span>
    </label>

    <label class="input__label" for="task-0-1">
        <span class="input__label-content ">be</span>
    </label>

    <label class="input__label " for="task-0-2">
        <span class="input__label-content">locate</span>
    </label>
.</p>
</div>
```

2.8. Сохранение статистики пользователя

Все вводимые пользователем данные при прохождении упражнений учитываются в статистике. Для ее хранения используется структура Json.

В ней сохраняется информация о источнике данных (url-адрес), информация о проделанных заданиях, информация о количестве правильных и неправильных ответов для каждого слова с учетом их частей речи и принадлежности к той или иной части предложения. Такие сведения позже могут комбинироваться для предоставления статистики пользователю (см. рис. 15), а также использоваться для построения рекомендательной системы.

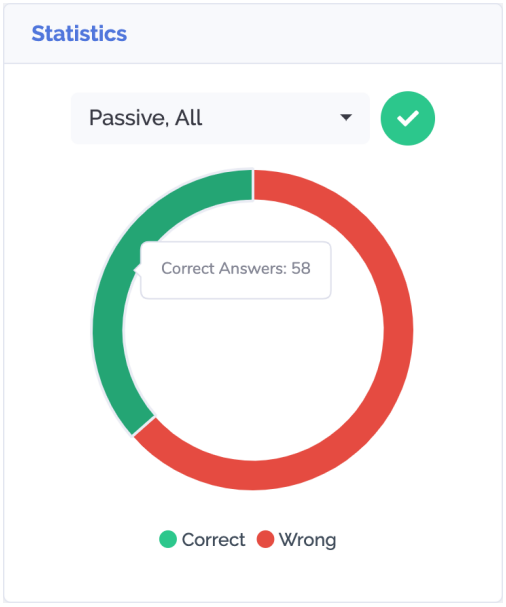

Рисунок 15 — Пример визуализации статистики

Хранение статистики происходит в веб-браузере путем использования *Window.localStorage*. Данные также дублируются в *NoSQL* базе данных *mongoDB*. Ввиду отсутствия на данный момент этапа авторизации в системе, в качестве идентификатора пользователя используется уникальный идентификатор расширения.

Восстановление данных статистики производится при каждом новом открытии страницы с заданиями. Краткий алгоритм состоит из перечисленных этапов, при этом переход к следующему происходит при неудаче на предыдущем:

1. Попытка восстановить данные из *localStorage*.
2. Попытка восстановить данные из *mongoDB*.
3. Инициализация новой структуры и запись ее в *localStorage* и *mongoDB*.

Обновление данных внутри *localStorage* происходит после каждого нового ответа пользователя, а запись в удаленную базу данных каждые 5 ответов, чтобы снизить нагрузку на нее и предотвратить задержки в работе приложения.

2.9. Повышение производительности системы

Было произведено тестирование скорости работы модели *en_core_web_lg*. Целью проведения данных тестов стало выявление оптимальных параметров, которые помогают повысить производительность системы.

Были выявлены следующие корреляции:

- Зависимость скорости работы от количества пакетов, на которые разбивается входной текст.
- Зависимость скорости работы от размера пакетов, на которые разбивается входной текст.
- Зависимость скорости работы от выбранного оборудования.
- Зависимость скорости работы от размера входного данных.

Для проведения тестов использовались тексты, длина которых составляла от 5 000 до 60 000 символов с шагом в 5 000 символов.

Для получения достаточного количества текстов, их формирование происходило на основе датасета CoNLL-2000. Для этого его текст был приведен в удобный для анализа вид, после чего были сформированы группы по 20 текстов каждой длины (для 60 000 символов - 18 текстов). Таким образом, тестовая выборка составила 238 различных текстов.

Для подвыборки из 20 текстов были проведены начальные тесты, целью которых было выявление разумных границ тестирования. Полученные результаты:

- Минимальный размер пакета: 700 символов
- Максимальный размер пакета: 5000 символов
- Шаг: 100 символов

На основе полученных данных был произведен полноценный тест.

Методика тестирования была следующей:

1. Каждый из сформированных текстов подавался на вход процедуре тестирования по 5 раз.
2. Для каждой итерации выполнялись разбиения входных данных на пакеты длиной 700…5000 символов с шагом = 100 (43 итерации).
3. Полученные значения записывались в csv файл.
4. Были найдены средние значения результатов времени обработки каждого текста.
5. После получения индивидуальных результатов для всех 238 текстов, результаты группировались по длине текстов, к которым они относились.
6. Были найдены средние значения для сгруппированных результатов.
7. Все результаты были проанализированы на предмет выявления зависимостей
8. Все результаты, как сгруппированные, так и индивидуальные, были визуализированы.
9. Аналогичные действия были произведены с использованием GPU.

В сумме было проведено 102 340 замеров (CPU + GPU).

Как мы видим из полученных результатов на рисунке 16 (полное сравнение приведено в приложении 1 и приложении 2), основной прирост скорости достигается за счет перехода с CPU на GPU. В отличие от изначального предположения, резкого роста времени работы на диапазоне 1000-5000 символов не наблюдается (отсекаем минимальный размер пакетов, так как из-за их большого количества производительность резко падает), однако проведенные тесты помогают

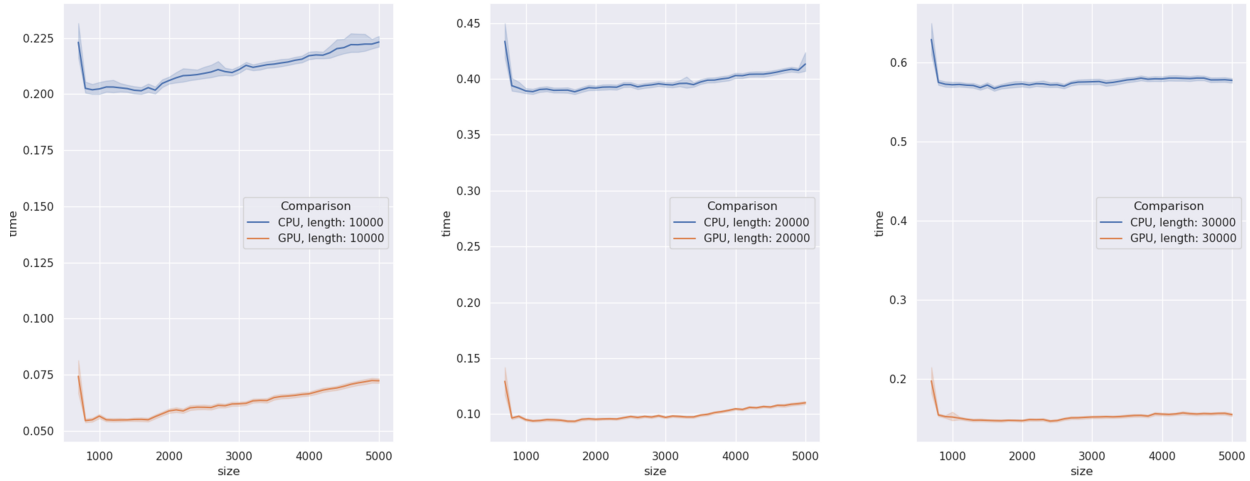

Рисунок 16 — Результат тестов производительности модели

выявить зависимость минимального времени выполнения от размера входных данных (длины текста).

По результатам теста можно сказать, что для больших текстов (>25К символов), выбор значения в данном промежутке не является принципиальным и следует выбирать больший, так как в дальнейшем алгоритм поиска грамматических конструкций будет работать быстрее за счет меньшего количества итераций цикла (меньше получаемых пакетов). Для малых же текстов правильный выбор параметра из данного промежутка может показать значительный рост производительности в процентном соотношении.

Полученные данные помогают динамически подбирать параметр разбиения текста на пакеты. Однако при реализации такого подхода требуется учитывать, что граница пакета может попасть на середину слова или предложения. Реализация такого алгоритма представлена в приложении 3.

Заключение

Типа сделал это хуйню:

- изучить методы Data Mining и Machine Learning в области обработки естественного языка;
- изучить методы векторного представления слов;
- разработать метод предварительной очистки текста;
- изучить модели машинного обучения для извлечения необходимых признаков из текста;
- разработать алгоритм поиска грамматических конструкций;
- разработать алгоритм формирования заданий на основе найденных конструкций;
- реализовать развертывание сервера;
- разработать web-платформу для взаимодействия с пользователями;
- проанализировать возможности повышения производительности системы;
- проанализировать возможности повышения точности обнаружения грамматических конструкций;
- реализовать базу данных результатов пользователей для ведения статистики.

Список литературы

1. *Turing A. M.* Computing Machinery and Intelligence // Mind. — 1950. — Т. 59, № 236. — С. 433—460. — (New Series). — ISSN 00264423. — URL: http://www.jstor.org/stable/2251299.
2. Natural Language Processing (almost) from Scratch / R. Collobert [и др.] // CoRR. — 2011. — Т. abs/1103.0398. — arXiv: 1103.0398. — URL: http://arxiv.org/abs/1103.0398.
3. *Watanabe T.*, *Sumita E.* Bidirectional Decoding for Statistical Machine Translation // COLING 2002: The 19th International Conference on Computational Linguistics. — 2002. — URL: https://www.aclweb.org/anthology/C02-1050.
4. *Bellman R.*, *Corporation R.* Dynamic Programming. — Princeton University Press, 1957. — (Rand Corporation research study). — URL: https://books.google.com.ua/books?id=rZW4ugAACAAJ.
5. *Saussure F. de*. Course in General Linguistics. — London : Duckworth, [1916] 1983. — (trans. Roy Harris).
6. Efficient Estimation of Word Representations in Vector Space / T. Mikolov [и др.] // 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings / под ред. Y. Bengio, Y. LeCun. — 2013. — URL: http://arxiv.org/abs/1301.3781.
7. *Rong X.* word2vec Parameter Learning Explained // ArXiv. — 2014. — Т. abs/1411.2738.
8. *Mikolov T.*, *Yih W.-t.*, *Zweig G.* Linguistic Regularities in Continuous Space Word Representations // Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. — Atlanta, Georgia : Association for Computational Linguistics, 06.2013. — С. 746—

751. — URL: https://www.aclweb.org/anthology/N13-1090.

9. *Collobert R.*, *Weston J.* A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning // Proceedings of the 25th International Conference on Machine Learning. — Helsinki, Finland : Association for Computing Machinery, 2008. — C. 160—167. — (ICML '08). — ISBN 9781605582054. — DOI: 10.1145/1390156.1390177. — URL: https://doi.org/10.1145/1390156.1390177.

10. *Pennington J.*, *Socher R.*, *Manning C. D.* Glove: Global Vectors for Word Representation. // EMNLP. T. 14. — 2014. — C. 1532—1543.

11. *Serrà J.*, *Karatzoglou A.* Getting deep recommenders fit: Bloom embeddings for sparse binary input/output networks // CoRR. — 2017. — T. abs/1706.03993. — arXiv: 1706.03993. — URL: http://arxiv.org/abs/1706.03993.

Оформить как приложение 1

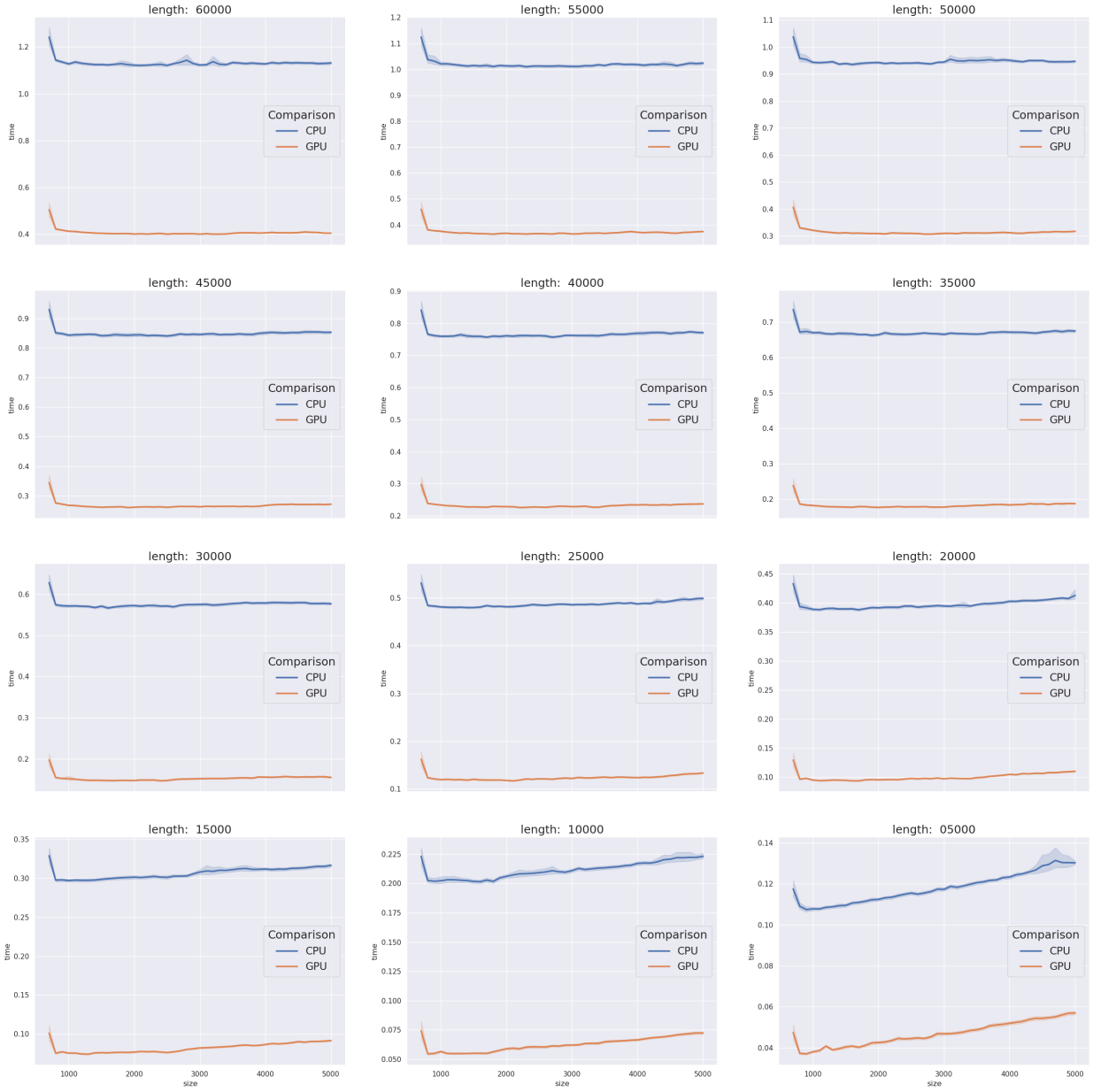

Рисунок 17 — Зависимость времени работы от длины текста

Оформить как приложение 2

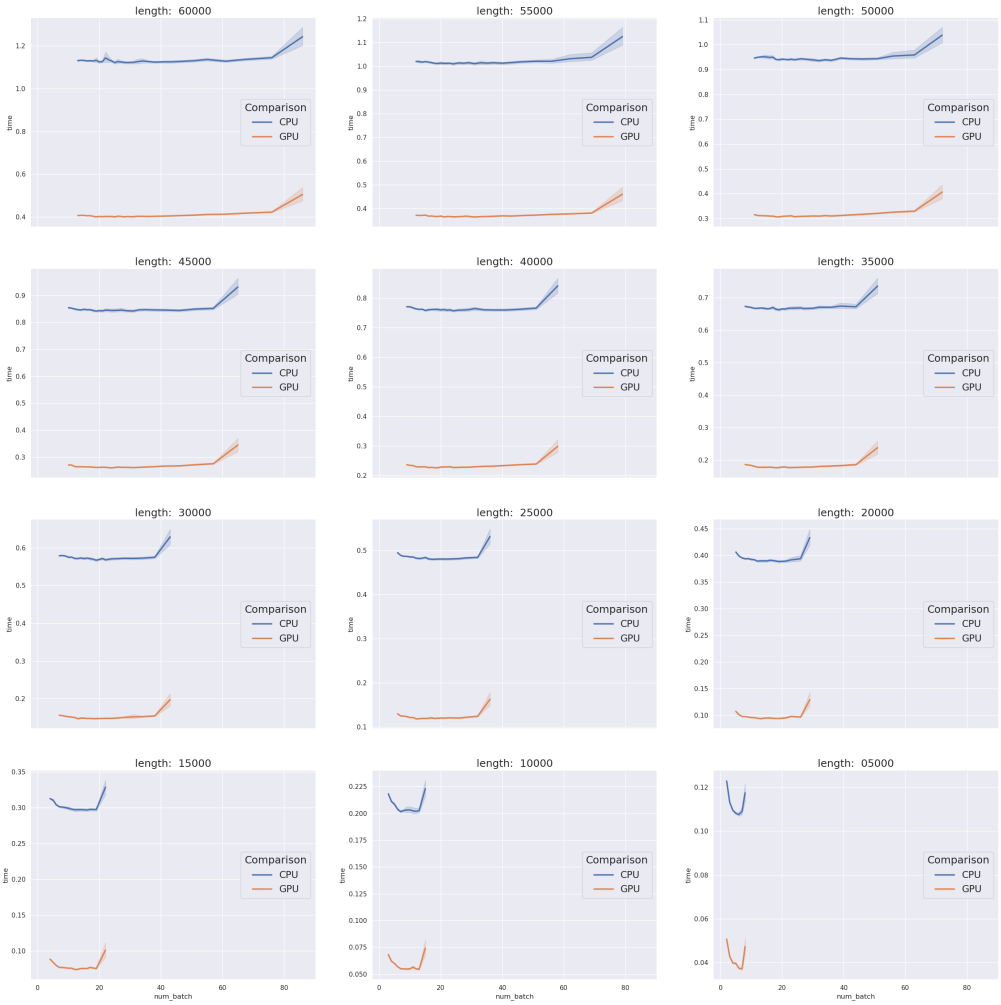

Рисунок 18 — Зависимость времени работы от количества пакетов

Оформить как приложение 3

Листинг 2.6. Обработка *'aux'* при поиске конструкций *Passive Voice*

```
def flexible_batch_indices(text, approximate_batch_size):
    exp = r'[.?!](?= [A-Z]|$)'
    cur_index = 0
    find_start = 0
    batch_indices = [0]
    sentence_found = False
    # Continue while start index < text length.
    while find_start < len(text):
        find_start = cur_index + approximate_batch_size
        find_end = find_start + approximate_batch_size

        # Check if the right index is less or equal to the length
            of the text.
        if find_end > len(text):
            find_end = len(text)

        match = re.search(exp, text[find_start:find_end])
        # If a match is found, recalculate the indices and add to
            the list.
        if match:
            cur_index = match.end() + find_start - 1
            batch_indices.append(cur_index)
            sentence_found = True
        else:
            # Just shift the index otherwise.
            cur_index += approximate_batch_size
            sentence_found = False
    if sentence_found:
        batch_indices.append(len(text))

    return batch_indices
```