

## **ВЫПУСКНОЙ ПРОЕКТ**

учащихся группы

**Старостина Ярослава Ильича и Рекута Николая Алексеевича**

(Специальность: программирование)

# **Создание Генератора списков оптимальных хэш-тегов для социальной сети “Twitter”**

Консультант

**Самир Ахмед, руководитель программы Microsoft Learn  
Student Ambassadors в Москве.**

Москва  
2021

# Содержание

<b>1</b>	<b>Введение</b>	<b>4</b>
1.1	Twitter как культурное явление . . . . .	4
1.2	Актуальность . . . . .	4
1.3	Анализ предметной области . . . . .	5
1.4	Постановка задачи . . . . .	5
1.5	Целевая аудитория . . . . .	6
1.6	Связанные и похожие задачи . . . . .	6
1.7	Обзор аналогов . . . . .	6
<b>2</b>	<b>Теоретические основы примененных алгоритмов</b>	<b>8</b>
2.1	Введение в нейронные сети . . . . .	8
2.1.1	Машинное обучение . . . . .	8
2.1.2	Исторический экскурс в создание нейронных сетей и глубокого обучения . . . . .	8
2.2	Введение в NLP . . . . .	9
2.3	Прероцессинг данных . . . . .	10
2.4	Векторизация слов . . . . .	11
2.4.1	Меточное кодирование . . . . .	11
2.4.2	Унарное кодирования . . . . .	12
2.4.3	Эмбединги . . . . .	12
2.5	Примененные алгоритмы машинного обучения . . . . .	14
2.5.1	Сверточные сети . . . . .	14
2.5.2	Google Bert . . . . .	15
<b>3</b>	<b>Поиск данных для обучения</b>	<b>17</b>
3.1	Данные специально для Twitter . . . . .	17
3.2	Использованные данные . . . . .	17
3.3	Проблемы с другими датасетами . . . . .	18
<b>4</b>	<b>Инструменты реализации</b>	<b>19</b>
4.1	Модель . . . . .	19
4.2	Сервер и интерфейс . . . . .	19
<b>5</b>	<b>Реализация</b>	<b>21</b>
5.1	Архитектура сети . . . . .	21
5.1.1	BERT-модель с линейными слоями . . . . .	21

---

5.1.2	Сверточная сеть с эмбедингами . . . . .	23
5.2	Сервер . . . . .	23
5.3	Пользовательский интерфейс . . . . .	24
<b>6</b>	<b>Результаты работы</b>	<b>25</b>
6.1	Полученный опыт . . . . .	25
6.2	Готовность продукта . . . . .	25
6.3	Архитектурная и метрическая оценки работы модели . . . . .	25
6.4	Выводы из работы модели . . . . .	26
<b>7</b>	<b>Возможности для дальнейшей разработки</b>	<b>27</b>
<b>8</b>	<b>Пример работы приложения</b>	<b>28</b>
<b>9</b>	<b>Исходный код</b>	<b>28</b>
9.1	Код, примененный для обучения модели . . . . .	28
9.1.1	Bert-модель . . . . .	28
9.1.2	Код не примененной рекуррентной модели . . . . .	38
9.2	Код сервера . . . . .	44
9.2.1	Код для связывания модели с сервером . . . . .	44
9.2.2	Код для обработки запросов сервером со стороны пользова- тельского интерфейса . . . . .	47
9.3	Код пользовательского интерфейса . . . . .	48

# 1 Введение

## 1.1 Twitter как культурное явление

В современном мире существует множество социальных сетей, при том каждая характеризуется целевой аудиторией и, как следствие, набором ключевых функций, созданных специально под целевую аудиторию. Так социальная сеть **Twitter** не столь популярная в нашей стране, но в англоязычном сегменте интернета она является чуть ли не главным медиаресурсом, на просторах интернета так точно. Так бывший президент США Дональд Трамп на своей [ныне заблокированной страничке](#) нередко делал политические заявления. В общем, Twitter можно и нужно считать мощным инструментом для людей, чья работа непосредственно связана с публичностью и работой с аудиторией. Как правило, целью этих людей является раскрутка аккаунта — увеличение количества подписчиков. Что же, наш проект может им помочь.

## 1.2 Актуальность

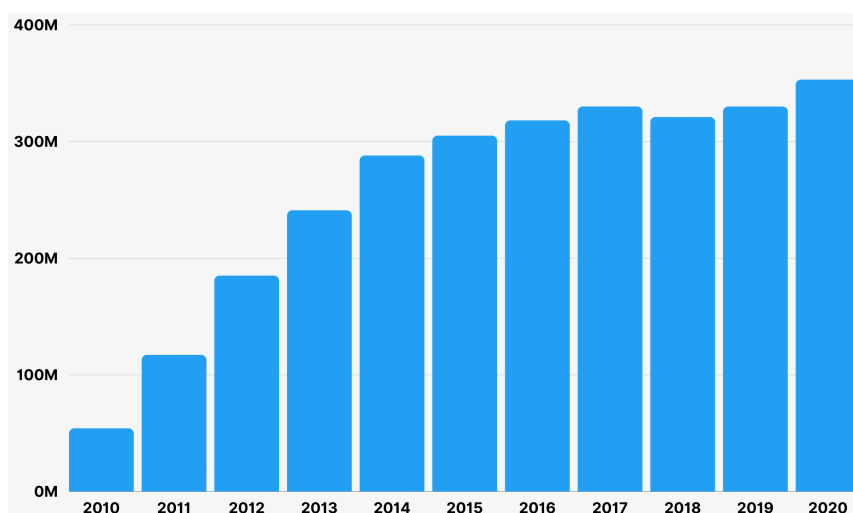


Рис. 1: Количество активных пользователей Twitter по годам

Источник: [backlinko.com](https://backlinko.com)

Наша тема сейчас как-никогда актуальна благодаря следующим факторам:

- Аудитория (активная часть пользователей) Twitter неуклонно растет, график роста представлен на рисунке [1](#)
- Достаточна значимая особенность Twitter заключается в том, что лента пользователей от части формируется из трендовых в данный момент постов, так

называемых *твитов*, причем подбор основывается на интересующем конкретного пользователя тематиках (политика, искусство, экономика, спорт, наука ...)

- В отличие от других социальных сетей, Twitter ориентирован является именно информационным ресурсом, а не развлекательным, его аудитория в среднем старше аудитории других социальных не менее популярных сетей: Facebook и Reddit. Графики распределения пользователей по возрасту представлены для всех трех социальных сетей на рисунках соответственно Twitter (рисунок ), Facebook (рисунок ), Reddit (рисунок )

### 1.3 Анализ предметной области

И так, стоит более конкретно рассказать о законах рекомендательной система контента в Twitter. Ключевую роль в ней выполняют **хештеги**. Идея простая: пользователь помечает свои записи, так называемые *твиты*, специальными ключевыми словами. Обычно, хештеги связаны со смыслом твита, и нужны для поиск твитов на конкретную тему. Так, хештеги играют огромную роль в том, насколько твит окажется популярным, банально сколько людей его увидят. Но не стоит забывать про некоторый нюанс: достаточно часто люди вставляют в твит хештеги, очень косвенно или вовсе не связанные по смыслу с самим текстом твита, лишь потому что данные хештеги сейчас популярны. У данного феномена есть более общее название — *кликбейт*. Кликбейты можно встретить отнюдь не только в Twitter, все дело в их действительной полезности тем, кто при их помощи буквально заманивает пользователей, искавших контент по совершенно другой теме. Пользователям редко нравится<sup>[4]</sup> попадаться на кликбейты, поэтому фильтрация кликбейтов является распространенной задачей и проблемой <sup>[3]</sup>.

### 1.4 Постановка задачи

Наш проект должен помочь людям подбирать к твиту наиболее подходящие хештеги. Более формально, программа должна получать на вход текст готовящего твита и подбирать к нему такие хештеги, что при их вставке в текст предполагаемое количество лайков на твите будет максимальным.

Стоит отметить, что данная задача не подразумевает поиск наиболее близких по теме и смыслу хештегов, так как максимизирующими количество лайков могут оказаться именно вышеописанные кликбейтные хештеги.

## 1.5 Целевая аудитория

Целевой аудиторией нашего проекта являются журналисты и работники *SMM*<sup>1</sup>. Именно им на уровне профессиональной деятельности нужно стремиться охватить максимальную аудиторию.

## 1.6 Связанные и похожие задачи

Поставленная перед нами задача является относительно стандартной задачей области *NLP*<sup>2</sup>. Если вкратце охарактеризовать алгоритмы решения задач в *NLP*, можно выделить 3 этапа их развития:

1. **Символический анализ.** Данные алгоритмы начали применяться примерно в середине прошлого века и был основан на анализе символов, входящих в текст. В основном алгоритмы состояли из частотного анализа и отдельных букв.
2. **Раннее машинное обучение.** С началом эры машинного обучения, оно достаточно быстро было применено и в *NLP*. Так примитивные модели [5], построенные, например, на решающих деревьях, могли выполнять достаточно примитивные задачи бинарной классификации текста, что уже было хорошим результатом по сравнению с Символическим анализом.
3. **Глубокое обучение.** Настоящую революцию в *NLP* произвели нейронные сети в примерно полторы декады назад, сейчас именно они повсеместно используются в данной области, так как показывают несравнимо превосходящие результаты.

Нами было решено также использовать нейронные сети в нашей задаче. Что же, теперь стоит рассказать о машинном обучении в принципе и особенностях его применения в *NLP*.

## 1.7 Обзор аналогов

В целом, все существующие аналоги можно разделить на две непересекающиеся категории: клиенто-ориентированные коммерческие продукты и научные работы.

Аналоги второй категории, как правило, представляют из себя исследовательские статьи об возможности анализа твитов в той или иной области для дальнейшего

---

<sup>1</sup>SMM (Social media marketing) — маркетинг в социальных сетях — комплекс мероприятий по использованию социальных медиа в качестве каналов для продвижения компаний или бренда и решения других бизнес-задач

<sup>2</sup>NLP (Natural Language Processing) — смежная область лингвистики и информатики, связанная с анализом естественных человеческих языков

предсказания хештегов или другой полезной статистики (сентиментальный анализ содержания, предсказывания информативности твита и др.). Такие работы, как правило, ограничиваются статье с выводами и приложенным исходным кодом, при этом находясь в заброшенном состоянии — не рассчитывается, что ими кто-то будет всерьез пользоваться, так как их писали в рамках исследования. Но зато в таких работах часто применяются передовые технологии анализа текста, в этом и смысл проведения такой научной деятельности.

Что же касается первой категории, в интернете существует несколько сайтов ([All-hashtag](#)), [Ingramer](#) и др.) которые предоставляют схожую с разрабатываемой нами программой функциональность. Большинство из них также ориентировано на социальную сеть Twitter. Исходя из описания на сайте, они работают на алгоритмах статического анализа текста, что не может давать хорошие результаты по сравнению с продвинутыми современными алгоритмами глубокого обучения. Возможно, такие сайты просто рекомендуют самые популярные на данный момент хештеги (забегая вперед, наша модель в частности тоже пришла к примерно таким же решениям поставленной перед нами проблемой, о чем более подробно написано на странице [26](#)). С другой стороны, этими сайтами может воспользоваться любой буквально за несколько десятков секунд, что уже лучше, чем просто теоретические статьи. При этом большинство из них существуют на коммерческой основе. Так услугами Ingramer можно пользоваться две недели по подписке за 37\$.

В идеале наш проект должен совместить в себе преимущества описанных выше вариантов: мы планируем и сделать продукт доступным для каждого пользователя через всем привычный пользовательский интерфейс, и использовать поистине современные алгоритмы для качественного подбора хештегов.

## 2 Теоретические основы примененных алгоритмов

### 2.1 Введение в нейронные сети

Поняв, что в нашей задаче нам бок о бок придется работать с прикладными алгоритмами машинного обучения, нами было принято решение изучить данную тему на должном уровне. Стоит рассказать о том, что же из себя представляет машинное обучение в принципе.

#### 2.1.1 Машинное обучение

Машинное обучение — класс методов искусственного интеллекта, характерной чертой которых является не прямое решение задачи, а обучение за счет применения решений множества сходных задач. Существует множество алгоритмов машинного обучения, наиболее известными можно считать решающие деревья, градиентный бустинг, метод ближайших соседей и нейронные сети. Так как в наш проект были непосредственно задействованы только последние, далее речь пойдет именно о них.

#### 2.1.2 Исторический экскурс в создание нейронных сетей и глубокого обучения

Нейронные сети — математические модели, по принципу работы схожие с нейронной структурой мозга. Сильно обобщая, такие модели состоят из Искусственные нейронные сети — это компьютерные программы, использующие несколько уровней узлов (или «нейронов»), работающие параллельно для изучения вещей, распознавания образов и принятия решений подобно человеку. Животный мозг состоит из миллиардов клеток, которые называли нейронами. Отличие нейронов от других клеток состоит в том, что они соединяются между собой и являются накопителями и передатчиками сигналов. Нейрон собирает от соседей, к которым он присоединен дендритами сигнал, преобразовывает его и передает к соседям, с которыми он соединен аксоном. По сути, нейронная сеть в математическом смысле делает то же самое: формально она является графом вычислений, где каждая вершина собирает от соседей, для которых она является непосредственным потомком, значения, преобразует их как-то и присваивает себе это преобразованное значение. Более подробное описание нейронных сетей представлено в [статье](#). Глубокое обучение есть ни что иное как построение сложных (многослойных) нейронных сетей и ее обучение на большом количестве данных.



## 2.2 Введение в NLP

Поскольку NLP подразумевает работу с языками, возникает вопрос, на каком уровне работать с языками: есть тексты, они состоят из предложений, они состоят из слов, они состоят из букв. На протяжении изучения NLP исследователями давались разные ответы на эти вопросы.

- **Анализ текстов на уровне букв.** Вероятно, наиболее тривиальным преобразованием над текстом есть разделение его на слова. Что же, компьютер прекрасно сможет справиться с этой задачей, записав все слова в одной из кодировок и просто проитерировавшись по тексту. Но какую информацию можно извлечь путем такого преобразования? Ну, наверное, можно найти наиболее встречающиеся в текстах буквы, или посчитать, насколько в тексте было больше согласных, чем гласных. Но можно ли извлечь какую-либо информацию о смысле исходного текста? Люди достаточно поняли что нет, нельзя, ведь буквы едва ли связаны со смыслами слов, в которых они были. В качестве аргумента в пользу буквенного анализа можно привести [работу по предугадыванию территориального происхождения имени по буквам в составе этого имени](#), но все же согласно более известным статьям[6], анализ текстов на уровне букв нельзя считать эффективным алгоритмом.
- **N-грамм анализ.** Достаточно необычным может показаться подход разбора исходных предложений на основе так называемых *N-грамм*<sup>3</sup>. Однако, основанные на разбиении исходного текста на N-граммы модели могут отлично работать при работе, например, с сокращенными словами[1]. Все же данный способ остается не особенно изученным и применяемым, хотя стоит отметить, что в последние годы он набирает популярность
- **Анализ текста на уровне слов.** Пожалуй, ближе всего к человеческому способу восприятию текста был бы алгоритм, работает со словами предложение по отдельности, а итоговый результат складывает из полученных характеристик каждого из слов. Более того, именно такие алгоритмы долгое время являлись передовыми в области NLP. При попытках создания подобных алгоритмов возникал достаточно сложный вопрос: а как же машина может понимать слова. В итоге, было придумано векторное представление слов, о котором мы расскажем в разделе [Векторизация слов](#)

---

<sup>3</sup>N-грамма — последовательность из n элементов. С семантической точки зрения, это может быть последовательность звуков, слогов, слов или букв. На практике чаще встречается N-грамма как ряд слов, устойчивые словосочетания называют коллокацией.

- **Анализ текста на уровне предложений.** Наиболее развивающимся на данный момент является подход по изучению сразу кусков из нескольких слов или целых предложений. Это помогает решить основной недостаток словесного анализа, а именно проблемы контекста, подробнее об этом описано в разделе [Векторизация слов](#). Основным инструментом при таком виде анализа являются рекуррентные нейронные сети, которые за счет своей архитектуры могут как бы запоминать прошлые входы, то есть при анализе какой-то части предложения сеть помнит все, что в этом предложении было до этого.

## 2.3 Препроцессинг данных

Особенностью Twitter и большинства других социальных сетей является неформальный стиль общения. Более того, огромное количество сокращений слов, от части грамматически неграмотная, разговорная речь тоже являются вполне приемлемыми в социальных сетях. Также в немалом количестве твитов можно заметить эмодзи, текстовые смайлики, написанные только заглавными буквами слова. Все эти факторы делают язык в Twitter своеобразным диалектом обычного языка, поэтому и подходы к нему нужны специальные. Поэтому первый шаг при работе с Twitter есть *препроцессинг данных* — набор мер по нормализации текста. В основном, это замена сокращений на полные формы слов, устранение опечаток в словах, разбиение предложения на отдельные слова. Этот шаг включает в себя кропотливую работу с регулярными выражениями и другими инструментами символьного анализа текста, которую, к счастью, уже проделали авторы библиотеки NLTK, создав пакет TweetTokenizer, который мы и использовали. Выполняющую препроцессинг данных инструменты принято называть *токенайзерами*. Итого, токенайзеры на вход получают твит в виде строки, а на выход возвращают список из слов и служебных меток, полученных из данного твита. Пример работы токенайзера изображен на рисунке [2](#)



Рис. 2: Пример работы класса TwitterTokenizer.

Источник: документация библиотеки NLTK


## 2.4 Векторизация слов

Кодирование букв алфавита никогда не вызывало проблем: кодировка [ASCII](#) была придумана более полувека назад, с тех ее лишь расширяли и изменяли сами значения. Такой вид кодирования, когда каждому возможному значению сопоставляется натуральное число, называется *меточным кодированием* (англ. Label Encoding). Но с кодированием, или векторизацией слов, все не так однозначно, есть несколько разных алгоритмов.

### 2.4.1 Меточное кодирование

Казалось бы, в чем проблема просто занумеровать все слова в словаре? Более формально, биективно отобразим все слова в последовательные натуральные числа, то есть применим достаточно известный прием категориального кодирования. Проблема в том, что получив набор чисел вместо слов, мы полностью утратили их смысл: теперь вместо слов "собака, кошка, стол, улица у нас есть числа "1, 2, 3, 4". Если до этого предложения "кошка пробежала на улице" "собака бежала на улице" и "стол стоял на улице" можно было достаточно просто сгруппировать на "про животных" и "про мебель то теперь между все три предложения имеют вид "{1, 2, 3}... 4" и смысловая близость слов "собака" и "кошка" оказалась такая же, как "кошка" и "стол". Конечно, человек может расшифровать эти приложения обратно в исходные и легко сказать, какие из них наиболее близки по смыслу, на машина это делать не умеет, и хорошо

бы научить ее понимать смысловую схожесть слов, ведь почти всегда главное при анализе текста — именно его смысл. Неоспоримым плюсом данного подхода является максимальная компактность кодирования — всего лишь одного число на слово.



Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

Рис. 3: Пример унарного кодирования в цветовых категориях.

#### 2.4.2 Унарное кодирования

Альтернативным подходом является унарное кодирования. В таком случае каждому слову присваивается вектор, в котором при длине словаря в  $N$  слов есть ровно  $N-1$  нуль и 1 единица, при том у каждого слова свой уникальный вектор. Очевидным недостатком является пропорциональная квадрату используемая для кодирования память, так как каждому из  $N$  слов нужен вектор из  $N$  чисел. Для лучшего понимания рекомендую изучить рисунок 3. С другой стороны, в некоторых случаях нужно мульти категориальное кодирование: если есть 3 цвета "красный", "желтый" и "зеленый" при этом у некоторого предмета цвет одновременно и красный, и желтый, в таком случае в кодировке данного предмета будет ровно 2 единицы, которые соответствуют красному и желтому цветам, и ровно 1 ноль, отвечающий за зеленый цвет. Все же, тут остается актуальной проблема потери смысла при кодировании. На данном способ базируется на данный момент устаревший метод кодирования слов "Мешок слов" (Bag of Words) [12], при котором каждому из слов словаря непосредственно приписывался унарный вектор, а предложения кодировались как сумма векторов слов данного предложения.

#### 2.4.3 Эмбединги

Описанные выше подходы были (и остаются) хороши для времен (или областей), где количество текстов мало и словарь ограничен, хотя, как мы видели, там тоже есть свои сложности. Но с приходом в нашу жизнь интернета все стало одновременно и сложнее и проще: в доступе появилось великое множество текстов, и эти тексты с

изменяющимся и расширяющимся словарем. С этим надо было что-то делать, а ранее известные модели не могли справиться с таким объемом текстов.

И тогда, как это часто бывает, был предложен выход по принципу “тот, кто нам мешает, тот нам поможет!” А именно, в 2013 году тогда мало кому известный чешский аспирант Томаш Миколов предложил свой подход к word embedding, который он назвал word2vec. Его подход основан на другой важной гипотезе, которую в науке принято называть гипотезой локальности — “слова, которые встречаются в одинаковых окружениях, имеют близкие значения”. Близость в данном случае понимается очень широко, как то, что рядом могут стоять только сочетающиеся слова. Например, для нас привычно словосочетание “заводной будильник”. А сказать “заводной апельсин” мы не можем — эти слова не сочетаются.

Основываясь на этой гипотезе Томаш Миколов предложил новый подход, который не страдал от больших объемов информации, а наоборот выигрывал. Модель, предложенная Миколовым очень проста (и потому так хороша) — мы будем предсказывать вероятность слова по его окружению (контексту). То есть мы будем учить такие вектора слов, чтобы вероятность, присваиваемая моделью слову была близка к вероятности встретить это слово в этом окружении в реальном тексте.

В целом, этот подход называется CBOW — continuous bag of words, continuous потому, что мы скормливаем нашей модели последовательно наборы слов из текста, а BoW потому что порядок слов в контексте не важен.

Также Миколовым сразу был предложен другой подход — прямо противоположный CBOW, который он назвал skip-gram, то есть “словосочетание с пропуском”. Мы пытаемся из данного нам слова угадать его контекст (точнее вектор контекста). В остальном модель не претерпевает изменений.

Что стоит отметить: хотя в модель не заложено явно никакой семантики, а только статистические свойства корпусов текстов, оказывается, что натренированная модель word2vec может улавливать некоторые семантические свойства слов. Классический пример из работы автора[7] представлен на рисунке 4:



Рис. 4: Пример word2vec кодирования. Вектор (Король; Короли) по смыслу очень близок вектору (Королева; Королевы)

Также в пример можно привести достаточно известную формулу, хорошо описывающую смысл такого кодирования:

$$King - Man + Woman = Queen$$

При идеально обученных эмбедингах слов данная формула верна, так как с ними можно взаимодействовать, как с обычными векторами из курса линейной алгебры. Поэтому немного распишем слагаемые просто из человеческого понимания исходных слов:

$$King = Male + Power$$

$$Queen = Female + Power$$

$$Man = Male + Person$$

$$Woman = Female + Person$$

А теперь подставим в исходное уравнение:

$$(Male + Power) - (Male + Person) + (Female + Person) = Female + Power$$

$$Female + Power = Female + Power$$

$$Queen = Queen$$

## 2.5 Примененные алгоритмы машинного обучения

### 2.5.1 Сверточные сети

Сверточные нейронные сети (англ. CNN, Convolutional Neural Network) — алгоритм машинного обучения, позволяющий в небольших потерях информации произвести сильное уменьшение размерности данных. Ниже приведены описания слоев, наличие которых отличает CNN от других алгоритмов машинного обучения. Поскольку перед

нами не стоит цели подробно рассказать о работе CNN, мы лишь кратко опишем основные ее компоненты:

1. Слой свертки (англ. convolutional layer) — это основной блок CNN. Данный слой содержит свой фильтр для каждого канала, ядро свертки которого обрабатывает предыдущий слой по фрагментам — поэлементно домножает значение фильтра на исходное значение фрагмента. После применения вышеописанного над фрагментом, эти действия повторяются над вторым, третьим и т.д. если первый фрагмент начинался с 0-ого элемента, то второй — с 1-ого, и т.д. соответственно.
2. Слой пулинга (англ. Pooling layer) — представляет собой нелинейное уплотнение карты признаков. Пулинг подразделяется на средний (берется среднее значение от  $i$ -ых координат рассматриваемых фрагментов, прошедших convolutional layer) и максимальный (берется максимальное значение среди  $i$ -ых координат рассматриваемых фрагментов). А также глобальный (осуществляется над всеми фрагментами) и не глобальный (осуществляется над группами фрагментов по  $n$  элементов.)

В итоге, сверточная сеть позволяет как бы по фрагментам анализировать входные данные, при этом работы с каждым из фрагментов происходит независимо, а результат этой работы есть результат пулинга над фрагментами. Смысл в этом следующий: допустим, стоит задача поиска маленького кота на огромной панорамной фотографии. За изображения кота на ней отвечает, скажем, меньше одного процента пикселей, поэтому логично каждый из фрагментов по несколько пикселей рассматривать независимо и проверять, не образуют ли они изображение искомого кота. Более сложно понять, какое это имеет отношение к анализу предложений (ведь в предложении все слова играют важную роль). Однако, исследования[10] показывают, что CNN может справляться с анализом текстов достаточно хорошо.

### 2.5.2 Google Bert

*Google BERT* (Bidirectional Encoder Representations from Transformers)[2] — известная предобученная модель от компании Google. Из существующих на данный момент моделей она, а также ее модификации (RoBERTa, SlavicBERT и др.), показывают наилучшие результаты в задачах смыслового анализа текста. Этого удалось достичь благодаря инновационной идее применения так называемых [Трансформеров](#). В целом, архитектура данной модели действительно сложна, поэтому мы использовали ее в черного ящика. А именно использованная нами модификация модели BERT — BERTweet[8] — предназначена специально для того, чтобы твит любой длины превращать в соответствующий ему смысловой вектор. Качество такого преобразования

сильно выше, чем при использовании эмбедингов.



## 3 Поиск данных для обучения

### 3.1 Данные специально для Twitter

При применении любых алгоритмов машинного обучения одной из наиболее насущных и определяющих проблем является поиск актуальных и отражающих действительность данных для обучения. В целом, Twitter является своеобразной кладью примером применения всех языков мира, всех жаргонов этих языков, всех нецензурных единиц этих языков, это делает Twitter наиболее частым местом для сбора данных при исследовании натуральных языков. С другой стороны, Twitter не имеет открытых интерфейсов для разработчиков (API), поэтому единственным решением остается поиск так называемых *датасетов* на популярных сайтах для разработчиков в области анализа данных: [kaggle.com](https://www.kaggle.com) и [github.com](https://github.com).

### 3.2 Используемые данные

Достаточно быстро мы поняли, что обучение имеет смысл производить только на узкоспециализированном датасете. Подробнее данное решение мы описали ниже в разделе 3.3. Также важным оставался вопрос поиска относительно недавно опубликованных данных, так как они все еще могут быть актуальны в реалиях современного Twitter. Кроме того, объем данных никак не мог превышать 100000 единиц для обучения ввиду скромных ресурсов для обучения. В итоге, основную работу по созданию и тестированию сети мы выполняли на [Датасете о выборах в США 2020](#). В данном датасете оказалось порядка 80000 единиц для обучения. Также он интересен тем, что двумя абсолютно лидирующими по использованности хештегами стали фамилии двух основных кандидатов в президенты: “**#Biden**” и “**#Trump**”, при этом большинство твитов были как-бы полярно разделены по политическим убеждениям: лишь в 5% от всех твитов с хотя бы одним из этих хештегов, встретились и первый, и второй. В общем, наша идея при обучении состояла в том, что научить работать абсолютно для всего твиттера мы не сможем просто ввиду нехватки мощностей, поэтому нам стоит сконцентрироваться на одном или нескольких тематических разделах Twitter и обучить сеть для них. Дело в том, что законы языка при описании политики и при описании спорта вряд ли в корнях различаются, поэтому если научиться обрабатывать одну из областей.

### 3.3 Проблемы с другими датасетами

При раннем обсуждении нашего проекта у нас была следующая идея: создать модель, которую можно будет ежедневно, или даже ежечасно, обучать на максимально свежих данных из Twitter. К сожалению, мы пришли к выводу, что в рамках школьного академического проекта данная задумка не может быть полностью реализована по следующим причинам:

1. **Объемы данных в Twitter** чрезмерно большие. Так каждую секунду создается примерно 6000 новых постов [11]. Для обучения модели на таком датасете, в котором содержатся посты за одни сутки, потребуется не менее нескольких десятком часов, так как мы не располагаем большими вычислительными мощностями
2. **Сложность анализа результатов.** При выборе сравнительно небольшого датасета, можно более легко оценивать качество работы модели.

## 4 Инструменты реализации

### 4.1 Модель

- **Язык Python** был использован нами для написания все кода, связанного с моделью. Этот язык де-факто является стандартом в области машинного обучения, а также смежный с ним анализом данных.
- **Среда выполнения Jupyter Notebook** представляет из себя удобный инструмент исполнения кода на языке Python. Позволяет удаленно редактировать файлы, удобно (в формате Markdown)<sup>4</sup>
- **Google Colab** позволяет создавать, редактировать и запускать Jupyter Notebook'и на серверах компании Google с мощными вычислительными элементами (графические и тензорные процессоры).
- **Pytorch** является одной из ведущих библиотек для создания и обучения нейронных сетей. Обладает множеством приятных возможностей, в том числе: тензорное исчисление, модуль автоматического дифференцирования по всем операциям (т.н. AutoGrad), динамический граф вычислений, ООП-совместимая архитектура.
- **Keras** — другая известная библиотека машинного обучения. На практике представляет из себя обертку над самой популярной библиотекой машинного обучения — *TensorFlow*. Keras сильно проще своего старшего собрата, но не уступает ему в возможностях.
- **Numpy, Scipy** — библиотеки для выполнения быстрых математических операциями над многомерными векторами.
- **Pandas** позволяет удобно анализировать датасеты, предоставляет SQL-подобный инструмент создания запросов, оперирует с Numpy-совместимы типами данных.
- **Sklearn, NLTK** — библиотеки для препроцессинга данных.

### 4.2 Сервер и интерфейс

- **Flask** есть *фреймворк*<sup>5</sup> для создания веб-сервисов, подробнее процесс его

<sup>4</sup>Markdown — язык разметки текста. Позволяет писать небольшие документы и комментарии быстрее и проще, чем HTML и L<sup>A</sup>T<sub>E</sub>X.

<sup>5</sup>Фреймворк (от framework — остов, каркас, структура) — программная платформа, облегчающая разработку и объединение разных компонентов большого программного проекта. На практике представляет из себя набор небольших библиотек.

применения описан в разделе [Сервер](#).

- **Vue.js** — передовой фреймворк, ориентированный на создание пользовательского веб-интерфейса. Отличительным свойством является простота, по сравнению с другими известными фреймворками (React и Angular), а также с обычной HTML-версткой без использования каких-либо фреймворков

## 5 Реализация

### 5.1 Архитектура сети

Во время работы над проектом нами было разработано 2 различных архитектуры нейронной сети. Ниже представлено подробное описание каждой из них. В итоге продукте в качестве основной модели нами было решено использовать BERT-модель с линейными слоями, так как они показывала более хорошие результаты и лучше спроектирована (ее проще дообучать и модернизировать). Стоит отметить, что во всех примененных нами архитектурах первым шагом был [Прероцессинг данных](#).

#### 5.1.1 BERT-модель с линейными слоями

Архитектура первой из наших моделей основана на применении сети эмбедингов предложений Google Bert, подробнее о которой вы можете прочитать на странице [15](#). Итого, наша сеть состоит из двух входов. Первым входом сеть принимает вектор действительных чисел длины 768. Далее над ним, чередуясь, применяется 3 линейных слоя и 3 функции активации ReLU, в итоге уменьшая размерность вектора до 64. Вторым же входом сеть принимает закодированные при помощи Унарного кодирования, описанного в разделе [2.4.2](#), хештеги, над которыми тоже производится две пары из линейного слоя и активации ReLU, получая итоговую размерность 64. Далее выходы первого и второго входа склеиваются в один вектор по первой размерности, получается вектор размерности 128. Над ним также есть 2 линейных слоя и функция активация между ними, которые преобразуют этот вектор в действительное число (т.е. вектор размерности один). Этот вектор соответствует ожидаемому количеству лайков на посте. То есть наша сеть по тексту твита и содержащимся в нем хештегам предсказывает, сколько на нем будет лайков. Далее, уже не как часть сети, производится жадный подбор хештегов к введенному пользователем тексту твита: среди всех хештегов выбираются несколько таких, которые в комбинации с твитом “дают” наибольшее количество лайков.

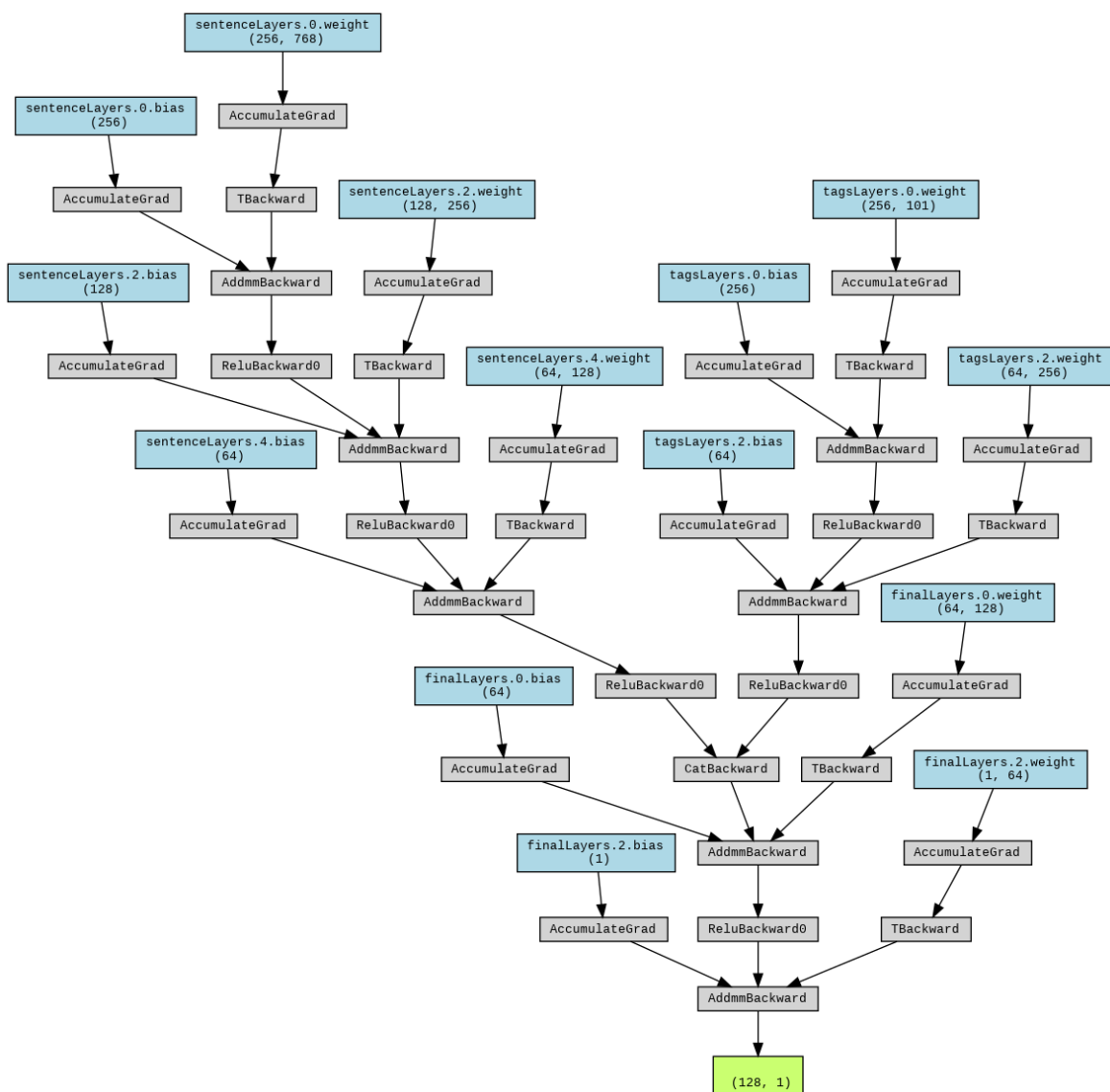


Рис. 5: архитектура bert-модели с линейными слоями.

sentenceLayer — вход для смыслового вектора предложения.

tagLayer — вход для вектора-унарной кодировки использованных тегов.

зеленым блоком обозначен выход — предсказанное сетью количество лайков на посте.

схема сгенерирована при помощи библиотеки [torchviz](#)

### 5.1.2 Сверточная сеть с эмбедингами

Кроме основанной на BERT сети, мы попытались обучить свою собственную сеть с применением алгоритмом свертки (см. страница 14) и линейных слоев. Идеей было применение эмбедингов GloVe[9] для Twitter, кодирование твитов с ее помощью и последующее применение сверточного и нескольких линейных слоев. В результате данная сеть не показала хороших результатов, поэтому в итоговой версии мы использовали именно BERT-основанную модель.

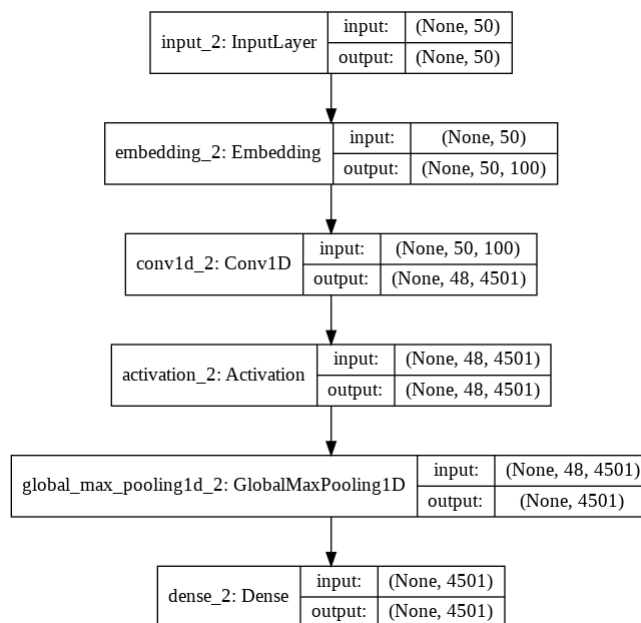


Рис. 6: Архитектура сверточной сети

## 5.2 Сервер

Поскольку перед нами стояла задача разработки приложения, готового для конечного пользователя, использование в качестве интерфейса взаимодействия с программой потоков стандартных ввода и вывода не являлось хорошей идеей. В связи с этим нам было необходимо разработать графический интерфейс. Для связи модели и графического интерфейса нами было решено использовать клиент-серверную архитектуру (см. рисунок 7), выполняя так называемые *REST-запросы*<sup>6</sup>. Поскольку модель мы писали на языке Python, то и сервер мы решили писать на нем. В целом, Python прекрасно подходит для написания малонагруженных и простых приложений, коим и является наш проект, ввиду своей синтаксической лаконичности и простоты использования.

<sup>6</sup>REST (от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределенного приложения в сети. Подробнее: <https://ru.wikipedia.org/wiki/REST>

Также, у нас был опыт работы с фреймворком Flask, и мы использовали именно его для реализации веб-сервиса.

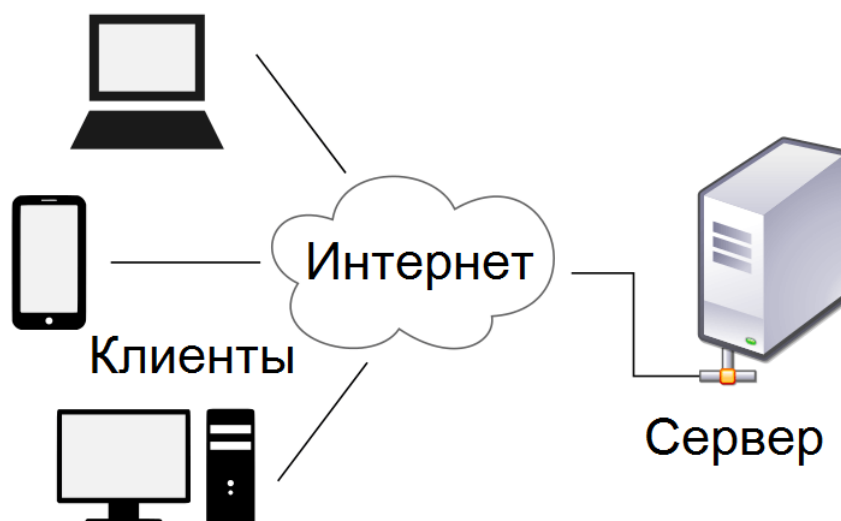


Рис. 7: Упрощенная схема клиент-серверной архитектуры

### 5.3 Пользовательский интерфейс

Как известно, пользователи встречают сервис “по одежке”, то есть по качеству интерфейса. Какую бы точность наша модель не демонстрировала, если пользователю не понравится интерфейс, в ней большого смысла не будет. Нами было принято решение разработать единые для всех устройств веб-интерфейс (с функцией адаптации под дисплеи мобильных устройств и другие нестандартные разрешения), что позволило значительно ускорить процесс разработки, так как не нужно было подстраиваться под особенности платформ — нашим приложением теперь можно пользоваться как со стандартного телефона, так и с умного телевизора или электронной книги. В целом, интерфейс нашего приложения состоит из буквально пары элементов: поле для ввода текста, кнопка на отправку запроса на сервер и поля для показа результата. Имплементация интерфейса базируется на веб-фреймворке для языка Javascript — [Vue.js](#).



## 6 Результаты работы

### 6.1 Полученный опыт

В результате выполнения работы нами был получен бесценный опыт работы с совершенно неизвестными нам до этого алгоритмами анализа данных и машинного обучения. С этой точки зрения, мы считаем проделанную нами работу удачной, так как мы освоили технологии построения пользовательского интерфейса, язык программирования Python и среду выполнения Google Colab, множество использованных нами библиотек, при этом изучив фундаментальную теорию алгоритмов глубокого машинного обучения.

### 6.2 Готовность продукта

В результате работы над проектом нами была выполнена задача создания работоспособного приложения: более того, наш продукт превосходит аналоги, имея субъективно более приятный интерфейс и используя передовые технологии машинного обучения. Недостатком разработанного нами продукта является узкая применимость: адекватно могут предсказываться лишь те теги, которые присутствовали в выбранных нами для обучения датасетах.

### 6.3 Архитектурная и метрическая оценки работы модели

В результате обучения модели мы пришли к следующим результатам

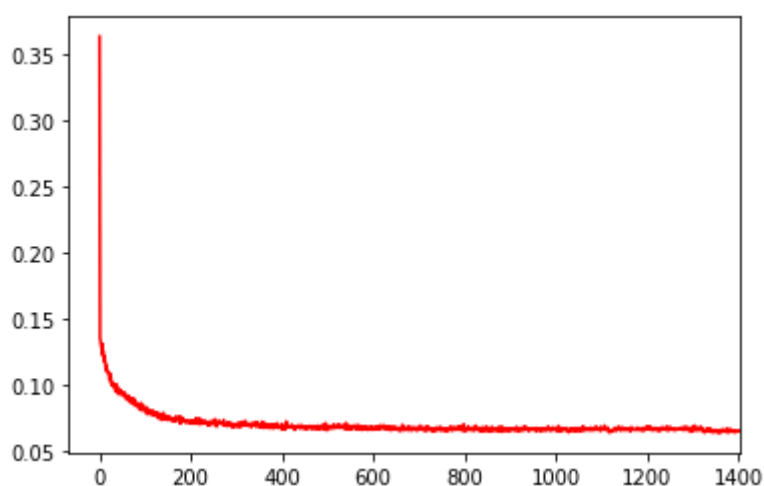


Рис. 8: График функции ошибки на валидационном наборе данных (20% от датасета).

По горизонтальной оси — количество пройденных циклов обучения (т.н. эпох)

По вертикальной оси — значения MSE-функции

## 6.4 Выводы из работы модели

Еще на стадии выбора темы нам было интересно, как же в итоге у нас получаться результаты: можно ли действительно предсказывать хештеги так, чтобы это помогало людям продвигать свой контент. Обучив и протестировав нашу модель, в рамках выбранного нами датасета мы пришли к следующим выводам:

- Практически всегда наиболее подходящими хештегами являются те, которые находятся в тренде. Так, наиболее оптимальным в контексте данных нашего датасета было использование самых трендовых, обобщенных хештегов: “#trump”, “#biden”, “#election2020”. Данные хештеги будут хорошим выбором к абсолютно любым твитам о прошедших в США выборах, из-за своей огромной популярности.
- Более тематические хештеги, такие как “#covid”, “#blm”<sup>7</sup>, “#maga”<sup>8</sup> Эти хештеги могут быть уместны только к более тематическим твитам — не во всяком твите о политике будет уместно поставить хештег об эпидемии коронавируса.

Исходя из выше представленных наблюдений, фактически, работа нашей сети состоит из двух этапов:

1. **Понять смысл нового твита.** За это отвечают использованные нами предобученные эмбединги слов и предложений. В итоге, становится понятно, к какой тематике относиться твит
2. **Найти хештеги из этой тематики.** Поскольку в любом датасете каждый твит может быть более или менее обобщенный, сеть пытается понять, насколько популярность хештега сопоставима с уместностью его использования: хоть хештеги про политику Трампа и пользуются популярностью, в твите про политику Трампа они как раз могут дать хороший результат, а про коронавирусные меры — уже сильно менее хороший; тогда нужно будет использовать специальный хештег про коронавирусные меры или же хештег про выборы в целом — он насколько общий, что к нему имеет отношение и коронавирус, ведь выборы проходят незадолго после конца эпидемии.

<sup>7</sup>BLM (англ. Black Lives Matter) – лозунг проходивших незадолго до и во время выборов массовых протестов. В одно время был самой обсуждаемой темой в Twitter в теме политики.

<sup>8</sup>MAGA (англ. Make America Great Again) — главный лозунг предвыборной кампании Дональда Трампа. Активно употреблялся его сторонниками

## 7 Возможности для дальнейшей разработки

Безусловно, наш проект нельзя назвать выполненным идеально. Среди возможностей для улучшения можно выделить несколько наиболее важных:

- **Обучения на большей выборке данных.** На странице [17](#) более подробно описано, почему мы обучали лишь на тематически-специализированном объеме данных, но, естественно, дообучение на других данных может позволить более широко применять разработанную нами модель
- **Локализация продукта** для наиболее популярных в мире языков
- **Улучшение архитектуры сети.** Не исключено, что через достаточно короткий промежуток времени, архитектура нашей сети устареет, ведь в области машинного обучения чуть ли не каждый год происходят революционные открытия

## 8 Пример работы приложения

## 9 Исходный код

### 9.1 Код, примененный для обучения модели

#### 9.1.1 Bert-модель

```

1 %tensorflow_version 1.x
2 !pip install -U torchviz
3 !pip install ekphrasis torch transformers emoji swifter
4 import torch
5 from transformers import AutoModel, AutoTokenizer
6 from torch.utils.data import Dataset, DataLoader
7 from sklearn.preprocessing import MultiLabelBinarizer
8 !pip install ekphrasis sentence-transformers
9 !pip install -U bert-serving-server # server
10 !pip install -U bert-serving-client # client, independent of
    ↪ `bert-serving-server`
11 from torch import nn
12 import torch.optim as optim
13 import numpy as np
14 import torch.nn.functional as F
15 import emoji
16 from google.colab import drive
17 from collections import Counter
18 import os
19 import matplotlib.pyplot as plt
20 import pandas as pd
21 import swifter
22 from bert_serving.client import BertClient
23 from nltk.tokenize import TweetTokenizer
24 from ekphrasis.classes.preprocessor import TextPreProcessor
25 from ekphrasis.classes.tokenizer import SocialTokenizer
26 from ekphrasis.dicts.emoticons import emoticons
27 !wget
    ↪ https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.z
28 !unzip -o uncased_L-12_H-768_A-12.zip

```

---

```

29 !nohup bert-serving-start -model_dir=./uncased_L-12_H-768_A-12
    ↪ -num_worker=4 > out.file 2>&1 &
30 bertClient = BertClient(check_length=False)
31 #bertTrans = SentenceTransformer('paraphrase-MiniLM-L6-v2')
32 device = "cuda" if torch.cuda.is_available() else "cpu"
33 #device = "cpu"
34 MOST_COMMON = 128
35 tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased",
    ↪ normalization=True, use_fast=False)
36 bertweet = AutoModel.from_pretrained("bert-base-uncased")
37
38
39
40 def _download_recourses():
41     drive.mount('/content/gdrive')
42     os.environ['KAGGLE_CONFIG_DIR'] = "/content/gdrive/My Drive"
43     %cd /content/gdrive/My Drive/Kaggle
44     !yes | kaggle datasets download -d
    ↪ manchunhui/us-election-2020-tweets
45     !unzip -o us-election-2020-tweets.zip
46     !cat hashtag_donaldtrump.csv | tail -n 10
47
48 def _bulk_load_from_files():
49     trump_unparsed_data = pd.read_csv('./hashtag_donaldtrump.csv',
    ↪ lineterminator='\n',
50                                     usecols=['user_screen_name', 'likes',
    ↪ 'tweet'])
51     biden_unparsed_data = pd.read_csv('./hashtag_joe Biden.csv',
    ↪ lineterminator='\n',
52                                     usecols=['user_screen_name', 'likes',
    ↪ 'tweet'])
53     unparsed_data = trump_unparsed_data.append(biden_unparsed_data)
54     data = unparsed_data.iloc[[index for index, value in
    ↪ unparsed_data.tweet.str.contains('#').iteritems() if value]]
55     data
    ↪ =data.reset_index(drop=True).sort_values('user_screen_name').sample(n=5000
    ↪ random_state=42)

```

---

```

56     return data
57
58 def _extract_tags(tweets: pd.Series):
59     return tweets.apply(lambda lstOfTokens: [token for token in
60         ↪ lstOfTokens if token.startswith('#')])
61
62 def _tokenize(row_tweets: pd.Series):
63     print("---TOKENIZING TWEETS NOW---")
64     text_processor = TextPreProcessor(
65         normalize=['url', 'email', 'percent', 'money', 'phone', 'user',
66             ↪ 'time', 'url', 'date', 'number'],
67         annotate={"hashtag", # "allcaps",
68             ↪ "elongated", "repeated",
69             ↪ 'emphasis', 'censored'},
70         fix_html=True, # fix HTML tokens
71         segmenter="twitter",
72         corrector="twitter",
73         ↪ #unpack_hashtags=True, # perform word segmentation on hashtags
74         ↪ unpack_contractions=True, # Unpack contractions (can't -> can
75             ↪ not)
76         ↪ spell_correct_elong=False, # spell correction for elongated
77             ↪ words
78         tokenizer=SocialTokenizer(lowercase=True).tokenize,
79         dicts=[emojis]
80     )
81     return row_tweets.apply(text_processor.pre_process_doc)
82
83 def _filter_bad_tweets(data: pd.DataFrame):
84     data = data[data['tweet'].apply(lambda tweet: 4 < len(tweet) <
85         ↪ 50)].reset_index(drop=True)
86     data = data[data['likes'].apply(lambda likes: likes >
87         ↪ 10)].reset_index(drop=True)
88     return data[data['tags'].apply(lambda lst : len(lst) >
89         ↪ 0)].reset_index(drop=True)
90
91 def _encode_with_bert(tweets: pd.Series):

```

---

```

86     return tweets.apply(lambda tweet: bertClient.encode([tweet],
    ↪     is_tokenized=True))
87
88 def _bertweet_encode(tweets: pd.Series):
89     tokenized = tweets.apply(lambda x: tokenizer.encode(x,
    ↪     add_special_tokens=True, max_length=128))
90     max_len = 0
91     for i in tokenized.values:
92         if len(i) > max_len:
93             max_len = len(i)
94     padded = np.array([i + [0]*(max_len-len(i)) for i in
    ↪     tokenized.values])
95     input_ids = torch.tensor(np.array(padded))
96     with torch.no_grad():
97         last_hidden_states = bertweet(input_ids)
98     return last_hidden_states[0][:,0,:].numpy()
99
100 def _one_hot_encode_tags(tags: pd.Series):
101     all_tags = []
102     tags.apply(all_tags.extend)
103     top_tags, top_counts =
    ↪     zip(*Counter(all_tags).most_common(MOST_COMMON))
104     recognized_tags = set(top_tags)
105     tags = tags.map(lambda prev_tags: [tag if tag in recognized_tags
    ↪     else "OTHER" for tag in prev_tags])
106     recognized_tags.add('OTHER')
107     mlb = MultiLabelBinarizer()
108     mlb.fit([recognized_tags])
109     return tags.apply(lambda lst: mlb.transform([lst]))
110
111 def _normalize_likes_by_author(df: pd.DataFrame):
112     likes_author = df[['user_screen_name', 'likes']]
113     a = likes_author.groupby('user_screen_name').transform(lambda x: x
    ↪     / x.max()).fillna(0)
114     a[a == float('inf')] = 0.5
115     return a
116

```

---

```

117 _download_recourses()
118
119 data = _bulk_load_from_files()
120 print("---DATA EXTRACTED FROM CSV FILES---")
121 data.head()
122
123 data = data[1::2].reset_index(drop=True)
124
125 data = data[data['likes'].apply(lambda likes: likes >
    ↪ 10)].reset_index(drop=True)
126
127 data['row'] = data['tweet']
128 # data['row'] = data.row.apply(emoji.demojize)
129
130 MOST_COMMON = 64
131
132 data.shape
133
134 MOST_COMMON = 64
135
136 data['tweet'] = data['tweet'].apply(emoji.demojize)
137 data['tweet'] = _tokenize(data['tweet'])
138 print("---TWEETS TOKENIZED---")
139 data['tweet'].head(20)
140
141 data.shape
142
143 data = data.sample(15000, random_state=42).reset_index(drop=True)
144
145 encoded_with_bertweet = pd.Series()
146 for i in range(1000, len(data) + 1, 1000):
147     encoded_with_bertweet = pd.concat([encoded_with_bertweet,
    ↪ pd.Series(list(_bertweet_encode(data.row[i-1000:i])))], ignore_index=True)
148     print(i)
149
150 assert(encoded_with_bertweet.shape[0] == 15000)
151

```



---

```

152 data['tags'] = _extract_tags(data['tweet'])
153 data = data.reset_index(drop=True)
154 print("---TAGS EXTRACTED FROM ROW TWEETS---")
155 data['tags'].head(20)
156
157
158
159 data = _filter_bad_tweets(data)
160 data = data.reset_index(drop=True)
161 print("---TWEETS FILTERED FOR WORDS---")
162
163
164
165 data['encoded_tweets'] = _encode_with_bert(data['tweet'])
166 print("---TWEETS ENCODED WITH BERT---")
167
168 data['encoded_tweets'] = encoded_with_bertweet
169
170 data['encoded_tags'] = _one_hot_encode_tags(data['tags'])
171 print("---TAGS ENCODED---")
172
173 data['encoded_tags'][0].sum()
174
175 data['normed_likes'] = _normalize_likes_by_author(data)
176
177
178
179 class PoliticsDataset(Dataset):
180     def __init__(self, dt, applyFunc=None):
181         self.data = dt
182         self.applyFunc = applyFunc
183         self.tags_vector_dimension =
            ↳ self.data['encoded_tags'][0].shape[1]
184
185         self.data['encoded_tags'] =
            ↳ self.data['encoded_tags'].apply(lambda v:
            ↳ toTensor(v).view(-1).to(device))

```

---

```

186     self.data['encoded_tweets'] =
        ↪ self.data['encoded_tweets'].apply(lambda v:
        ↪ toTensor(v).view(-1).to(device))
187 self.data['normed_likes'] =
        ↪ self.data['normed_likes'].apply(lambda v:
        ↪ toTensor(v).view(-1).to(device))
188 print("---INIT FINISHED---")
189 def __len__(self):
190     return self.data.shape[0]
191
192 def __getitem__(self, index):
193     input = (self.data['encoded_tweets'][index],
        ↪ self.data['encoded_tags'][index])
194     target = self.data['normed_likes'][index]
195     return (input, target)
196     #input =
        ↪ (self.applyFunc(self.data['encoded_tweets'][index]).view(-1).to(device),
        ↪ self.applyFunc(self.data['encoded_tags'][index]).view(-1).to(device))
197     #target =
        ↪ self.applyFunc(self.data['normed_likes'][index]).view(-1).to(device)
198
199 def toTensor(x: np.float64):
200     return torch.from_numpy(np.asarray(x)).float()
201
202
203
204 from sklearn.model_selection import train_test_split
205 data_train, data_val = train_test_split(data, test_size=0.2,
        ↪ random_state=42)
206
207 dataset_train = PoliticsDataset(dt=data_train.reset_index(drop=True),
        ↪ applyFunc=toTensor)
208 dataset_val = PoliticsDataset(dt=data_val.reset_index(drop=True),
        ↪ applyFunc=toTensor)
209
210 dataloader = DataLoader(dataset=dataset_train, batch_size=64,
        ↪ shuffle=True)

```

---

```

211 dataloader_val = DataLoader(dataset=dataset_val, batch_size =
    ↪ len(dataset_val), shuffle=True)
212
213
214
215 class LinearBertForSentences(nn.Module):
216
217     def __init__(self):
218         super(LinearBertForSentences, self).__init__()
219         self.sentenceLayers = nn.Sequential(
220             nn.Linear(768, 256),
221             nn.ReLU(),
222             nn.Linear(256, 128),
223             nn.ReLU(),
224             nn.Linear(128, 64),
225             nn.ReLU()
226         )
227         self.tagsLayers = nn.Sequential(
228             nn.Linear(MOST_COMMON + 1, 256),
229             nn.ReLU(),
230             nn.Linear(256, 64),
231             nn.ReLU()
232         )
233         self.finalLayers = nn.Sequential(
234             nn.Linear(128, 64),
235             nn.ReLU(),
236             nn.Linear(64, 1)
237         )
238     def forward(self, batch: list):
239         assert(len(batch) == 2)
240         sentence, tags = batch
241         sentence = self.sentenceLayers(sentence)
242         tags = self.tagsLayers(tags)
243         #catted = torch.cat([sentence, tags, followers], dim=0)
244         return self.finalLayers(torch.cat([sentence, tags], dim=1))
245
246 data.encoded_tags[0]

```

---

```

247
248 running_loss = 0.0
249 model = LinearBertForSentences()
250
251 if device is not None and device != 'cpu':
252     model = model.cuda()
253 lossFunc = nn.MSELoss()
254 optimizer = optim.Adam(model.parameters())
255 batch_processed, losses, eval_losses = [], [], []
256 #Evaluation before any learning (kinda trash expected, cos weights
    ↪ are random values from memory)
257 model.eval()
258 train_item = next(iter(dataloader_val))
259 with torch.no_grad():
260     ls = lossFunc(model(train_item[0]), train_item[1])
261     print(f'EPOCH: {0}\teval_loss: {ls}')
262     eval_losses.append(ls)
263 model.train()
264
265
266 for ep in range(32000):
267     for i, dataitem in enumerate(dataloader, 0):
268         inputs, labels = dataitem
269         optimizer.zero_grad()
270         outputs = model(inputs)
271         loss = lossFunc(outputs, labels)
272         loss.backward()
273         optimizer.step()
274         running_loss += loss.item()
275         if i % 10 == 9:
276             batch_processed.append(i + ep*len(dataloader))
277             losses.append(running_loss)
278             running_loss = 0.0
279 model.eval()
280 with torch.no_grad():
281     ls = lossFunc(model(train_item[0]), train_item[1])
282     print(f'EPOCH: {ep + 1}\teval_loss: {ls}')

```

---

```

283     eval_losses.append(ls)
284     model.train()
285
286     #plt.bar(batch_processed, losses, label="Unlogged", color='r')
287
288     #plt.fill_between(range(len(eval_losses)), eval_losses, color='r')
289     plt.xlabel('Amount of epoch processed')
290     plt.ylabel('Loss value on validation data')
291     plt.plot(range(len(losses)), losses, color='r')
292     plt.show()
293
294     torch.save(model.state_dict(), './model.bin')
295
296     all_tags = []
297     dataloader.dataset.data['tags'].apply(all_tags.extend)
298     top_tags, top_counts =
299     ↪ zip(*Counter(all_tags).most_common(MOST_COMMON))
300     recognized_tags = set(top_tags)
301     recognized_tags
302     recognized_tags.add('OTHER')
303     mlb = MultiLabelBinarizer()
304     mlb.fit([recognized_tags])
305     toTensor(mlb.transform([[next(iter(recognized_tags))]]))
306
307     "while True:"
308     sent = input("Input sentence PLSLSLL:")
309     text_processor = TextPreProcessor(
310         normalize=['url', 'email', 'percent', 'money', 'phone', 'user',
311             'time', 'url', 'date', 'number'],
312         annotate={"hashtag", # "allcaps",
313             "elongated", "repeated",
314             'emphasis', 'censored'},
315         fix_html=True, # fix HTML tokens
316         segmenter="twitter",
317         corrector="twitter",
318         #unpack_hashtags=True, # perform word segmentation on hashtags

```

```

318     unpack_contractions=True, # Unpack contractions (can't -> can
    ↪ not)
319     spell_correct_elong=False, # spell correction for elongated
    ↪ words
320     tokenizer=SocialTokenizer(lowercase=True).tokenize,
321     dicts=[emojis]
322 )
323 sent = toTensor(_bertweet_encode(pd.Series(sent))).to(device)
324 model.eval()
325 res = {}
326 items = []
327 with torch.no_grad():
328     for tag in recognized_tags:
329         nowRes = model([sent,
    ↪ toTensor(mlb.transform([[tag]]).to(device))]
330         res[nowRes.item()] = tag
331         items.append(nowRes.item())
332         items.sort(reverse=True)
333     [(k, res[k]) for k in items]
334 #dict(sorted(res.items(), key=lambda item: item[1]))
335
336 from torchviz import make_dot, make_dot_from_trace
337 graph = make_dot(model(next(iter(dataloader))[0]),
    ↪ params=dict(model.named_parameters()))
338 graph.format = 'png'
339 graph.render()
340 graph.save()
341
342 model(next(iter(dataloader))[0])

```

### 9.1.2 Код не примененной рекуррентной модели

```

1 # -*- coding: utf-8 -*-
2 """torch.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at

```

```

7
8   ↪ https://colab.research.google.com/drive/17eyty6zBcAN9PJ1DQifUtMksfcMiaXPa
9   """
10  !rm ./tweets.csv* # Удалить дубликаты и старые версии файла на всякий
11  ↪ случай
12  !wget
13  ↪ 'https://raw.githubusercontent.com/Nick18899/Dataset/main/tweets.csv'
14  !cat './tweets.csv' | head -n 10
15
16  """# Загрузка датасета в объект ``pandas.DataFrame``"""
17
18  import pandas as pd
19  import numpy as np
20  from keras.preprocessing.text import Tokenizer
21
22  unparsed_data = pd.read_csv('./tweets.csv', usecols=['author',
23  ↪ 'content', 'date_time', 'language', 'number_of_likes',
24  ↪ 'number_of_shares'])
25
26  # data.head()
27  # Отфильтровать твиты на наличие хештегов и английский язык
28  multilanguage_data = unparsed_data.iloc[[index for index, value in
29  ↪ unparsed_data.content.str.contains('#').iteritems() if value]]
30
31  data = unparsed_data.iloc[[index for index, value in
32  ↪ multilanguage_data.language.str.contains('en').iteritems() if
33  ↪ value]]
34
35  assert(data.count().author < multilanguage_data.count().author <
36  ↪ unparsed_data.count().author)
37
38  # Язык везде английский
39  data = data.reset_index(drop=True).drop(columns='language')
40
41  """# Токенизация
42
43  * Каждый твит нужно распарсить на отдельные слова или лексемы
44  * Также отделить каждый тег проблема, чтобы потом проще было
45  ↪ экстрадировать теги из твита
46  """

```

```

34
35 try:
36     from tokenizer import tokenizer
37 except ModuleNotFoundError:
38     !pip install git+https://github.com/erikavaris/tokenizer.git
39     from tokenizer import tokenizer
40 T = tokenizer.TweetTokenizer(preserve_case=False,
    ↪ preserve_handles=False, preserve_hashes=True, regularize=True,
    ↪ preserve_len=False, preserve_emoji=False, preserve_url=False)
41
42 data['content'] = data['content'].apply(lambda tweet:
    ↪ T.tokenize(str(tweet).replace('# ', '#').replace('@ ', '@'))) #
    ↪ некоторые челики ставят пробле между # и текстом самого твита,
    ↪ нам такого не надо
43
44 """# Выделение твитов в отдельный столбец"""
45
46 data['tags'] = data['content'].apply(lambda separatedTweet:
    ↪ list(filter(lambda token: '#' in token, separatedTweet)))
47 data.head()
48
49
50
51 """# Убираем хештеги из самих твитов, так как мы не рассматриваем их,
    ↪ как смысловую часть текста"""
52
53 data['content'] = data['content'].apply(lambda separatedTweet:
    ↪ np.array(list(filter(lambda token: not ('#' in token),
    ↪ separatedTweet))))
54 data['content'].head()
55
56 import gensim.downloader
57 EMBEDDINGS_DIMENSIONS = 50
58 glove_vectors = gensim.downloader.load('glove-twitter-' +
    ↪ str(EMBEDDINGS_DIMENSIONS)) ## Импорт эмбедингов при помощи
    ↪ обертки gensim
59 # Допустимые размерности векторов: 25, 50, 100, 200

```



---

```

60 # Чтобы посмотреть список эмбеддингов:
    ↪ gensim.downloader.info(name_only=True)
61
62 """# Тестим, насколько эмбединги [соответсвуют
    ↪ идеалу](https://www.technologyreview.com/2015/09/17/166211/king-man-woman-queen
63
64 word = 'ball' # Работа со эмбедингами: можно по токену узнать вектор
    ↪ для него, можно по слову узнать его индекс в словаре, что позже
    ↪ надо будет сделать для распарсенных эмбедингов
65 glove_vectors.get_vector(word)
66 ind = glove_vectors.vocab[word].index;
67 glove_vectors.get_vector(glove_vectors.index2word[ind])
68 len(glove_vectors.vocab)
69
70 import numpy as np
71 test = glove_vectors.get_vector('nepo')
72 glove_vectors.similar_by_vector(test)
73
74 """#Убрать из твитов какие-то левые слова , которых нет в
    ↪ эмбедингах"""
75
76 data['content'] = data['content'].apply(lambda tweet:
    ↪ list(filter(lambda word: word in glove_vectors.vocab.keys(),
    ↪ tweet)))
77
78 # glove_vectors.get_keras_embedding
79
80
81
82 """# ``Multilable-Кодирование`` твитов при помощи sklearn
    ↪ ``MultiLabelBinarizer``
83
84 """
85
86
87

```

---

```

88 """Каждому твиту в соответствие поставим вектор с информацией, какие
    ↳ хештеги содержатся в данном твите. Позже это пригодится для
    ↳ решение задачи multilable-классификации тегов"""
89
90 try:
91     from sklearn.preprocessing import MultiLabelBinarizer
92 except ModuleNotFoundError:
93     !pip install sklearn
94     from sklearn.preprocessing import MultiLabelBinarizer
95
96 all_tags = []
97 data['tags'].apply(all_tags.extend)
98 # all_tags = [[all_tags[i], i] for i in range(len(all_tags))]
99 all_tags[:10]
100
101
102
103
104
105 from collections import Counter
106 encoder = MultiLabelBinarizer()
107 categorized_labels_from_tags = encoder.fit_transform(data['tags'])
108 top_tags, top_counts = zip(*Counter(all_tags).most_common(10))
109
110 #sum(values[29])
111 # max_tags_count = max(map(sum, values))
112 # print(max_tags_count)
113
114 import torch # Запишем все в торч
115 torch_tags_categories = torch.tensor(categorized_labels_from_tags)
116 torch_like_tensors = torch.tensor(data['number_of_likes'])
117
118 #torch_tweets_vectors = [[glove_vectors.get_vector(word) for word in
    ↳ tweet] for tweet in data['content']]
119 def get_matrix_from_tweet(tweet):
120     return [[glove_vectors.get_vector(word) for word in tweet]]
121

```

---

```

122 def tweet_to_tensor(tweet):
123     return torch.tensor(get_matrix_from_tweet(tweet))
124
125 tensored_tweets = data['content'].apply(tweet_to_tensor)
126 tensored_tweets[0].shape
127
128
129
130 """# TODO:
131     * Подготовка данных текста
132     * [x] Удалить из текстов твитов слова с хештегами, (они уже
    ↪ лежат в отдельном столбце в dataframe ``data``)
133     * [x] Токенизировать исходные тексты твитов; получить
    ↪ ``vocabulary``
134     * Подготовка данных хештегов
135     * [x] Каждый хештег должен представлять из себя категорию
    ↪ (количество хештегов - мощность категориального вножества)
136     * [x] У каждого твита может быть несколько хештегов
137     * [x] Заюзать ``one-hot`` кодирование в вектор длиной в количество
    ↪ хештегов``, с помощью которого будет описано, какие хештеги
    ↪ встретились в твите (к каким категориям его отнес автор)
138
139     * Сделать обучение....
140     * Рекурент очка
141     * На вход
142         1. Текст твита (матрица из эмбедингов)
143         2. Хештеги в твите (категориальный признак)
144     * На выход
145         1. Число (предполагаемое количество лайков на твите)
146 """
147
148
149
150 """# Deep ~~Dark Fantasy~~ Learning Part"""
151
152 import torch # Загрузка эмбедингов из gensim в torch
153 import torch.nn as nn

```

```

154 weight = torch.FloatTensor(glove_vectors.vectors)
155 embedding = nn.Embedding.from_pretrained(weight, freeze=True)
156 assert(embedding.weight.size()[0] == len(glove_vectors.vocab) and
    ↪ embedding.embedding_dim == glove_vectors.vector_size) # Проверка,
    ↪ что эмбединги загрузились правильно (правильное количество
    ↪ эмбедингов и правильная размерность у каждого)
157 glove_vectors.similar_by_vector(weight[5].cpu().detach().numpy())
158 # glove_vectors.similar_by_vector(weight[0])
159 # glove_vectors.similar_by_vector(test_vector)
160
161 class RNN(nn.Module):
162     def __init__(self, input_size, hidden_size, output_size):
163         super(RNN, self).__init__()
164         self.hidden_size = hidden_size
165         self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
166         self.i2o = nn.Linear(input_size + hidden_size, output_size)
167         self.softmax = nn.LogSoftmax(dim=1)
168
169     def forward(self, input, hidden):
170         combined = torch.cat((input, hidden), 1)
171         hidden = self.i2h(combined)
172         output = self.i2o(combined)
173         output = self.softmax(output)
174         return output, hidden
175
176     def initHidden(self):
177         return torch.zeros(1, self.hidden_size)
178
179 n_hidden = 128
180 # rnn = RNN()

```

## 9.2 Код сервера

### 9.2.1 Код для связывания модели с сервером

```

1 import torch
2 from transformers import AutoModel, AutoTokenizer
3 import pandas as pd

```

---

```

4  from sklearn.preprocessing import MultiLabelBinarizer
5  import numpy as np
6  from torch import nn
7
8  class LinearBertForSentences(nn.Module):
9
10     def __init__(self):
11         super(LinearBertForSentences, self).__init__()
12         self.sentenceLayers = nn.Sequential(
13             nn.Linear(768, 256),
14             nn.ReLU(),
15             nn.Linear(256, 128),
16             nn.ReLU(),
17             nn.Linear(128, 64),
18             nn.ReLU()
19         )
20         self.tagsLayers = nn.Sequential(
21             nn.Linear(64 + 1, 256),
22             nn.ReLU(),
23             nn.Linear(256, 64),
24             nn.ReLU()
25         )
26         self.finalLayers = nn.Sequential(
27             nn.Linear(128, 64),
28             nn.ReLU(),
29             nn.Linear(64, 1)
30         )
31     def forward(self, batch: list):
32         assert(len(batch) == 2)
33         sentence, tags = batch
34         sentence = self.sentenceLayers(sentence)
35         tags = self.tagsLayers(tags)
36         #catted = torch.cat([sentence, tags, followers], dim=0)
37         return self.finalLayers(torch.cat([sentence, tags], dim=1))
38
39 class ModelService():
40     def __init__(self):

```

---

```

41     self.tokenizer =
        ↳ AutoTokenizer.from_pretrained("bert-base-uncased",
        ↳ normalization=True, use_fast=False)
42     self.bertweet =
        ↳ AutoModel.from_pretrained("bert-base-uncased")
43     self.recognized_tags = {'#',
44     '#2020election', '#abdninsecimi', '#america',
        ↳ '#americadecides2020', '#arizona', '#biden',
        ↳ '#biden2020', '#bidenharris', '#bidenharris2020',
        ↳ '#byebyetrump', '#china', '#coronavirus', '#covid',
        ↳ '#covid19', '#debates2020', '#donaldtrump', '#eeuu',
        ↳ '#elecciones', '#elecciones2020', '#eleccioneseuu',
        ↳ '#election', '#election2020', '#election2020results',
        ↳ '#electionday', '#electionday2020', '#electionnight',
        ↳ '#electionresults2020', '#elections', '#elections2020',
        ↳ '#elezioniusa2020', '#florida', '#georgia', '#joebiden',
        ↳ '#maga', '#maga2020', '#michigan', '#nevada', '#obama',
        ↳ '#pennsylvania', '#presidentialdebate2020', '#texas',
        ↳ '#trump', '#trump2020', '#trumpislosing',
        ↳ '#trumpmeltdown', '#trumpvsbiden', '#us', '#usa',
        ↳ '#usa2020', '#usaelection2020', '#usaelections2020',
        ↳ '#uselection', '#uselection2020', '#uselectionresults',
        ↳ '#uselections', '#uselections2020',
        ↳ '#uspresidentialelections2020', '#uswahl2020',
        ↳ '#uswahlen2020', '#vote', '#vote2020', '#wisconsin',
        ↳ 'OTHER'}
45     self.mlb = MultiLabelBinarizer()
46     self.mlb.fit([self.recognized_tags])
47     self.model = LinearBertForSentences()
48
        ↳ self.model.load_state_dict(torch.load('/home/yars/Documents/LastModel.b
        ↳ map_location=torch.device('cpu')))
49     self.model.eval()
50
51
52     def _bertweet_encode(self, tweets: pd.Series):

```

---

```

53     tokenized = tweets.apply(lambda x: self.tokenizer.encode(x,
    ↪     add_special_tokens=True, max_length=128, truncation=True))
54     max_len = 0
55     for i in tokenized.values:
56         if len(i) > max_len:
57             max_len = len(i)
58     padded = np.array([i + [0]*(max_len-len(i)) for i in
    ↪     tokenized.values])
59     input_ids = torch.tensor(np.array(padded))
60     with torch.no_grad():
61         last_hidden_states = self.bertweet(input_ids)
62     return last_hidden_states[0][:,0,:].numpy()
63
64
65     def toTensor(self, x: np.float64):
66         return torch.from_numpy(np.asarray(x)).float()
67
68
69     def get_tags(self, sentence: str, count: int):
70         encoded_sentence =
71         ↪ self.toTensor(self._bertweet_encode(pd.Series(sentence)))
72         self.model.eval()
73         res = {}
74         items = []
75         with torch.no_grad():
76             for tag in self.recognized_tags:
77                 nowRes = self.model([encoded_sentence,
78                 ↪ self.toTensor(self.mlb.transform([[tag]]))])
79                 res[nowRes.item()] = tag
80                 items.append(nowRes.item())
81         items.sort(reverse=True)
82         return [res[k] for k in items[5:min(5+count, len(items))]]

```

### 9.2.2 Код для обработки запросов сервером со стороны пользовательского интерфейса

```

1 from flask_cors import CORS, cross_origin
2 from flask import Flask, request, jsonify

```

```

3 import json
4 from modelService import ModelService
5
6 app = Flask(__name__)
7 model = ModelService()
8 cors = CORS(app)
9 app.config['CORS_HEADERS'] = 'Content-Type'
10
11 @app.route("/gettingTags", methods=['GET', 'POST', 'DELETE', 'PUT'])
12 @cross_origin()
13 def gettingTags():
14     text = request.get_json()
15     print(text['value'])
16     result = model.get_tags(text['value'], int(text['number']))
17     return jsonify(hashtags=result)
18
19
20 def main():
21     app.run(port=5050, debug=True)
22
23
24
25 if __name__ == "__main__":
26     main()

```

### 9.3 Код пользовательского интерфейса

```

1 <template>
2   <div id="app">
3     <section class="container-fluid bg">
4       <section class = "row justify-content-center">
5         <section class = "col-12 col-sm-6 col-md-3">
6           <form class = "form-container">
7             <div class = "form-group">
8               <label>Enter some text here:</label>
9             </div>
10            <div class = "form-group">

```



---

```

11     <BaseInput :value="inputString" type="string"
      ↪   v-model="inputString"/>
12 </div>
13 <div class = "form-group">
14     <BaseNumberInput :value="inputNumber" type="number"
      ↪   v-model="inputNumber"/>
15 </div>
16     <BaseButton @click="sendInputString">Analyze</BaseButton>
17 <br>
18 <label style="margin-top: 10px">Corresponding hashtags for your
      ↪   text:</label>
19 <p style="white-space: pre-line">{{tags}}</p>
20 </form>
21 </section>
22 </section>
23 </section>
24 </div>
25 </template>
26
27 <script>
28 import BaseInput from "@components/BaseInput.vue";
29 import BaseButton from "@components/BaseButton.vue";
30 import BaseNumberInput from "@components/BaseNumberInput";
31
32 export default {
33   name: 'app',
34   components: {BaseNumberInput, BaseButton, BaseInput},
35   data(){
36     return {
37       inputString: "",
38       inputNumber: 1,
39       outputString: "",
40       tags: " "
41     }
42   },
43   methods: {
44     sendInputString: async function() {

```

---

```

45     console.log("alpha")
46     const resp = await
47         ↪ fetch('http://localhost:5050/gettingTags',{
48         method: "POST",
49         body: JSON.stringify({
50             value: this.inputString,
51             number: this.inputNumber
52         })),
53         headers: {
54             "Content-Type": 'application/json'
55         }
56     const jsn = await resp.json()
57     this.tags = jsn.hashtags
58 }
59 }
60
61 }
62 </script>
63
64
65 <style scoped>
66     #app { font-family: Roboto, Helvetica, Arial, sans-serif; }
67 </style>

1  import {createApp} from "vue";
2  import App from "./App.vue";
3  import 'mdb-vue-ui-kit/css/mdb.min.css'
4  import './assets/sass/main.scss';
5  import ElementPlus from 'element-plus';
6  import 'element-plus/lib/theme-chalk/index.css';
7
8
9  createApp(App)
10     .use(ElementPlus)
11     .mount("#app");

```

## Список литературы

- [1] Steffen Bickel, Peter Haider и Tobias Scheffer. “Predicting Sentences using N-Gram Language Models.” В: янв. 2005. DOI: [10.3115/1220575.1220600](https://doi.org/10.3115/1220575.1220600).
- [2] Jacob Devlin и др. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. В: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, июнь 2019, с. 4171—4186. DOI: [10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423). URL: <https://www.aclweb.org/anthology/N19-1423>.
- [3] Indurthi. “Predicting Clickbait Strength in Online Social Media”. В: *Proceedings of the 28th International Conference on Computational Linguistics*. Barcelona, Spain (Online): International Committee on Computational Linguistics, дек. 2020, с. 4835—4846.
- [4] Anil Karaca. “News Readers’ Perception of Clickbait News”. Master’s Thesis. Kadir Has University. School of Graduate Studies, 2019.
- [5] Roland Kuhn и Renato De Mori. “The application of semantic classification trees to natural language understanding”. В: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* (июнь 1995), с. 449 —460.
- [6] Dongyun Liang, Weiran Xu и Yingge Zhao. “Combining Word-Level and Character-Level Representations for Relation Classification of Informal Text”. В: *Proceedings of the 2nd Workshop on Representation Learning for NLP*. Vancouver, Canada: Association for Computational Linguistics, авг. 2017, с. 43—47. DOI: [10.18653/v1/W17-2606](https://doi.org/10.18653/v1/W17-2606). URL: <https://www.aclweb.org/anthology/W17-2606>.
- [7] Tomas Mikolov и др. “Efficient Estimation of Word Representations in Vector Space”. В: янв. 2013, с. 1—12.
- [8] Dat Quoc Nguyen, Thanh Vu и Anh Tuan Nguyen. “BERTweet: A pre-trained language model for English Tweets”. В: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2020, с. 9—14.
- [9] Jeffrey Pennington, Richard Socher и Christopher Manning. “GloVe: Global Vectors for Word Representation”. В: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, окт. 2014, с. 1532—1543. DOI: [10.3115/v1/D14-1162](https://doi.org/10.3115/v1/D14-1162). URL: <https://www.aclweb.org/anthology/D14-1162>.

- [10] Morteza Rohanian и др. *Convolutional Neural Networks for Sentiment Analysis in Persian Social Media*. 2020. arXiv: [2002.06233](https://arxiv.org/abs/2002.06233) [cs.SI].
- [11] Kit Smith. *60 Incredible and Interesting Twitter Stats and Statistics*. <https://www.brandwatch.com>. 2020.
- [12] Yin Zhang, Rong Jin и Zhi-Hua Zhou. “Understanding bag-of-words model: A statistical framework”. В: *International Journal of Machine Learning and Cybernetics* 1 (дек. 2010), с. 43—52. DOI: [10.1007/s13042-010-0001-0](https://doi.org/10.1007/s13042-010-0001-0).