Оглавление

Введение

С каждым днем количество информации в мире становится все больше и больше. Актуальная задача сейчас не просто обрабатывать эту информацию, а научить компьютер понимать ее. Понимание естественного языка, одна из таких задач. Также популярность набирает задача реализации чат-ботов, которые могут использоваться в различных сферах. Для решения поставленных задачи в данной курсовой работе будет реализован в мессенджере Telegram интеллектуальный чат-бот, который сможет понимать текстовые сообщения от пользователей и на основе этих сообщений выдавать ответы.

Глава 1. Теоретическая часть

1.1 NLP

NLP (Natural Language Processing, обработка естественного языка) — это направление в машинном обучении, посвященное распознаванию, генерации и обработке устной и письменной человеческой речи. Находится на стыке дисциплин искусственного интеллекта и лингвистики.

1.2 Применение NLP

Инженеры-программисты разрабатывают механизмы, позволяющие взаимодействовать компьютерам и людям посредством естественного языка. Благодаря NLP компьютеры могут читать, интерпретировать, понимать человеческий язык, а также выдавать ответные результаты. Как правило, обработка основана на уровне интеллекта машины, расшифровывающего сообщения человека в значимую для нее информацию.

Приложения NLP окружают нас повсюду. Это поиск в Google или Яндексе, машинный перевод, чат-боты, виртуальные ассистенты вроде Siri, Алисы, Салюта от Сбера и пр. NLP применяется в digital-рекламе, сфере безопасности и многих других.

1.3 Как устроена обработка языков

Раньше алгоритмам прописывали набор реакций на определенные слова и фразы, а для поиска использовалось сравнение. Это не распознавание и понимание текста, а реагирование на введенный набор символов. Такой алгоритм не смог бы увидеть разницы между столовой ложкой и школьной столовой.

NLP — другой подход. Алгоритмы обучают не только словам и их значениям, но и структуре фраз, внутренней логике языка, пониманию контекста. Чтобы понять, к чему относится слово «он» в предложении «человек носил костюм, и он был синий», машина должна иметь представление о свойствах понятий «человек» и «костюм». Чтобы научить этому компьютер, специалисты используют алгоритмы машинного обучения и методы анализа языка из фундаментальной лингвистики.

1.4 Задачи NLP

NLP решает множество задач обработки естественного языка, например: распознавание речи, обработка текста, извлечение информации, анализ информации, генерация текста и речи, автоматические пересказ, машинный перевод.

В данной курсовой работе реализовано решение задачи обработки текста. В современном мире эта задача является весьма актуальной, например, помощников в банковских приложениях реализуют через чат-ботов, которые могут обрабатывать запросы, полученные в вольном формате, и выдавать на них ответы. Чат-боты позволяет разгрузить сотрудников от простых и рутинных вопросов, в связи с чем вырастает и качество обслуживания.

1.5 Обработка текста

Нельзя взять текст, введенный пользователем, и отдать его алгоритму. Эти текстовые данные нужно сначала «приготовить» или преобразовать в вид, доступный для восприятия компьютером. Процесс готовки данных называется препроцессинг и включает в себя несколько этапов. Рассмотрим эти этапы подробнее.

1.5.1 Токенизация по предложениям и по словам

Токенизация (иногда – сегментация) по предложениям – это процесс разделения письменного языка на предложения-компоненты. В английском, русском и некоторых других языках мы можем вычленять предложение каждый раз, когда находим определенный знак пунктуации – точку. Но точка используется не только для конца предложений, она также применяется при сокращении слов. В этом случае, чтобы предотвратить неправильную расстановку границ предложений, сильно поможет таблица сокращений.

Токенизация (иногда – сегментация) по словам – это процесс разделения предложений на слова-компоненты. В английском, русском и многих других языках, использующих ту или иную версию латинского алфавита, пробел – это неплохой разделитель слов. Но и здесь не все просто так просто, потому что

могут попасться такие слова, которые являются одним словом, но при этому будут писаться через пробел.

1.5.2 Лемметация и стемминг текста

Обычно тексты содержат разные грамматические формы одного и того же слова, а также могут встречаться однокоренные слова. Лемматизация и стемминг преследуют цель привести все встречающиеся словоформы к одной, нормальной словарной форме. Пример приведения разных словоформ к одной: dog, dogs, dog’s приводятся к словоформе dog.

Лемматизация и стемминг – это частные случаи нормализации.

Стемминг – это грубый эвристический процесс, который отрезает «лишнее» от корня слов, часто это приводит к потере словообразовательных суффиксов.

Лемматизация – это более тонкий процесс, который использует словарь и морфологический анализ, чтобы в итоге привести слово к его канонической форме – лемме.

Отличие этих операций в том, что стеммер действует без знания контекста и, соответственно, не понимает разницу между словами, которые имеют разный смысл в зависимости от части речи. Однако у стеммеров есть и свои преимущества: их проще внедрить и они работают быстрее. Плюс, более низкая «аккуратность» может не иметь значения в некоторых случаях.

Примеры работы лемматизации и стемминга:

1. Слово good – это лемма для слова better. Стеммер не увидит эту связь, так как здесь нужно сверяться со словарем.
2. Слово play – это базовая форма слова playing. Тут справятся и стемминг, и лемматизация.

1.5.3 Стоп-слова

Стоп-слова – это слова, которые выкидываются из текста до/после обработки текста. При применении машинного обучения к текстам, такие слова могут добавлять много шума, поэтому необходимо избавляться от нерелевантных слов.

В качестве стоп-слова обычно используются артикли, междометия, союзы и т.д., в общем слова, которые не несут смысловой нагрузки. Не существует универсального списка стоп-слов, все зависит от конкретного случая.

1.5.4 Мешок слов

Алгоритмы машинного обучения не могут напрямую работать с сырым текстом, поэтому необходимо конвертировать тексты в наборы цифр (векторы). Это называется извлечением признаков.

Мешок слов – это популярная и простая техника извлечения признаков, используемая при работе с текстом. Она описывает вхождения каждого слова в текст.

Чтобы использовать модель нужно:

1. Определить словарь известных слов (токенов);
2. Выбрать степень присутствия известных слов.

Любая информация о порядке или структуре слов игнорируется. Вот почему это называется мешком слов. Эта модель пытается понять, встречается ли знакомое слово в документе, но не знает, где именно оно встречается.

Интуиция подсказывает, что схожие документы имеют схожее содержимое. Также, благодаря содержимому, можно узнать кое-что о смысле документа.

1.6 Чат-боты

Чат-бот – это виртуальный собеседник, программа, которая может решать типовые задачи. Чат-боты используются в разных сферах, например, в службе доставки, службе поддержки, банкинге, поддержание легких бесед (так называемые small talk боты) и не только. Использование чат-ботов увеличивает прибыль компаний и положительно сказывается на пользователях, потому что для получения ответа на свой вопрос им нужно затратить в разы меньше времени.

Один из примеров использования чат-ботов – это использование их в службе поддержки. Чат-боты доступны 24/7 без перерывов на обед, они

позволяют разгрузить операторов от рутинных и банальных вопросов, оставляя им возможность работать над вопросами, требующими вмешательства человека. В случае если чат-бот не сможет справиться с ответом, то он по крайней мере соберет первичные данные, например когда был сделан заказ, номер заказа, проблемный вопрос или определит тип проблемы, чтобы направить вопрос к квалифицированному сотруднику. Чем качественнее будет реализован чат-бот, тем больше пользователей он сможет обслужить, а это значит, что компания сможет сэкономить деньги на найме операторов и подготовки рабочих мест для них.

Чат-боты, как правило, реализуются в стандартных мессенджерах, к которым люди давно уже привыкли, например WhatssApp, Viber, Telegram. Использование мессенджера позволяет не тратить силы на разработку приложения, а сконцентрироваться на разработке самого чат бота. В курсовой работе чат-бот будет реализован для мессенджера Telegram.

1.7 Выбор языка программирования

С поставленной задачей реализовать интеллектуального чат-бота лучше всего справится язык Python. Python – это бесспорный лидер среди языков программирования ИИ. Для это языка существует большое количество различных готовых к использованию библиотек, которые облегчают и ускоряют написание кода.

1.8 Выбор библиотек

Для работы потребуются следующие библиотеки: pyTelegramBotAPI, NLTK, PyTorch, NumPy.

Библиотека pyTelegramBotAPI понадобится для создания бота в мессенджере Telegram.

Библиотека NLTK является одной их лучших библиотек Python для решения задачи обработки естественного языка. Она предоставляет множество полезных функций для обработки текстов, включая токенизацию, выделение корней, синтаксический анализ и многие другие функции, необходимые для создания моделей машинного обучения.

Библиотека PyTorch – современная библиотека глубокого обучения, обеспечивающая тензорные вычисления с GPU-ускорением, подобно NumPy. PyTorch предлагает насыщенный API для решения прикладных задач, связанных с нейронными сетями. PyTorch отличается от других фреймворков машинного обучения тем, что здесь не используются статические расчетные графы – определяемые заранее, сразу и окончательно – как в TensorFlow, Caffe2 или MXNet. Напротив, расчетные графы в PyTorch динамические и определяются на лету. Таким образом, при каждом вызове слоев в модели PyTorch динамически определяется новый расчетный граф.

Библиотека NumPy добавляет поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами.

1.9 Данные для обучения модели

Точность работы модели напрямую зависит от качества данных для обучения. Лучшие команды мира, тратят достаточно много времени на улучшение своих тренировочных данных. Нередко компании предоставляют код своих моделей машинного обучения в открытый доступ, но данные, на которых модели обучались, часто остаются в тайне. В интернете есть довольно много различных датасетов, но не всегда они подходят на все 100%.

В работе используется датасет «Small talk: Intent Classification data»[1], который был взят с сайта www.kaggle.com. Этот датасет пришлось доработать, так как он содержал только названия классов (tag) и запросы пользователя (patterns), а блока возможных ответов (responses) не было, поэтому возможные ответы пришлось придумывать самостоятельно.

[1] Small talk: Intent Classification data : сайт. – URL: https://www.kaggle.com/datasets/salmanfaroz/small-talk-intent-classification-data (дата обращения: 01.11.2022)

Глава 2. Практическая реализация

2.1 Структура проекта

Реализовывать все функции в одном файле – очень неэффективно, поэтому весь проект реализован в нескольких файлах со следующими названиями: «IntelligentChatBotTelegram.py», «settings.py», «NLTK_utils.py», «model.py», «train.py», «console_chat_with_Jarvis.py». Каждый их этих файлов имеет определенную зависимость между собой, но чтобы ее показать, необходимо еще обозначить папку «source». В ней находятся файлы «intents.json», «data.pth» и картинки, которые используются для мессенджера Telegram. Структура проекта приведена на рисунке 2.1.

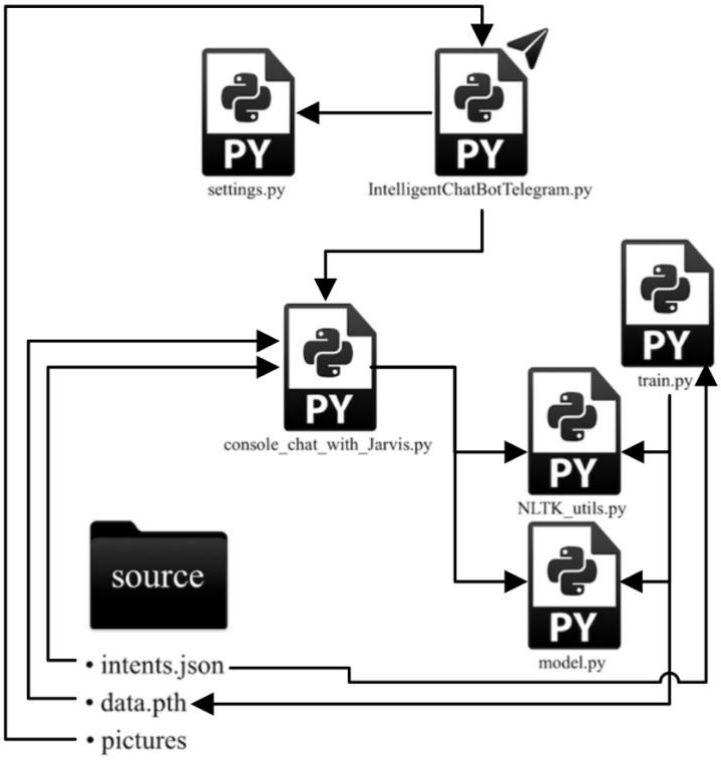

Рисунок 2.1 – Структура проекта

Обзор содержимого файлов стоит начать с «NLTK_utils.py» и «model.py». В первом файле реализованы функции токенизации (tokenize), стемминга с понижением регистра (stem) и мешка слов (bag_of_words). Во втором файле реализована модель нейронной сети прямого распространения (feed forward) с двумя скрытыми слоями. Схема нейронной сети изображена на рисунке 2.2.

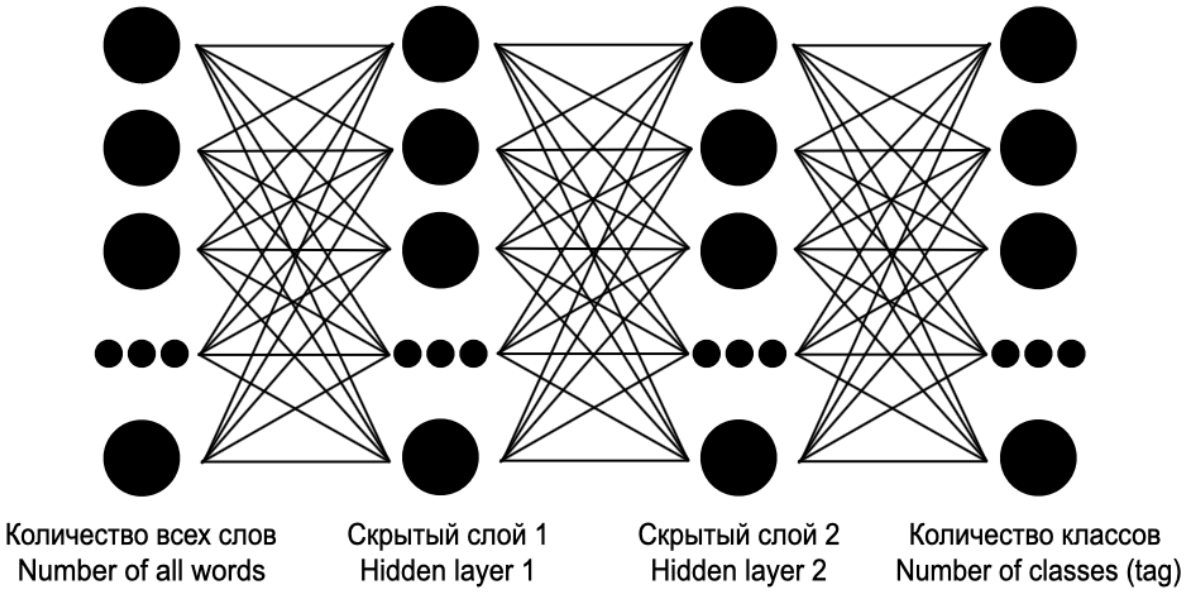

Рисунок 2.2 – Схема нейронной сети

Файл «intents.json» - содержит датасет. Структура датасета отображена в листинге 2.1. Датасет состоит из множества блоков, каждому блоку принадлежит «tag», «patterns» и «responses». Обобщение блока хранится в «tag», в «patterns» хранится то, что может написать пользователь, а в «responses» хранятся возможные ответы, при использовании данного блока. Чем больше будет «tag», тем на большее количество тем, бот будет понимать. Чем больше будет «patterns», тем точнее модель будет определять тему. Чем больше будет «responses», тем более разнообразнее будут ответы бота.

Листинг 2.1 – Структура датасета

```
{
    "intents": [
        {
            "tag": "",
            "patterns": [
                "",
                "",
                "",
                "",
                ""
            ],
            "responses": [
                "",
                "",
                "",
                "",
                ""
            ]
        }
        {
```

```
            "tag": "",
            "patterns": [
                "",
                "",
                ""
            ],
            "responses": [
                "",
                "",
                ""
            ]
        }
    ]
}
```

Файл «train.py» - как видно из названия, реализует обучение модели. В этом файле реализовано чтение данных из датасета, «приготовление» данных для обучения, создание модели с заданными гиперпараметрами, обучение модели и запись результата обучения файл «data.pth», для дальнейшего использования. Обучение на большом количестве данных занимает много времени, поэтому выгодно сохранить результат обучения для дальнейшего его использования в других файлах. При изменении файла «intents.json» или каких-то других манипуляциях с моделью ее нужно обучать заново, т.е. запускать файл «train.py»

После обучения модели можно воспользоваться файлами «console_chat_with_Jarvis.py» и «IntelligentChatBotTelegram.py».

В файле «console_chat_with_Jarvis.py» реализован интеллектуальный чат-бот, здесь создается обученная модель (благодаря файлу «data.pth») принимаются сообщения от пользователя и выдаются соответствующие ответы, которые хранятся в датасете «intents.json».

В файле «IntelligentChatBotTelegram.py» содержится реализация Telegram бота. Сообщения, полученные в мессенджере Telegram, будут отправляться к интеллектуальному боту («console_chat_with_Jarvis.py»), там обрабатываться и результат обработки будет возвращен назад в мессенджер и доставлен пользователю. Также файл «IntelligentChatBotTelegram.py» связан с «settings.py», в котором хранится API ключ (токен) бота Telegram. Кроме ключа в этом файле больше ничего нет. Это сделано для осуществления безопасности. Например, при размещении кода в публичном репозитории

GitHub, не сохранив API ключ в безопасности, мы фактически перестаем быть единственным владельцем бота.

2.2 NLP Preprocessing Pipeline

Подготовка данных к работе – весьма важный процесс. Все необходимые действия, которые необходимо совершить с полученной строкой, показаны на рисунке 2.3. Для реализации этого Preprocessing Pipeline понадобится библиотека NLTK, в которой уже реализованы данные функции.

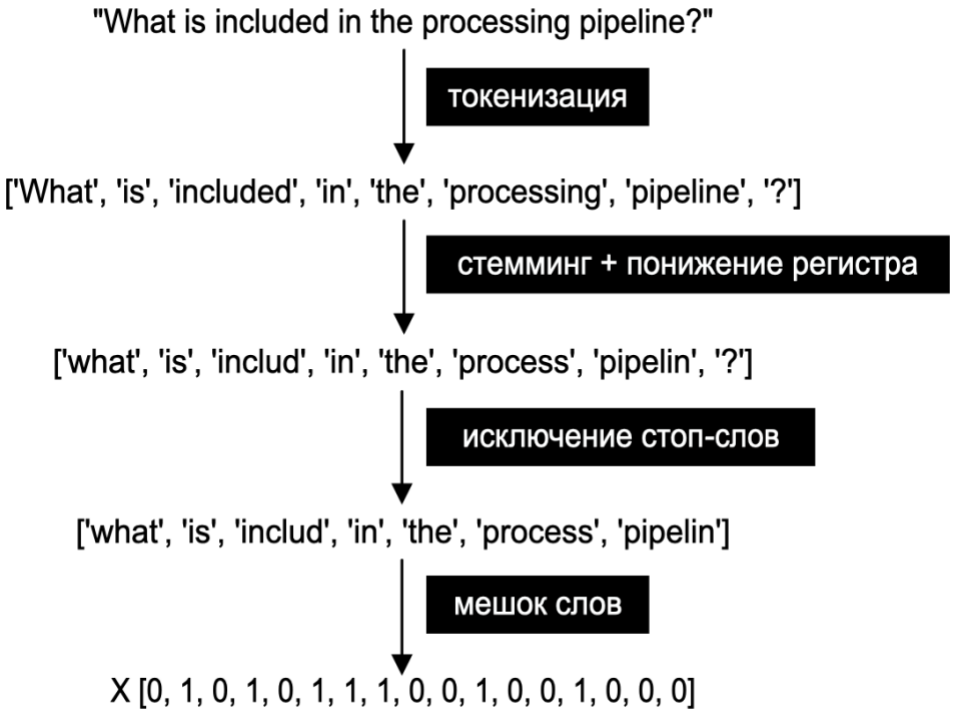

Рисунок 2.3 – NLP Preprocessing Pipeline

2.2.1 Токенизация по словам

Обработка начинается с токенизации. Важным фактом является то, что для работы функции токенизации из библиотеки NLTK должен быть скачен модуль «punkt». Для этого при первом запуске необходимо выполнить строчку кода «nltk.download('punkt')», при последующих запусках эту строчку нужно закомментировать, т.к. необходимый пакет скачен и перекачивать его каждый раз не имеет смысла. Функция, в который реализуется токенизация, показана в листинге 2.2.

Листинг 2.2 – Реализация функции токенизации

```
def tokenize(sentence):
    """
```

```
    Разделение предложения на отдельные слова/токены
    Токенами могут быть: слова, пунктуация, цифры
    """
    return nltk.word_tokenize(sentence)
```

Очевидно преимущество использования библиотек. Нет необходимости задумываться над реализацией алгоритма, потому что в библиотеке он уже реализован, остается только применить его. В итоге реализация токенизации занимает всего 1 строчку кода. Процесс написания программы становится похожий на сборку конструктора из готовых компонент, это здорово ускоряет написание кода. Пример работы алгоритма токенизации показан на рисунке 2.4.

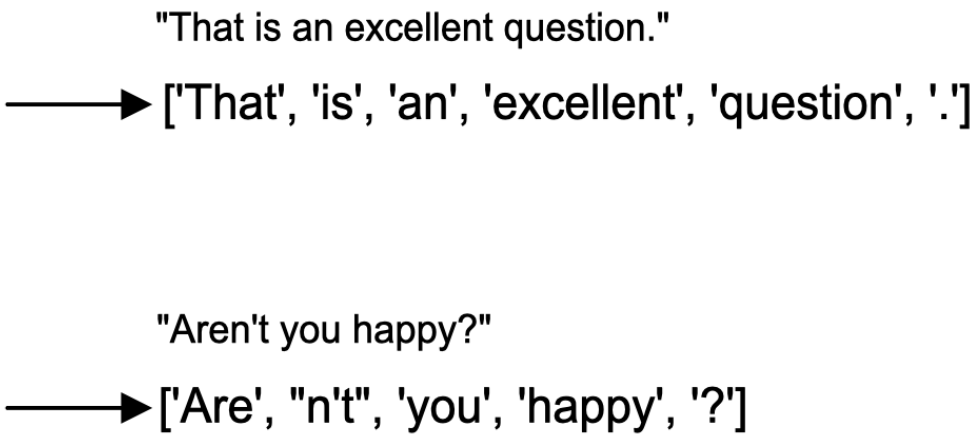

Рисунок 2.4 – Пример работы токенизации

2.2.2 Стемминг, понижение регистра, исключение стоп-слов

Следующий этап подготовки включает в себя сразу несколько действий. Реализация функции стемминга выполнена при помощи все той же NLTK библиотеки, в которой присутствует множество различных исполнений алгоритма стемминга. В проекте будет использован «Porter Stemmer», который был реализован Мартином Портером в 1980 году, но с современными улучшениями от команды NLTK. В функции стемминга сразу же будет применена функция понижения регистра. Реализация в проекте показана в листинге 2.3.

Листинг 2.3 – Реализация функции стемминга с понижением регистра

```
stemmer = PorterStemmer()
def stem(word):
    """
    stemming = ищет корневую форму слова
```

```
    Пример:
    words = ["organize", "organizes", "organizing"]
    words = [stem(w) for w in words]
    -> ["organ", "organ", "organ"]
    """
    return stemmer.stem(word.lower()) # К слову будет применен Стеммер и
слово будет преобразовано в нижний регистр при необходимости
```

Стоп-слова, которые определены в массиве «stop_words», будут исключаться во время обучения при создании вектора «all_words».

Вектор «all_words» получается следующим образом: применяется токенизация ко всем «patterns» из датасета. Получится список всех слов, но еще не обработанный до конца. К этому списку применяется стемминг с понижением регистра и исключением стоп-слов. Также стоит отметить, что повторные слова не включаются в вектор всех слов. Реализация получения вектора всех слов («all_words») показана в листинге 2.4.

Листинг 2.4 – Реализация получения вектора всех слов («all_words»)

```
all_words = [stem(w) for w in all_words if w not in stop_words]
all_words = sorted(set(all_words))
```

Первая строчка вызывает функцию стемминга с понижением регистра при необходимости, если слово не является стоп-словом. А во второй строчке, полученный новый массив всех слов сортируется и повторные слова отсекаются.

На рисунке 2.5 первый пример показываем работу стемминга с понижением регистра и удаление стоп-слов. Второй пример на этом рисунке демонстрирует, что для трех слов, которые написаны по-разному, но близких по смыслу, после стемминга получается одинаковое слово, поэтому в работе для вектора всех слов используются только уникальные результаты стемминга. На третьем примере видно, как слова похожие по написанию, но обозначающие совершенно разные вещи, имеют одинаковое слово в качестве результата стемминга. В этом и заключается главная особенность стемминга: его значительно проще внедрить и скорость работы у него выше, чем у лемминга, но за это приходится расплачиваться низкой «аккуратностью», которая, возможно, и не будет иметь особого значения в некоторых случаях.

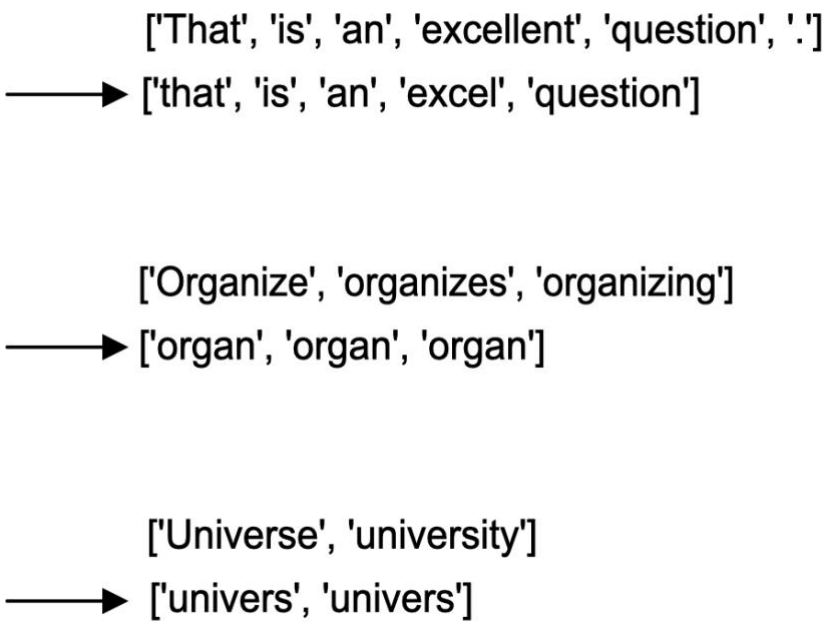

Рисунок 2.5 – Пример работы стемминга с понижением регистра и удаление стоп-слов

2.2.3 Мешок слов

После того, как был создан вектор всех слов, появляется возможность использовать функцию мешка слов. Реализация этой функции приведена в листинге 2.5

Листинг 2.5 – Реализация функции мешка слов

```
def bag_of_words(tokenized_sentence, all_words):
    """
    Вернет массив "bag_of_words":
        1 - ставится если слово есть в предложении;
        0 - ставится если слова нет в предложении.
    Пример:
    sentence (incoming) = ["hello", "how",   "are", "you"]
    all_words     = ["hi",    "hello", "I",   "you", "bye", "thank", "cool"]
# Собраны из всех patterns, которые хранятся в файле json
    bag           = [  0 ,       1 ,    0 ,     1 ,    0 ,     0 ,      0]
    """
    tokenized_sentence = [stem(w) for w in tokenized_sentence]

    bag = np.zeros(len(all_words), dtype=np.float32)
    for idx, w in enumerate(all_words):
        if w in tokenized_sentence:
            bag[idx] = 1.0

    return bag
```

На вход этой функции подаются два параметра: токенизированное предложение и вектор всех слов. К токенизированному предложению применяется функция стемминга с понижением регистра при необходимости.

Удалять повторные или стоп-слова из этого предложения нет необходимости. На основе вектора всех слов создается вектор «bag» такой же размерности, который заполнен нулями. Далее в цикле проверяется есть ли слово из предложения в векторе всех слов, если да, то в созданном векторе «bag» соответствующая ячейка заменяется на 1. Именно поэтому из обработанного вектора предложения не удаляются одинаковые и стоп-слова (если слово встречается несколько раз, то это ни на что не повлияет, так как вектора «bag» уже стоит 1, а для стоп-слов не предусмотрено даже ячеек с 0, поэтому наличие или отсутствие стоп-слов ни на что не повлияет). В результате этой функции будет получен так называемый «one-hot» вектор, который используется в модели нейронный сети в качестве входных данных.

2.3 Нейронная сеть

В проекте реализована нейронная сеть прямого распространения (feed forward) с двумя скрытыми слоями. Реализация модели находится в файле «model.py», а ее схематичный вид можно посмотреть на рисунке 2.2.

Количество входных нейронов равняется количеству элементов вектора всех слов (вектор «all_words»), получение этого вектора показано в листинге 2.4. Количество выходных нейронов равняется количеству «tag» из датасета. Сколько брать скрытых слоев, как и количество нейронов в этих слоях никто не знает, но чем глубже нейронная сеть, тем более сложные зависимости она способна определить. В работе для определения к какому классу относится присланное сообщения было выбрано два скрытых слоя. Количество нейронов в этих слоях равняется половине входных нейронов. Эти гиперпараметры настраиваются при создании модели, в файле «train.py», где будет происходить обучение модели. Помимо этих гиперпараметров есть еще и другие: «num_epochs», «batch_size» и «learning_rate». В ходе многократных тестов были определены оптимальные значения гиперпараметров модели, которые приведены в листинге 2.6.

Листинг 2.6 – Гиперпараметры модели

```
# Hyper-parameters
```

```
num_epochs = 2000                       # Кол-во тренировок для всех наборов
данных
batch_size = 30                         # Размер входных данный для одной
итерации
learning_rate = 0.00001                 # Скорость обучения
input_size = len(X_train[0])            # Кол-во нейронов на входном слое #
input_size = len(all_words) - так нагляднее
hidden_size = int(len(X_train[0])/2)    # Кол-во нейронов в скрытом слое
output_size = len(tags)                 # Кол-во нейронов в выходном слое
```

2.4 Обучение, сохранение результата в файл и использование модели

Обучение модели реализовано в файле «train.py» и проходит в два этапа: прямой ход и обратный ход. При прямом ходе тестовые данные прогоняются через нейронную сети и получается ответ, который сравнивается с заранее известным ответом. Дальше наступает обратный ход (по-другому он еще называется метод обратного распространения ошибки), в котором ошибка этого сравнения используется для изменения весов нейронов таким образом, чтобы ошибка постепенно уменьшалась. Величина, на которую изменяются веса, определяются параметром «learning_rate».

В каждом нейроне, после подсчета весов (веса считаются при прямом ходе), применяется функция активации. Функция активации — это нелинейная функция, которая вносит нелинейные изменения в выход (результат вычислений) слоя. Она гарантирует, что сеть способна вычислить всевозможные сложные функции, включая очень сильно нелинейные. Наиболее распространённые функции «сигмоид», «гиперболический тангенс» и «ReLU», графики которых показаны на рисунке 2.6. В проекте используется функция «ReLU».

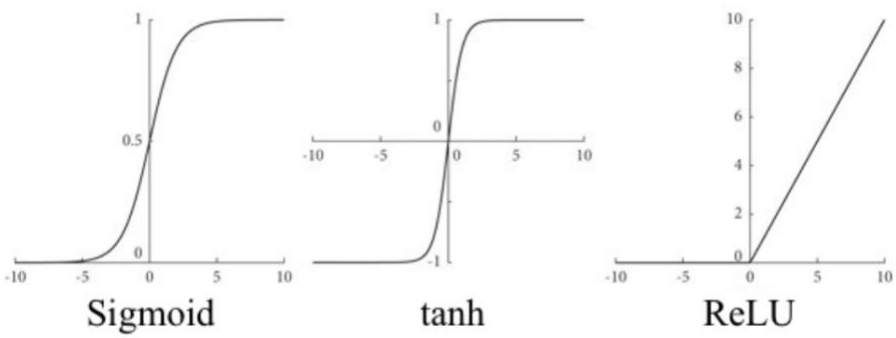

Рисунок 2.6 – Графики функции активации

Основные свойства функции «ReLU»:

- Выход равен входу, если входное значение больше 0. В ином случае выход равняется нулю;
- Принимает значения в диапазоне от 0 до бесконечности. Это значит, потенциально выходное значение может быть очень большим. Могут быть проблемы с взрывающимся градиентом;
- Преимущество ReLU в том, что она оставляет сеть более легкой, так как некоторые нейроны могут иметь выходное значение 0, предотвращая тем самым одновременную активность всех нейронов;
- Проблема ReLU в том, что её левая часть абсолютно плоская. Это означает, что значение её градиента — скорости изменения — будет нулевым, что негативно сказывается на эффективности вычислений;
- Вычислить значение ReLU очень просто, для компьютера это дешёвая операция;
- В настоящее время ReLU — наиболее используемая функция активации на внутренних слоях.

Основана цель обучения — это минимизировать функцию потерь. Функция потерь — это мера того, насколько хорошо модель прогнозирования предсказывает ожидаемый результат (или значение). Минимизация функции потерь происходит за счет обновления параметров нейронной сети – это обычно веса связей. Этим занимается оптимизатор, который представляет собой метод достижения лучших результатов, путем незначительных изменений параметров, таких как веса и скорость обучения, чтобы модель работала правильнее.

В качестве функции потерь («loss function») используется перекрестная энтропия («CrossEntropyLoss») из библиотеки «PyTorch». А в качестве оптимизатора используется адаптивная оценка момента («Adam») из той же библиотеки.

Результат обучения и гиперпараметры модели сохраняются в файл «data.pth», для того чтобы не проводить обучение каждый раз при запуске бота

и не задумывать при использовании модели в других места с какими гиперпараметрами она создавалась.

Результат запуска обучения модели с гиперпараметрами, которые указаны в листинге 2.6, показан на рисунке 2.7. Значение функции потерь с каждой эпохой постепенно снижается до 0, что логично и является правильным. То насколько хорошо обучилась модель, можно будет проверить после запуска бота, которому будут отправлены сообщения.

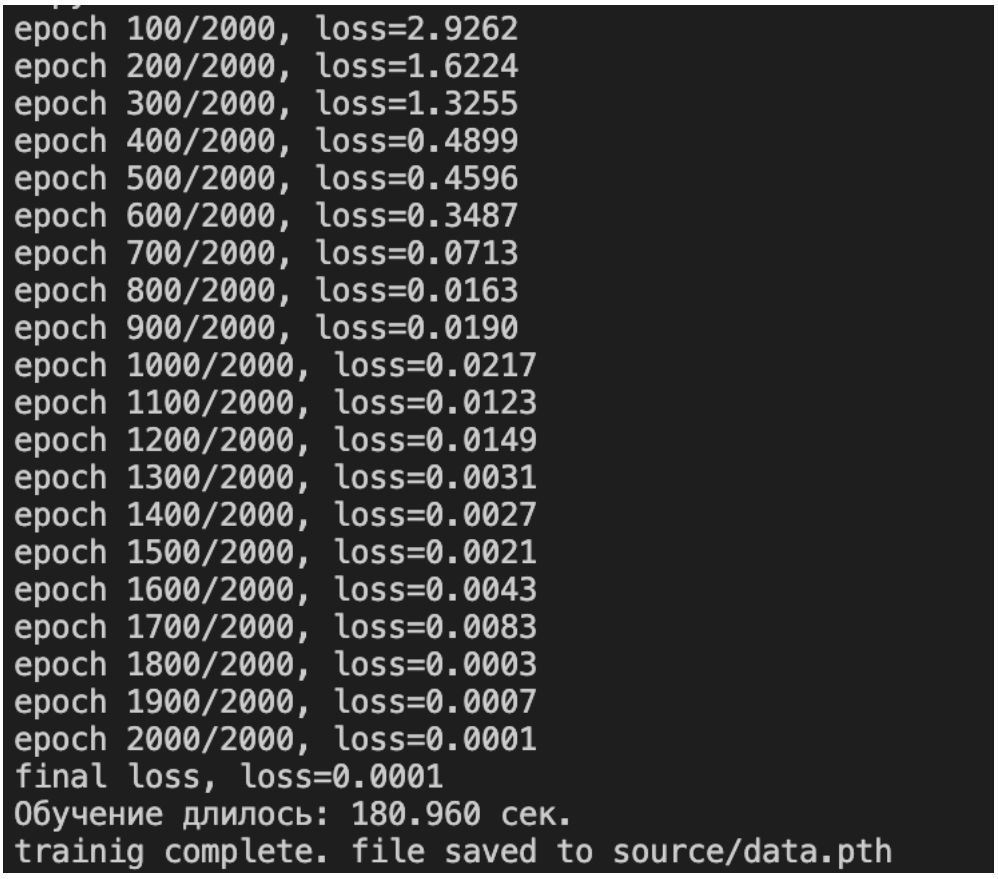

Рисунок 2.7 – Результат обучения модели

При использовании модели на реальных данных есть одна особенность, к выходному слою не применяется функция активации, вместо нее используется функция softmax, которая позволяет интерпретировать результат как вероятность получения класса. Функция softmax делит результат на сумму всех выходов, поэтому сумма всех выходных сигналов будет равняться 1. Softmax гарантирует, что значение суммы вероятностей, связанных с каждым классом, всегда будет равняться 1. На рисунке 2.8 демонстрируется полная работа нейронной сети.

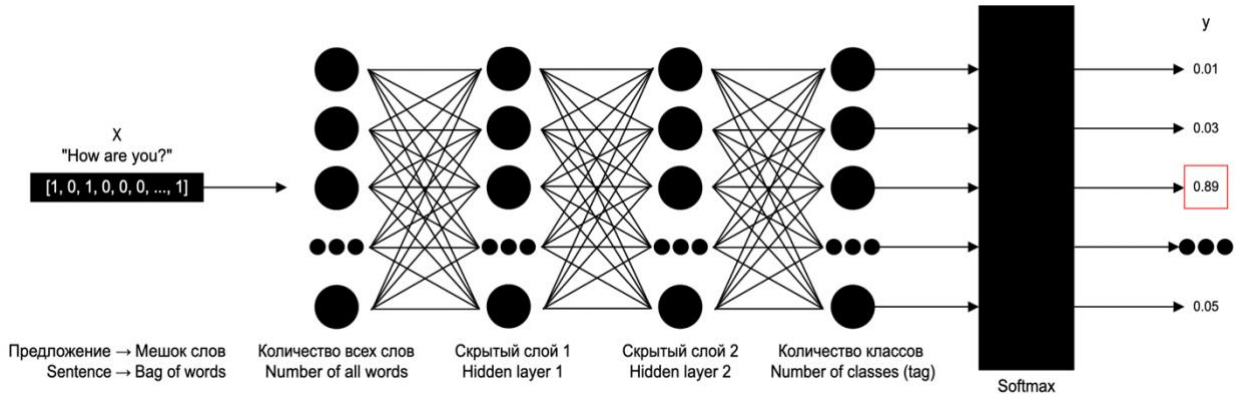

Рисунок 2.8 – Работа нейронной сети

2.5 Запуск бота

После обучения появляется возможность запустить бота, собственно ради чего все и затевалось. Есть две возможности запуска: запуск в автономном режиме и запуск в мессенджере Telegram. Запустив файл «console_chat_with_Jarvis.py», чат-бот будет работать в консоли, т.е. автономно, а если запустить «IntelligentChatBotTelegram.py», то чат-бот будет работать в Telegram.

Разумный ответ бота (ответ из датасета) будет получен только в том случае, если он уверен, что полученное сообщение относится к определенному тегу, больше чем на 75%. Если он не уверен в этом, то ответ будет таким: «I do not understand...».

Чтобы проверить как работает чат-бот, хорошо ли он обучен, нужно что-то написать ему. Результат проверки отображен на рисунке 2.9. Из рисунка видно, что чат-бот хорошо понимает тему полученного сообщения и отвечает соответствующим образом.

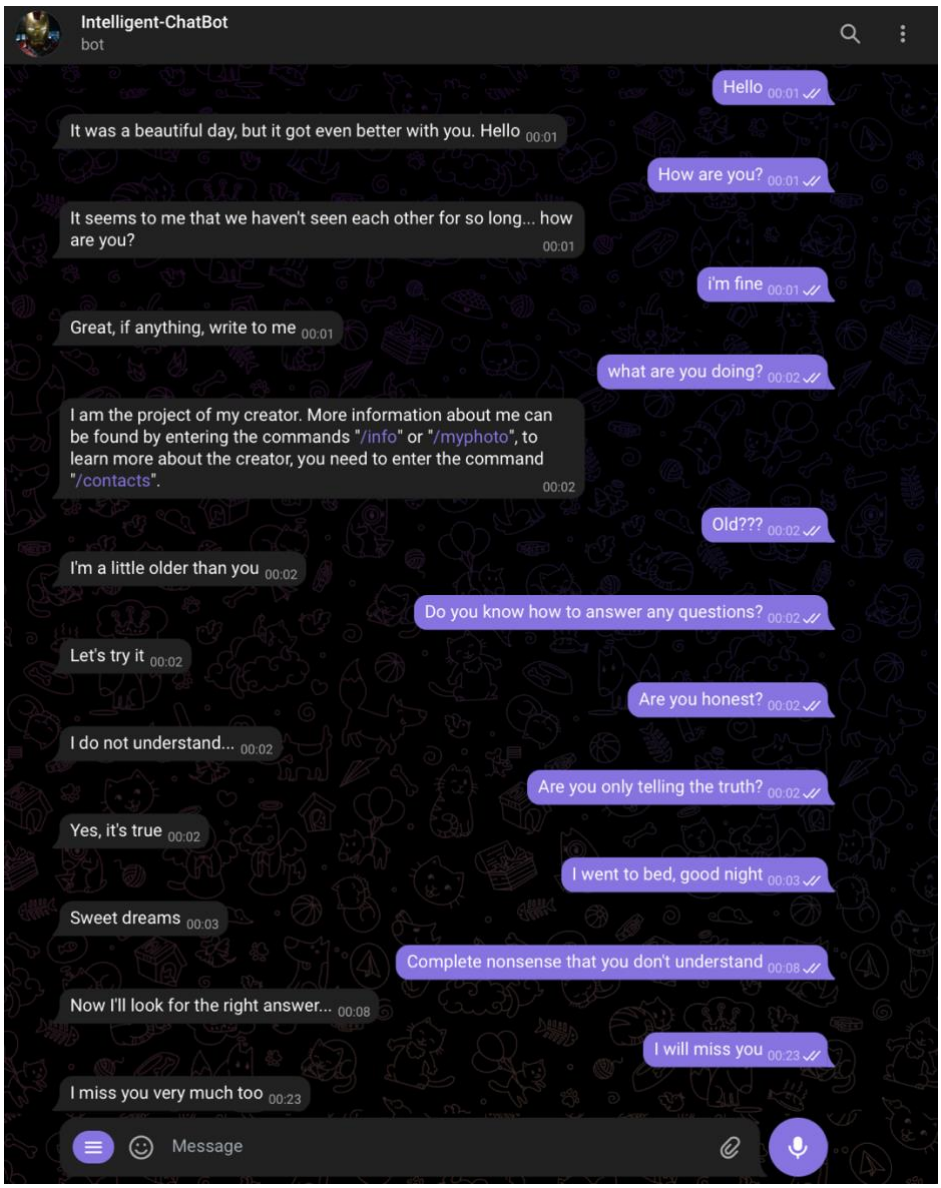

Рисунок 2.9 – Результат работы интеллектуального чат-бота в Telegram

Заключение

На сегодняшний день многие компании стали использовать чат-ботов, в связи с чем возрастает и потребность в программистах, умеющих их создавать. Качественно созданный чат-бот может как увеличить прибыль компании, так и сократить ее расходы. Все зависит от типа внедренного чат-бота. Но какой бы тип был выбран, у всех чат-ботов есть общая черта, все они направленны на повышение опыта взаимодействия с компанией. Чат-бот понимающий смысл человеческих слов, одна из актуальных реализаций чат-бота, который используется в компаниях. Такая реализация полезна тем, что позволяет разгрузить сотрудников службы поддержки или удержать пользователей в своем приложении, в случае если используется бот, поддерживающий легкую беседу.

Чат-бота из курсовой работы, еще можно улучшить. Помимо решения задачи понимания текста, к боту еще можно дополнительно реализовать понимание эмоционального окраса текста, что поможет в определении настроения пользователей. Также можно улучшить выбор ответов, чтобы они не выбирались из набора различных заготовленных ответов, а генерировались нейросетью. Две эти задачи позволять повысят качества бота, потому что теперь он будет каждый раз выдавать различные ответы так еще и будет генерировать их на основе текущего настроения пользователя.

Список литературы

1. International Journal of Innovative Research in Computer Science & Technology (IJIRCST) ISSN: 2347-5552, Volume-6, Issue-3, May 2018 DOI: 10.21276/ijircst.2018.6.3.2

2. NumPy documentation : сайт. – URL: https://numpy.org/doc/stable/ (дата обращения: 31.10.2022)

3. NLTK documentation : сайт. – URL: https://www.nltk.org (дата обращения: 31.10.2022)

4. PyTorch documentation : сайт. – URL: https://pytorch.org/docs/stable/index.html (дата обращения: 31.10.2022)

5. PyTelegramBotApi documentation : сайт. – URL: https://pytba.readthedocs.io/en/latest/ (дата обращения: 31.10.2022)

Приложение

IntelligentChatBotTelegram.py

```
import telebot
from telebot import types

from console_chat_with_Jarvis import Jarvis, bot_name
import settings # Импортируем TOKEN

bot = telebot.TeleBot(settings.TOKEN)

# Команда "/start" - выведет небольшое знакомство -> Работа с командами
@bot.message_handler(commands=['start'])
def start(message):
    welcome = f'Привет, <b>{message.from_user.first_name}</b>.\
            \nМеня зовут <b>{bot_name}</b>. Мой создатель <b>Святченко
Артём, студент РТУ МИРЭА</b>.\
            \n\nБольше информации обо мне можно узнать, введя
команду "/info" или воспользовавшись соответствующей функциональной
кнопкой.\
            \n\nA чтобы узнать больше о моем создателе, введите
команду "/contacts"или воспользуйтесь соответствующей функциональной
кнопкой.\
            \n\nЕсли вдруг функциональные кнопки не появились, то их нужно
создать командой "/buttons".'
    bot.send_message(message.chat.id, welcome, parse_mode='html') # 1
параметр - в какой чат отправляет ответ, 2 параметр - наш ответ, 3 параметр
- режим отправки ответов

# Команда "/buttons" - создаст функциональные кнопки. (При первом запуске
бота, обычно этих кнопок нет) -> Работа с командами + Создание
функциональных кнопок
@bot.message_handler(commands=['buttons'])
def create_function_buttons(message):
   markup = types.ReplyKeyboardMarkup(resize_keyboard=True, row_width=3) #
resize_keyboard = True - масштабирование кнопок под ПК и телефон,
row_width=1 - кол-во кнопок в ряду
   item_start = types.KeyboardButton('/start') # Параметр - текст на кнопке
   item_info = types.KeyboardButton('/info')
   item_contacts = types.KeyboardButton('/contacts')
   item_documentation = types.KeyboardButton('/documentation')
   item_photo = types.KeyboardButton('/myphoto')
```

```
    item_buttons = types.KeyboardButton('/buttons')

    markup.add(item_start, item_info, item_contacts, item_documentation,
item_photo, item_buttons)
    bot.send_message(message.chat.id, "Функциональные кнопки готовы к
работе.", reply_markup=markup)

# Команда "/info" - выведет больше информации о боте -> Работа с командами +
Создание кнопок, встроенных в сообщение
@bot.message_handler(commands=['info'])
def info_about_Jarvis (message):
    answer = f'Меня зовут <b>{bot_name}</b>. Мой создатель <b>Святченко
Артём</b> дал мне это имя будучи вдохновлен искусственным интеллектом,
созданным <b>Тони Старком, он же Железный человек, из киновселенной
Марвел</b>.\
        \n\nЧто я умею?\
        \n- Я умею общаться. Для этого всего-то нужно написать что-то на
английском и разговор завяжется сам собой. Это моя главная задача, для этого
я был создан.\
        \n- Еще я знаю ссылки на документации различных python библиотек.
Чтобы я вам их показал, нужно воспользоваться командой "/documentation" или
соответствующей функциональной кнопкой.\
        \n- Также я могу посмотреть на вашу фотографию и показать свою.
Чтобы посмотреть на меня, нужно воспользоваться командой "/myphoto" или
соответствующей функциональной кнопкой, а чтобы я посмотрел на ваше фото,
просто пришлите мне ее.\
        \n\nМой создатель относит меня к ботам типа small talk.\
        \nSmall talk — это непринужденный разговор на отвлеченные темы,
например разговор о погоде.\
        \nФактически я являюсь решением одной из задач NLP (Natural Language
Processing).\
        \n\nРаботаю я под управлением нейронной сети с прямой связью и двумя
скрытыми слоями.\
        \nКонвейер предварительной обработки (NLP preprocessing pipeline)
следующий: string(messeg) -> tokenize -> lower+stem -> exclude punctuation
characters(stop words) -> bag of word->getting a one-hot vector.\
        \n\nМою реализацию можно найти в GitHub репозитории.'

    markup = types.InlineKeyboardMarkup() # InlineKeyboardMarkup - класс,
который позволяет создавать различные встроенные в сообщения вещи (различные
кнопки, изображения и т.д.)
    markup.add(types.InlineKeyboardButton("GitHub repository",
```

```
url="https://github.com/rezabungel/Intelligent-ChatBot")) # 1 параметр - текст, который написан на кнопке, 2 параметр - URL-адрес

    bot.send_message(message.chat.id, answer, reply_markup=markup, parse_mode='html')

# Команда "/contacts" - выведет больше информации о создателе (Святченко Артём) -> Работа с командами + Создание кнопок, встроенных в сообщение
@bot.message_handler(commands=['contacts'])
def info_about_creator (message):
    answer = 'Святченко Артём, студент РТУ МИРЭА.\
            \nСсылка на мой GitHub профиль.'

    markup = types.InlineKeyboardMarkup() # InlineKeyboardMarkup - класс, который позволяет создавать различные встроенные в сообщения вещи (различные кнопки, изображения и т.д.)
    markup.add(types.InlineKeyboardButton("GitHub profile", url="https://github.com/rezabungel")) # 1 параметр - текст, который написан на кнопке, 2 параметр - URL-адрес

    bot.send_message(message.chat.id, answer, reply_markup=markup)

# Команда "/documentation" - выдаст названия и ссылки на документации python библиотек -> Работа с командами + Создание кнопок, встроенных в сообщение
@bot.message_handler(commands=['documentation'])
def documentation(message):
    python_libraries = [["Scikit-learn", "https://scikit-learn.org/stable/#"], ["NLTK (Natural Language Toolkit)", "https://www.nltk.org"],
["NumPy", "https://numpy.org/doc/stable/"],
["Pandas", "https://pandas.pydata.org/docs/"],
                        ["Pytorch", "https://pytorch.org/docs/stable/index.html"], ["Matplotlib", "https://matplotlib.org/stable/api/index.html"],
["Jupyter", "https://docs.jupyter.org/en/latest/"],
["pyTelegramBotAPI", "https://pytba.readthedocs.io/en/latest/index.html"]]

    for library in python_libraries:
        markup = types.InlineKeyboardMarkup() # InlineKeyboardMarkup - класс, который позволяет создавать различные встроенные в сообщения вещи (различные кнопки, изображения и т.д.)
        markup.add(types.InlineKeyboardButton(library[0], url=library[1])) # 1 параметр - текст, который написан на кнопке, 2 параметр - URL-адрес
```

```
        bot.send_message(message.chat.id, f"Документация {library[0]}",
reply_markup=markup)

# Команда "/myphoto" - выведет фотографию Jarvis и IronMan -> Работа с
командами + Вывод фотографий
@bot.message_handler(commands=['myphoto'])
def show_photo(message):
    #Отправка фотографии 1
    bot.send_message(message.chat.id, "Так я вижу себя:", parse_mode='html')
    photo = open('source/Jarvis.png', 'rb') # Открытие фотографии
стандартными методами Питона. # 'rb' - тип открытия фотографии
    bot.send_photo(message.chat.id, photo)
    #Отправка фотографии 2
    bot.send_message(message.chat.id, "Может быть вы ходите посмотреть еще
на Железного человека:", parse_mode='html')
    photo = open('source/IronMan.jpg', 'rb')
    bot.send_photo(message.chat.id, photo)

# Работа с текстом (отслеживание любых текстовых сообщений)
@bot.message_handler(content_types=['text'])
def get_user_text(message):
    Jarvis_answer = Jarvis(message.text)
    bot.send_message(message.chat.id, Jarvis_answer, parse_mode='html')

    # Вывод сообщений пользователей и ответов бота на них в консоль
    # Вывод имени, логина, id и сообщения пользователя в консоль. (Имя,
логин, id - позволяют определить конкретного пользователя, который вводил
сообщение)
    print(f'Name:{message.from_user.first_name}
Username:{message.from_user.username} id:{message.from_user.id}
message_from_user:   {message.text}')
    # Вывод ответа бота в консоль
    print(f"Jarvis' answer:   {Jarvis_answer}\n")

# Работа с документами (боту отправляют фото)
@bot.message_handler(content_types=['photo'])
def photo_detected(message):
    answer = 'Обнаружил, что вы отправили мне фотографию, но я умею только
разговаривать на английском языке.\
        \n\nКстати, если хотите посмотреть на меня, введите
команду "/myphoto" или воспользуйтесь соответствующей функциональной
кнопкой.'
```

```
    bot.send_message(message.chat.id, answer, parse_mode='html')


# Запуск бота на постоянное выполнение.
bot.polling(non_stop=True)
```

console_chat_with_Jarvis.py

```
import random
import json

import torch

from model import NeuralNet
from NLTK_utils import tokenize, bag_of_words

def Jarvis(messeg): # Функция, выдающая ответ на введенное пользователем сообщение
    messeg = tokenize(messeg)
    X = bag_of_words(messeg, all_words)
    X = X.reshape(1, X.shape[0])
    X = torch.from_numpy(X).to(device)

    output = model(X) # Предсказываем значение ответов (тегов)
    _, predicted = torch.max(output, dim=1) # Ищем наиболее подходящий ответ (тег - с наибольшим значением)
    tag = tags[predicted.item()] # Получаем фактический тег ответа

    probs = torch.softmax(output, dim=1) # Применяем softmax для получения значения вероятности (уверенности) выборов тегов. (Дальше сделаем проверку на основе вероятности лучшего тега)
    prob = probs[0][predicted.item()] # Берем значение вероятности лучшего тега

    if prob.item() > 0.75: # Если вероятность (уверенность) выбора тега достаточно велика, то мы ищем ответ в JSON файле, иначе говорим, что не понимаем, что ввел пользователь
        for intent in intents["intents"]:
            if tag == intent["tag"]:
                answer = random.choice(intent['responses']) # Случайно выбираем один из возможных ответов
    else:
        answer = "I do not understand..."

    return answer



device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # Если
```

```
есть возможность, то обработка будет производиться на GPU, если нет, то на
CPU

with open('source/intents.json', 'r') as json_data:  # Безопасное открытие
файла (+ гарантия его закрытия). Режим открытия: r - read mood (открыть файл
в режиме чтения)
    intents = json.load(json_data) # Возвращает JSON объект как словарь
(Чтения содержимого JSON файла)

FILE = "source/data.pth"
data = torch.load(FILE)

model_state = data["model_state"]
input_size = data["input_size"]
output_size = data["output_size"]
hidden_size = data["hidden_size"]
all_words = data["all_words"]
tags = data["tags"]

model = NeuralNet(input_size, hidden_size, output_size).to(device)
model.load_state_dict(model_state) # Теперь модель знает наши изученные
параметры
model.eval() # Устанавливаем модель в режим оценки

# Реализация чата в консоли

bot_name = "Jarvis"

# Общение в консоли. Работает, когда запускаем файл с этим кодом. (Если
импортируем этот файл, то этот блок кода не выполнится)
if __name__ == "__main__":
    print("Let's chat! (type 'quit' to exit)")

    while True:
        sentence = input("You:  ")
        if sentence == "quit":
            break

        answer = Jarvis(sentence)
        print(f"{bot_name}:  {answer}")
```

train.py

```
import numpy as np
import json

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

from NLTK_utils import tokenize, stem, bag_of_words
from model import NeuralNet

import time # Используем для вычисления времени, которое уходит на обучение модели

with open('source/intents.json', 'r') as f:  # Безопасное открытие файла (+гарантия его закрытия). Режим открытия: r - read mood (открыть файл в режиме чтения)
    intents = json.load(f) # Возвращает JSON объект как словарь (Чтения содержимого JSON файла)

all_words = []
tags = []
xy = []

# Цикл через каждое предложение в нашем intents patterns (Токенизируем все слова из patterns и связываем их с тегом в кортеж)
for intent in intents['intents']:
    tag = intent['tag']
    tags.append(tag) # Добавляем тег в массив
    for pattern in intent['patterns']:
        w = tokenize(pattern) # Токенизируем каждое слово в предложении
        all_words.extend(w) # Добавляем токены в наш массив слов
        xy.append((w, tag)) # Связываем соответсвующий тег и токены в кортеж и добавляем его в массив

# Удаление стоп-слов (знаки пунктуаций) и применяем Стеммера (+преобразование из верхнего регистра в нижний)
stop_words = ['?', '!', '.', ',']
all_words = [stem(w) for w in all_words if w not in stop_words]

all_words = sorted(set(all_words)) # Сортировка и удаление повторных слов
tags = sorted(set(tags)) # Сортировка и удаление повторных тегов (ничего не
```

```
удалиться, так как все теги уникальны)

# Bag of words
# Creating Training Data
X_train = []
y_train = []

for (pattern_sentence, tag) in xy:
    bag = bag_of_words(pattern_sentence, all_words)
    X_train.append(bag) # X: bag of words for each pattern_sentence
    label = tags.index(tag)
    y_train.append(label) # y: PyTorch CrossEntropyLoss needs only class
labels, not one-hot

X_train = np.array(X_train)
y_train = np.array(y_train)

class ChatDataset(Dataset):
    def __init__(self):
        self.n_samples = len(X_train)
        self.x_data = X_train
        self.y_data = y_train

    # Поддерживание индексации таким образом, чтобы dataset[i] можно было
использовать для получения i-й выборки
    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index] # Возвращает в виде
кортежа

    # Можем вызвать len(dataset), чтобы получить размер
    def __len__(self):
        return self.n_samples

# Hyper-parameters
num_epochs = 2000                       # Кол-во тренировок для всех наборов
данных
batch_size = 30                         # Размер входных данный для одной
итерации
learning_rate = 0.00001                 # Скорость обучения
input_size = len(X_train[0])            # Кол-во нейронов на входном слое #
input_size = len(all_words) - так нагляднее
hidden_size = int(len(X_train[0])/2)    # Кол-во нейронов в скрытом слое
```

```
output_size = len(tags)                # Кол-во нейронов в выходном слое

#print(input_size, len(all_words))
#print(output_size, tags)
#print(input_size, output_size)

dataset = ChatDataset()
train_loader = DataLoader(dataset=dataset, batch_size=batch_size,
shuffle=True, num_workers=0) # shuffle - перемешивание, num_workers -
мультипоток (в win может быть ошибка, если стоит не 0)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # Если
есть возможность, то обработка будет производиться на GPU, если нет, то на
CPU
model = NeuralNet(input_size, hidden_size, output_size).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss() # Функция потерь - мера того, насколько
хорошо модель прогнозирования предсказывает ожидаемый результат
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) #
Оптимизаторы используются для обновления весов и скорости обучения, то есть
внутренних параметров модели, чтобы уменьшить ошибку, а следовательно,
увеличить точность работы модели

start_time = time.time() # Запуск секундомера
# Train the model
for epoch in range(num_epochs):
    for (words, labels) in train_loader:
        words = words.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(words)                    # Передний пропуск:
определение выходного класса
        loss = criterion(outputs, labels)         # Определение потерь:
разница между выходным классом и предварительно заданной меткой

        # Backward and optimize
        optimizer.zero_grad()                     # Инициализация скрытых масс
до нулей
        loss.backward()                           # Обратный проход:
определение параметра weight
```

```
        optimizer.step()                        # Оптимизатор: обновление
параметров веса в скрытых узлах

    if ((epoch+1) % 100 == 0):                  # Вывод информации о каждых
100 эпохах
        print(f"epoch {epoch+1}/{num_epochs}, loss={loss.item():.4f}")



end_time = time.time() - start_time # Остановка секундомера
print(f"final loss, loss={loss.item():.4f}")
print(f'Обучение длилось: {"%.3f" % end_time} сек.')

# Сохраним результат нашей обученной модели в файл
data = {
    "model_state": model.state_dict(),
    "input_size": input_size,
    "output_size": output_size,
    "hidden_size": hidden_size,
    "all_words": all_words,
    "tags": tags
}

FILE = "source/data.pth"
torch.save(data, FILE)

print(f"trainig complete. file saved to {FILE}")
```

NLTK_utils.py

```
import nltk
import numpy as np

from nltk.stem.porter import PorterStemmer # Есть различные Стеммеры, мы воспользуемся этим

#nltk.download('punkt') # Используется при первом запуске для работы word_tokenize

# tokenize
def tokenize(sentence):
    """
    Разделение предложения на отдельные слова/токены
    Токенами могут быть: слова, пунктуация, цифры
    """
    return nltk.word_tokenize(sentence)

# stem
stemmer = PorterStemmer()
def stem(word):
    """
    stemming = ищет корневую форму слова
    Пример:
    words = ["organize", "organizes", "organizing"]
    words = [stem(w) for w in words]
    -> ["organ", "organ", "organ"]
    """
    return stemmer.stem(word.lower()) # К слову будет применен Стеммер и слово будет преобразовано в нижний регистр при необходимости

# bag of words
def bag_of_words(tokenized_sentence, all_words):
    """
    Вернет массив "bag_of_words":
        1 - ставится если слово есть в предложении;
        0 - ставится если слова нет в предложении.
    Пример:
    sentence (incoming) = ["hello", "how",   "are", "you"]
    all_words           = ["hi",    "hello", "I",   "you", "bye", "thank", "cool"] # Собраны из всех patterns, которые хранятся в файле json
    bag                 = [  0 ,       1 ,    0 ,     1 ,    0 ,     0
```

```
,       0]
    """
    tokenized_sentence = [stem(w) for w in tokenized_sentence]

    bag = np.zeros(len(all_words), dtype=np.float32)
    for idx, w in enumerate(all_words):
        if w in tokenized_sentence:
            bag[idx] = 1.0

    return bag
```

model.py

```
import torch
import torch.nn as nn

# Neural network with direct connection and two hidden layers
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        # No activation and no softmax at the end
        return out
```