КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня магістра
за спеціальністю 122 Комп'ютерні науки

на тему:

ПОБУДОВА АВТОМАТИЧНОЇ СИСТЕМИ

ВІДПОВІДЕЙ НА ПИТАННЯ

Виконав студент 2 курсу магістратури
Олексій БАШУК

(підпис)

Науковий керівник:
асистент, кандидат технічних наук
Олексій ФЕДОРУС

(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«15» травня 2023 р.,
протокол № 10
Завідувач кафедри
В. М. Терещенко

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг дипломної роботи складає 58 сторінок, 10 ілюстрацій, 5 джерел.

Перелік ключових слів: BERT, ВІДПОВІДЬ, НЕЙРОННІ МЕРЕЖІ, ПИТАННЯ, ПОДІБНІСТЬ, СЛОВНИК, ЧАТ-БОТ.

Об’єктом дослідження є проектування та розробка чат-боту для відповідей на питання, що вже були задані раніше.

Метою роботи є реалізація чат-боту здатного виявляти питання та, враховуючи специфіку чату, генерувати повідомлення-відповідь по виявлених відповідях на попередні подібні запитання.

З використаних методів досліджень та розробки можна виокремити: аналіз інтернет джерел, консультації в експертів в області штучного інтелекту.

В процесі роботи була розроблена теоретична модель, котра передбачає чотири основні етапи: побудова словнику, виявлення питання, пошук подібних питань, знаходження відповіді на минулі питання, генерація відповіді. На основі цієї теоретичної моделі також було успішно реалізовано чат-бота мовою Python для платформи Telegram. При цьому, основна логіка була максимально відокремлена, завдяки чому чат-бота можна легко інтегрувати на інші платформи.

Розробка такого чат-боту є актуальною через свою новизну, відсутність аналогів та пряму користь, як для звичайних користувачів, так і для техпідтримки чи модераторів чатів, а завдяки своїй інтегровності може бути легко впровадженим на інших, в тому числі корпоративних, платформах за потреби.

Рекомендації щодо використання результатів роботи: з огляду на структуру та побудову бота, він буде ефективним в чатах з великою кількістю користувачів, де часто задають схожі та однотипні питання. Максимальної ефективності він набуває в чатах, де є доступ до історії повідомлень, та де яскраво виділена тематика чату.

Сфера застосування: допомога та спрощення роботи техпідтримки у відкритих чатах.

Значимість даної роботи проявляється в її новизні та корисності, що в критичних ситуаціях може навіть врятувати життя.

Серед можливих пропозицій подальшого розвитку чат-бота можна виділити: використання баз даних для стабільнішої роботи; використання більшої кількості сигналів для встановлення відповіді на питання; використання фонових нейронних мереж для покращення якості побудови відповідей на питання з плином часу; використання технологій GPT для генерації повідомлення-відповіді на базі знайдених відповідей.

ЗМІСТ

ВСТУП

У нашому житті вже давно велику роль займають нейронні мережі [1]. Ми можемо зустріти їх всюди: коли робимо фотографію на телефон, коли намагаємось перекласти текст з однієї мови на іншу, коли залишаємо машину на автопілоті. Часто ми навіть не помічаємо, як користуємось ними. І хоч остання хвиля їх активного розвитку розпочалась відносно нещодавно, через обмеженість обчислювальних потужностей процесорів у минулому, вже було спроектовано та досліджено чимало їх різновидів націлених на виконання різного роду завдань.

Чимала частина цих досліджень присвячена саме обробці текстів. Це і не дивно: в електронному просторі людина обмінюється інформацією переважно текстово. І щоб всіляко спростити та покращити цей обмін, а також мати можливість спілкування безпосередньо з машиною, за допомогою нейронних мереж було розв’язано цілий ряд так званих задач обробки природної мови (NLP). До них входять: видобування даних, синтез мовлення, розпізнавання мови, генерування природної мови, машинний переклад, спрощення тексту, отримання зв’язків, семантичний аналіз та багато інших. Але машина, на відміну від людини, не може просто так, побачивши текст, зрозуміти значення кожного окремого слова та зміст речення загалом. Тому в більшості NLP задач все починається з питання представлення речення у вигляді векторів чисел, які є чимось більш знайомим для машин.

Однією з новітніх та інноваційних моделей, присвячених саме представленню речення у вигляді векторів чисел, стала вже попередньо навчена модель під назвою BERT [2][3], запропонована компанією Google. Її результати приголомшили суспільство та значно спростили процес побудови штучного інтелекту.

Та з розвитком досліджень у сфері штучного інтелекту, росте і кількість задач, які вже розв‘язані та які ще потребують розв‘язання. Так, однією доволі поширеною проблемою стала проблема повторних питань. У більшості людей в

житті бували ситуації, коли, через кращу обізнаність в якійсь сфері, до них приходили з одним і тим самим питанням. Ще частіше з проблемою повторюваних питань зустрічаються люди, що є відповідальним за надання відповідей користувачам на форумах чи чатах техпідтримки. Такі повтори виснажують людину морально та викликають роздратованість, бо знайти стару відповідь буває проблематично, а писати кожен раз заново набридає.

Ця проблема має і складніші виявлення та навіть може ставити життя під загрозу. Так, на початку повномасштабного вторгнення в Україну, особливо в перші дні, всюди панував хаос. В аптеках майже одразу позникали всі критичні ліки, нові завозились в дуже обмежених кількостях, а навіть звичайне пересування по місту було сильно ускладнене. Для людей з вадами серця чи діабетом, для яких було життєво необхідним щоденне вживання певних ліків, це було критичною проблемою. Та навіть звичайну питну воду в деяких районах міста було проблематично дістати, а через погану якість Інтернету було складно просто знайти працюючі джерела чи точки видачі. Щоб вийти з такого становища, набув популярності телеграм канал “Обмін ліками Київ”, в якому збиралась будь-яка корисна інформація, люди допомагали один одному, просили необхідні ліки та готові були віддати ті, яких було вдосталь. Але навіть так, через ту саму проблему зі зв’язком, не у всіх була можливість знаходити потрібне для себе в чаті. Щоб якось допомогти, я намагався зв’язувати людей за їх запитом з пропозицією, проте і мій ресурс був сильно обмеженим.

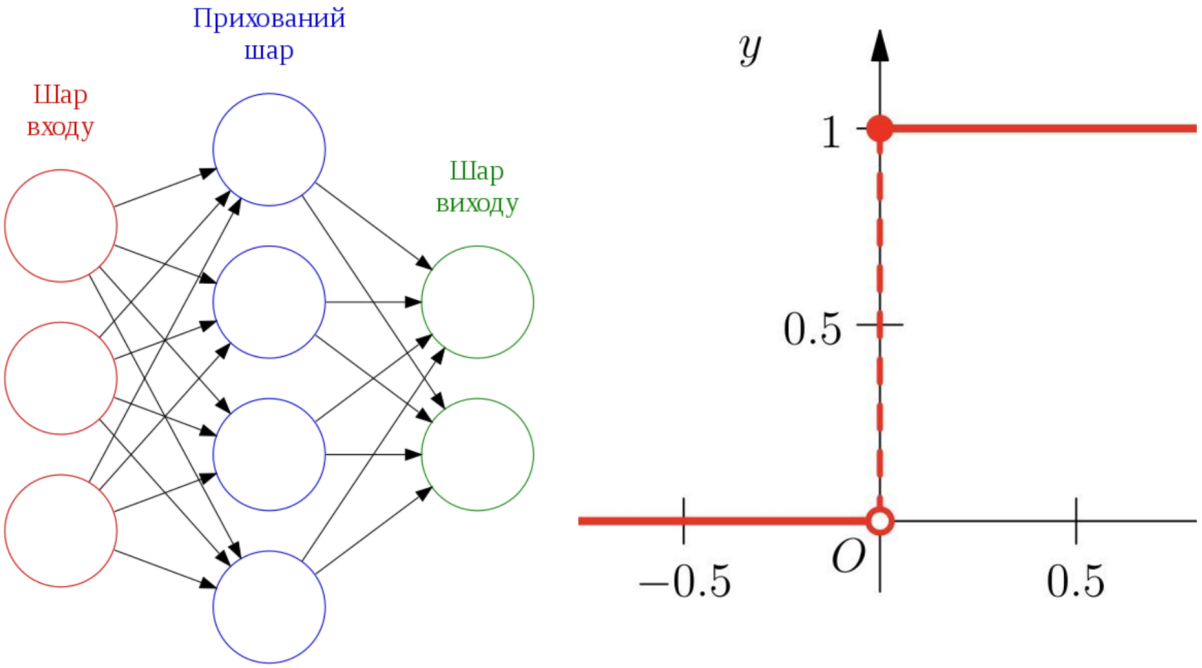
Ефективним розв‘язком таких проблем може стати чат-бот [4] з штучним інтелектом, що здатний слідкувати за перепискою чату та автоматично відповідати на питання, базуючись на попередніх повідомленнях. Саме побудові його теоретичної моделі, з активним використанням моделі BERT, та подальшій реалізації моделі і присвячена ця робота.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Нейронні мережі (NN)

Нейронні мережі, також відомі як штучні нейронні мережі (ANN) або змодельовані нейронні мережі (SNN), є підмножиною машинного навчання і лежать в основі алгоритмів глибинного навчання. Їх назва та структура натхненні людським мозком, імітуючи спосіб, яким біологічні нейрони сигналізують один одному.

Штучні нейронні мережі (ШНМ) складаються з шарів нейронів (рисунок 1.а), таких як: вхідний шар, один або кілька прихованих шарів і вихідний шар. Кожен штучний нейрон з'єднується з іншим і має певну вагу. Також невід'ємною частиною є функція активації (рисунок 1.б), в якості якої часто може виступати функція Гевісайда. Якщо вихід будь-якого окремого нейрона перевищує порогове значення 0, цей нейрон активується, надсилаючи дані на наступний рівень мережі. В іншому випадку дані не передаються на наступний рівень мережі.



а) б)

Рисунок 1 – Складові нейронних мереж: а – шари нейронів; б – функція активації

Нейронні мережі покладаються на навчальні дані, щоб із часом вчитись та покращувати свою точність. Проте, як тільки нейронні мережі досягають бажаного рівня точності, вони стануть потужними інструментами в інформатиці та штучному інтелекті, що дозволяє нам класифікувати та кластеризувати дані з високою швидкістю, а також наближено розв‘язувати задачі до цього непідвладні точним алгоритмам. Завдання розпізнавання мовлення або зображення можуть займати хвилини проти годин, витрачених експертами на ручну ідентифікацію. Однією з найвідоміших нейронних мереж є пошуковий алгоритм Google.

1.1.1 Принцип роботи

Про кожен окремий вузол мережі можна думати, як про окрему модель лінійної регресії, що складається з вхідних даних, вагових коефіцієнтів, зміщення та вихідних даних. Формула буде виглядати приблизно так:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m + bias$$

$$output = f(x) = 1, \text{якщо } \sum_{i=1}^{m} w_i x_i + bias \geq 0, \text{інакше } -0$$

Після визначення вхідного шару задаються ваги. Ці коефіцієнти допомагають визначити важливість будь-якої даної змінної, причому більші з них роблять більш значний внесок у результат порівняно з іншими вхідними значеннями. Потім усі вхідні дані множаться на відповідні ваги, а потім підсумовуються. Після цього сума передається в функцію активації, яка визначає вихід. Якщо цей вихід перевищує заданий поріг, він «запускає» (або активує) нейрон, передаючи дані наступному шару в мережі. Це призводить до того, що вихід одного шару стає на вхід наступного шару. Цей процес передачі даних від одного шару до наступного визначає цю нейронну мережу як мережу прямого зв’язку.

Розберімося, як може виглядати один шар, використовуючи двійкові значення. Ми можемо застосувати цю концепцію до більш наявного прикладу. Наприклад, чи варто займатися серфінгом (Так: 1, Ні: 0)? Рішення йти чи не йти — це наш прогнозований результат, або y-hat. Припустимо, що на наше рішення впливають три фактори:

- Хвилі хороші? (Так: 1, Ні: 0)
- Чи порожній берег? (Так: 1, Ні: 0)
- Чи був нещодавно напад акул? (Так: 0, Ні: 1)

Тоді припустимо наступне, надаючи нам такі вхідні дані:

- $x_1 = 1$, оскільки хвилі високі
- $x_2 = 0$, оскільки натовп вийшов
- $x_3 = 1$, оскільки нещодавнього нападу акули не було

Тепер нам потрібно призначити деякі ваги, щоб визначити важливість. Більші вагові коефіцієнти означають, що певні змінні мають більшу важливість для рішення чи результату.

- $w_1 = 5$, оскільки великі хвилі піднімаються рідко
- $w_2 = 2$, оскільки ми звикли до натовпу
- $w_3 = 4$, оскільки ми боїмось акул

Наприкінці, ми також візьмемо за порогове значення число 3, а тому значення зміщення рівне –3. Зібравши усі ці числа ми можемо підрахувати значення формули, щоб отримати бажаний результат:

$$\widehat{y} = f(x) = (1 * 5) + (0 * 2) + (1 * 4) - 3 = 6$$

Використавши функцію активації, доходимо висновку, що вихідний результат цього вузла буде 1, оскільки 6 більше 0. У цьому випадку ми будемо займатися серфінгом; проте змінюючи числа, ми можемо досягати різних результатів від моделі. Спостерігаючи за прийняттям одного рішенням, як у

наведеному вище прикладі, ми можемо приймати дедалі складніші рішення (наприклад чи йти на пляж) залежно від результатів попередніх рішень або шарів. Так будується нейронна мережа.

У наведеному вище прикладі ми використали персептрони (значення 0 або 1), щоб проілюструвати математичні формули, які було наведено, але нейронні мережі використовують сигмоїдні нейрони, які набувають значень від 0 до 1. Оскільки нейронні мережі поводяться так само, як дерева прийняття рішень, передаючи дані з одного нейрона до іншого, проміжок можливих значень від 0 до 1 зменшить вплив будь-якої зміни однієї змінної на значення виходу будь-якого подальшого нейрона, а згодом і на вихід нейронної мережі.

Коли ми почнемо думати про більш практичні варіанти використання нейронних мереж, як-от розпізнавання або класифікація зображень, ми використовуватимемо такі популярні методи, як кероване навчання або навчання по відмічених наборах даних, для навчання мережі. Під час навчання моделі нам захочеться оцінити її точність за допомогою функції витрат (або втрат). Її також зазвичай називають середньоквадратичним відхиленням (MSE). У рівнянні нижче,

- i - індекс вибірки
- $\widehat{y}$ - обрахований результат
- y - фактичне значення
- m - кількість зразків.

$$Cost\ Function = MSE = \frac{1}{2m}\sum_{i=1}^{m}(\widehat{y} - y)^2$$

При побудові та навчанні мережі, наша мета полягає в тому, щоб мінімізувати нашу функцію витрат, щоб забезпечити правильність відповідності для будь-якого спостереження. Для коригування своїх вагів та зміщень, модель використовує так звану функцію вартості та метод навчання з підкріпленням, щоб досягти точки наближення або, так званого, локального мінімуму. Процес, у якому алгоритм коригує свої ваги, здійснюється шляхом градієнтного спуску, що

дозволяє моделі визначати напрямок, в бік якого слід рухатися, щоб зменшити похибку (або мінімізувати функцію вартості). З кожним навчальним прикладом параметри моделі коригуються, щоб поступово зближатися до мінімуму.

Більшість глибинних нейронних мереж мають лише прямий зв'язок, тобто вони протікають тільки в одному напрямку, від входу до виходу. Однак навіть таку модель можна навчати шляхом зворотного поширення; тобто рухатися в протилежному напрямку від виходу до входу. Зворотне розповсюдження дозволяє розраховувати й приписувати помилки кожному нейрону, що дозволяє нам належним чином налаштувати та підігнати параметри моделі.

1.2 BERT

BERT або Bidirectional Encoder Representations from Transformers - це інноваційна модель, побудована для обробки природної мови, розроблена компанією Google у 2018 році та використана у своїх продуктах у 2019. Як можна здогадатись з назви, BERT базується на трохи старішій моделі Transformer [5], де основною ідеєю є розуміння самої мови через розуміння зв'язків слів між собою та загального контексту. Проте, на відміну від старіших моделей, таких як Bi-LSTM, де розуміння контексту будується на так званій довгій короткостроковій пам'яті, BERT дивиться на речення цілком і позбувається проблем втрати старих знань та розділу пам'яті на дві незалежні ланки, що спрямовані в протилежні боки.

Коли BERT було опубліковано, вона досягла найвищого рівня продуктивності в низці задач розуміння природної мови, а саме General Language Understanding Evaluation (GLUE), Stanford Question Answering Dataset (SQuAD) і Situations With Adversarial Generations (SWAG). Результати виявились настільки приголомшливими, що постало питання заміни повної LSTM на BERT.

Але перш, ніж ми розберемо структурні особливості BERT, спершу розглянемо структуру Трансформеру.

1.2.1 Трансформер

Одне з базових застосувань трансформеру - машинний переклад. Модуль приймає на вхід речення та повертає рядок із його перекладом. Якщо дивитись всередину моделі (рисунок 2), то можна виокремити два основні блоки: блок кодування, що переводить речення в вектори чисел, і блок декодування, що бере вихід з кодування і початкове речення, а повертає його переклад. Ми зосередимо увагу саме на першому блоці.

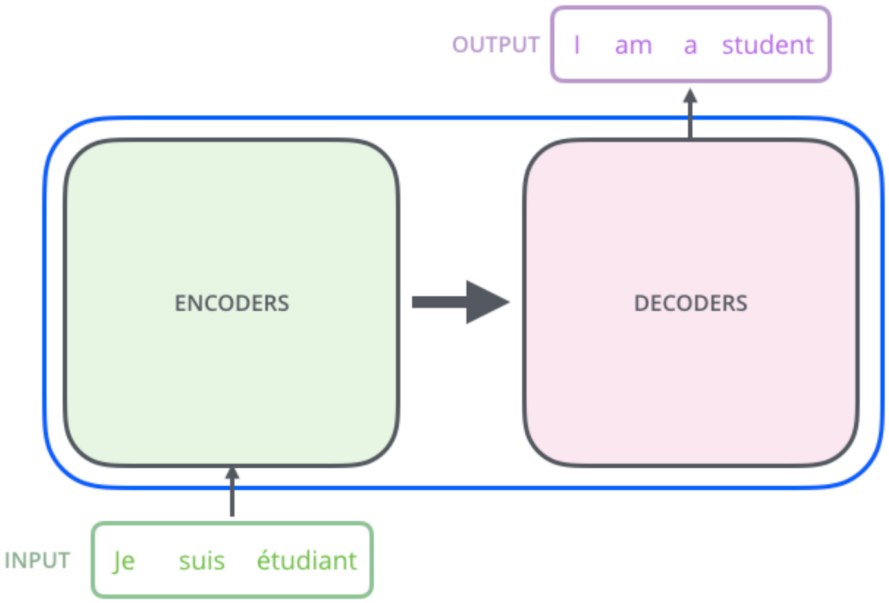


Рисунок 2 – Структура трансформеру

Блок кодування - це стек енкодерів, які почергово передають свій вихід на вхід наступному енкодеру. Кожен енкодер, в свою чергу, можна розділити на два підшари: шар багатоголової внутрішньої уваги та шар прямого розповсюдження. Перший шар складніший і, при розгляді кожного окремого слова, допомагає енкодеру звернути увагу на інші слова. Другий же шар є більш допоміжним і просто покращує роботу системи, роблячи певну нормалізацію та повнозв‘язне з‘єднання.

Як і у разі будь-якої NLP-задачі, ми починаємо з того, що перетворюємо слово на вектор чисел, використовуючи алгоритм ембеддингу слів. Кожне слово перетворюється на вектор розмірністю 512. Ембеддинги застосовуються тільки в

нижньому енкодері. Далі всі енкодеру приймають і повертають набори векторів розмірності 512. Розмір цього набору векторів є гіперпараметром, який ми можемо встановлювати, і, по суті, рівний довжині самого довгого речення в навчальному корпусі.

Тепер розглянемо, що відбувається у шарі внутрішньої уваги при розгляді кожного окремого вектора слова. Спершу йде обчислення трьох векторів по кожному вхідному вектору: вектор запиту (Query vector), вектор ключа (Key vector) та вектор значення (Value vector). Ці вектори створюються за допомогою перемноження ембедингу на три матриці, які навчаються в процесі навчання.

Множення x1 на матрицю ваг WQ виробляє q1, вектор "запиту", що відноситься до цього слова. Аналогічно ми створюємо проекції «запиту», «ключа» та «значення» для кожного слова у вхідному реченні, як показано на рис. 3:

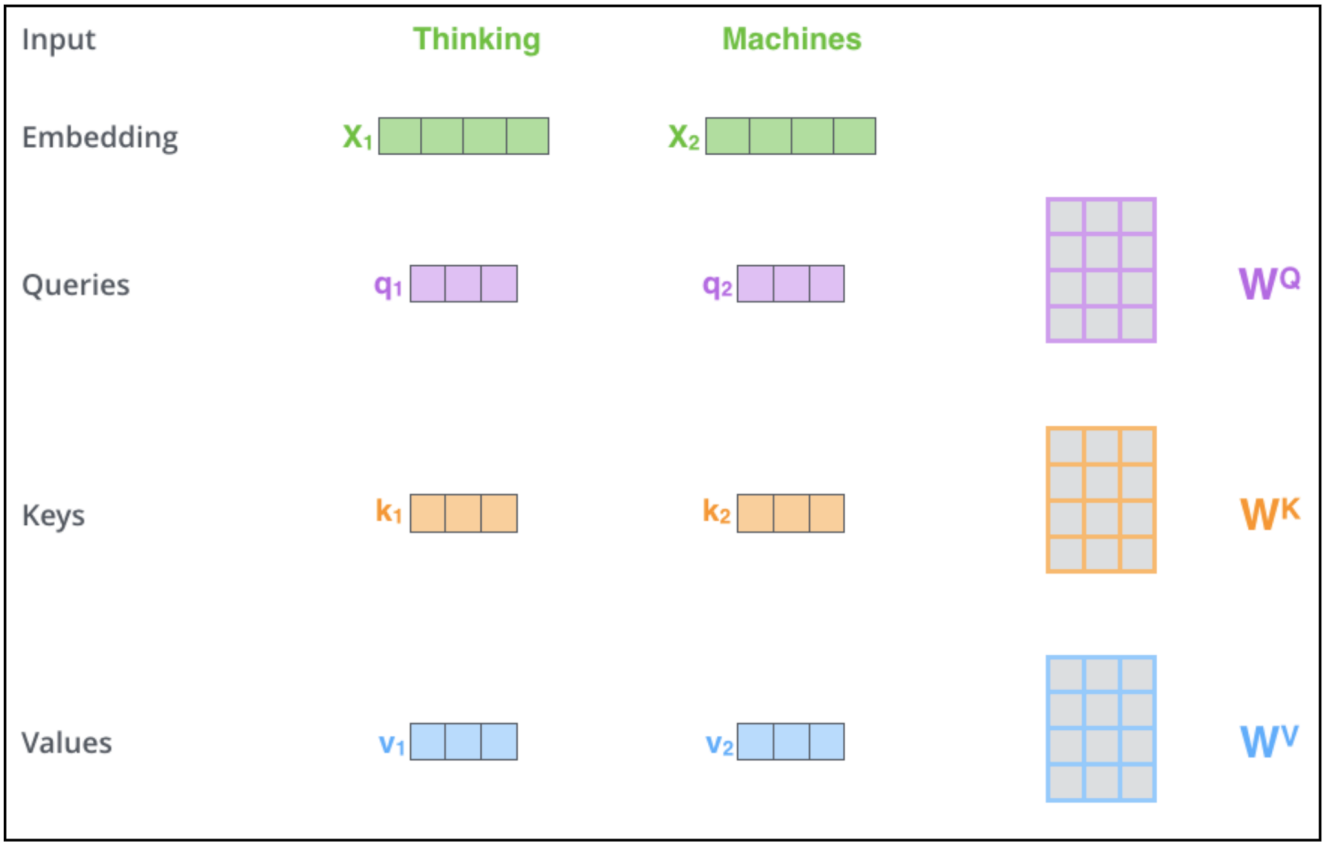


Рисунок 3 – Побудова векторів запиту, ключа та значення

Далі наша ціль вирахувати внутрішню увагу у вигляді коефіцієнтів для кожного слова у вхідному реченні відносно обраного. Коефіцієнт вказує те,

наскільки треба сфокусуватись на інших словах вхідного речення під час кодування обраного слова.

Коефіцієнт підраховується за допомогою скалярного добутку вектора запиту обраного слова та вектора ключа іншого слова. Отже, якщо ми обчислюємо внутрішню увагу слова на першій позиції, перший коефіцієнт буде скалярним добутком q1 і k1, другий — скалярним добутком q1 і k2, і т.д. Після цього, знайдені коефіцієнти слід розділити на певну константу, рівну кореню розміру вектора (це робиться для більш стабільного градієнту). Далі слід пропустити вектор скорочених результатів скалярних добутків через функцію softmax, яка нормалізує коефіцієнти так, щоб вони були додатні та в сумі давали 1. Отримані коефіцієнти вказують, якою мірою вони впливають на обране слово. Вочевидь саме це слово з високою ймовірністю буде впливати на себе найбільше, але бувають і вийнятки.

Знайшовши потрібні вагові коефіцієнти нам залишається лише просумувати вхідні вектори, попередньо помноживши їх на відповідні їм вагові значення. Отриманий вектор і буде представляти «внутрішню увагу». Такий підхід можна пояснити тим, що менш важливі слова мають менше впливати на наше уявлення про обране слово і навпаки. Повну схема знаходження внутрішньої уваги можна побачити на рис. 4:

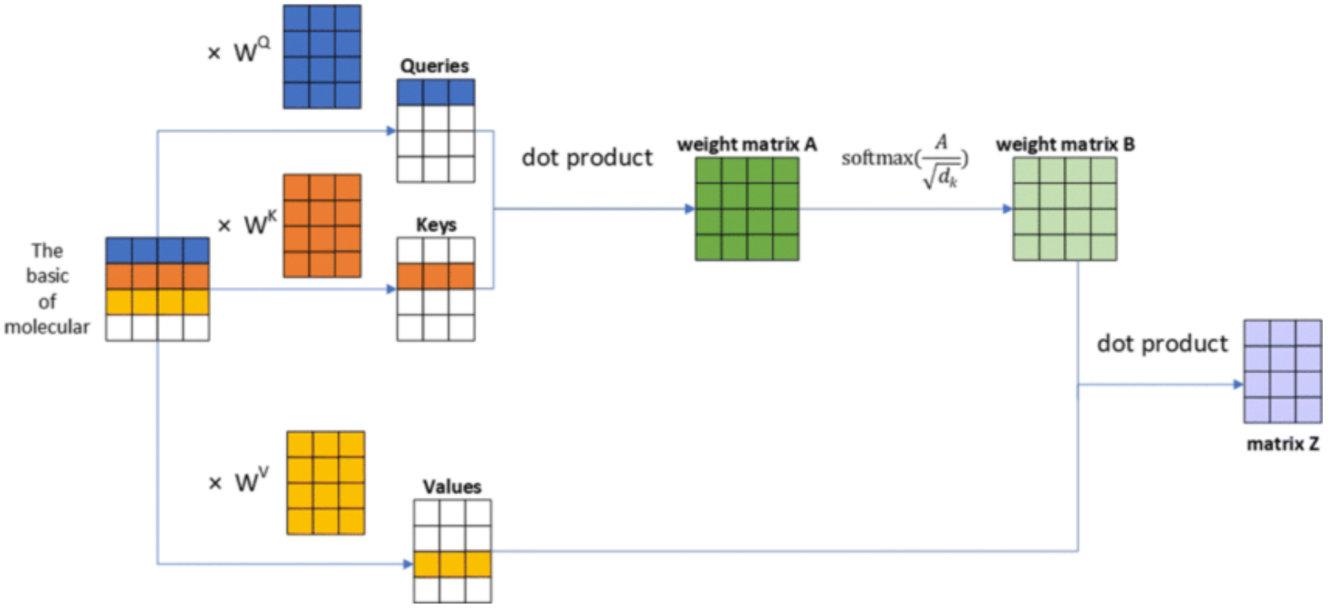


Рисунок 4 – Схема знаходження внутрішньої уваги

Зауважимо, що так знаходиться внутрішня увага у векторній формі, але в реальній моделі всі вищезазначені дії відбуваються у матричній формі, що дає можливість значно прискорити обчислення.

Проте, як виявилось, одної внутрішньої уваги все ж замало. Річ у тім, що внутрішня увага не завжди знаходить всі важливі для обраного слова слова, а частіше концентрується на окремих представниках. Через це в цьому шарі одночасно йде обрахунок декількох окремих внутрішніх уваг з різними матрицями вагів WQ, WK та WV. Їх називають головами внутрішньої уваги, яких в стандартній моделі 8. Знайшовши матриці уваг по кожній з голів, їх конкатенують і додатково множать на четверту вагову матрицю WO, в результаті чого отримуємо вихідну матрицю шару багатоголової внутрішньої уваги. Далі вектори кожного слова проходять через шар прямого розповсюдження, який, в силу незалежності його від інших векторів слів, може обраховуватись по всіх векторах паралельно. Разом маємо енкодер, яких в блоці кодування по замовчуванню 6. Повну послідовність дій одного енкодера можна побачити на рис. 5:

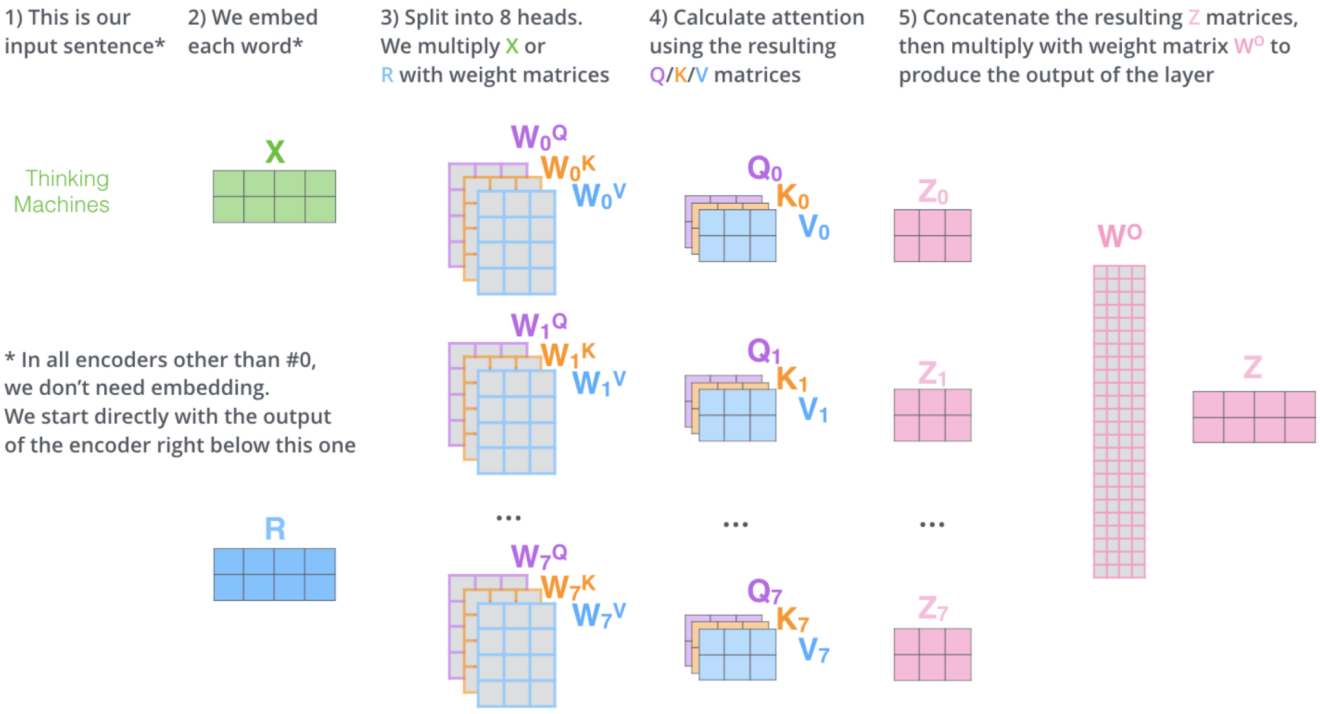


Рисунок 5 – Покрокові дії одного енкодера

1.2.2 Структурні особливості

Модель BERT (рисунок 6) можна назвати наслідником Трансформеру, бо фактично вона складається з боку кодування. Ця модель була запропонована одразу в двох варіаціях: стандартній та великій. І якщо стандартний BERT ще можна порівняти з Трансформером по розмірах, то друга варіація по дійсному величезна і досягла неймовірних результатів.

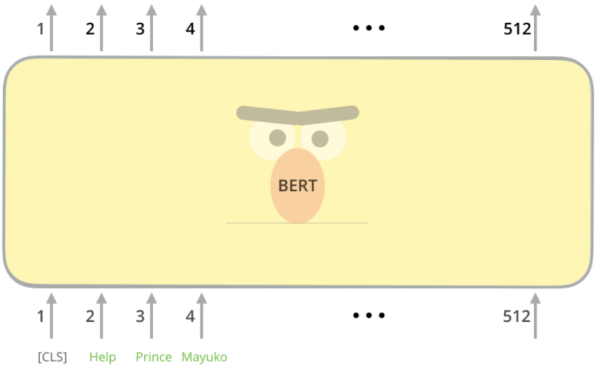


Рисунок 6 – Модель BERT

У обох версій BERT значна кількість енкодерів: 12 для базової версії та 24 для великої. Крім того, кожному слову відповідають вектори більшого розміру: 768 та 1024 відповідно, відносно 512 в Трансформері. Остання суттєва відмінність в структурі - це кількість голів внутрішньої уваги. Так в Трансформері їх всього 8, в той час, як в BERT їх 12 та 16. Всі ці відмінності, хоч і ускладнюють процес навчання, але на достатньо великих датасетах надали BERT-у просто неповторних результатів.

Є також певна відмінність у самому процесі. Так, вхідне речення доповнюється на початку спеціальним токеном [CLS]. Він потрібен для вираження суті всього речення. В подальшому весь процес проходження енкодерів один за одним повністю ідентичний Трансформеру.

На виході ми так само отримуємо множину векторів, які представляються значення всіх вхідних токенів, в тому числі токену [CLS].

1.2.3 Приклад використання

Публікаціях цих двох моделей, попередньо навчених на великих наборах даних для понад 70 мов світу, дала можливість будь-якому розробнику вбудовувати готовий обраний модуль в свої моделі обробки мови і тим самим значно полегшувати та пришвидшувати процес написання та навчання власної моделі.

Так, приведемо один з найпоширеніших прикладів використання BERT - класифікація фрагменту тексту, а точніше чи є електронний лист спамом, чи ні. Для цього нам достатньо поверх першого вектору виходу BERT (який містить в собі загальний контекст речення), накласти один додатковий повнозв‘язний шар класифікатору. Оскільки нам треба визначити лише, чи є повідомлення спамом, чи ні, нам достатньо мати всього два вихідних нейрон в класифікаторі, які попередньо пропускаються через функцію софтмакс (дивись рисунок 7).

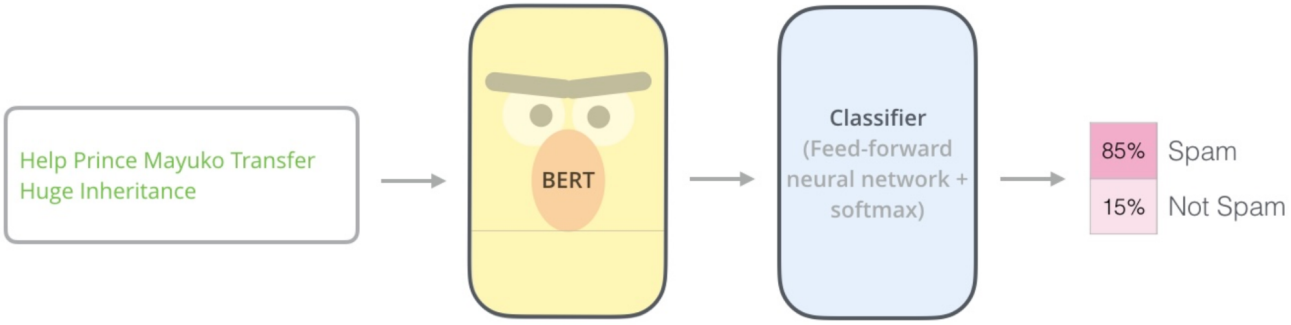


Рисунок 7 – Класифікація спами за допомогою BERT

Далі нам достатньо просто знайти відповідний датасет для нашої задачі і навчити по ньому лише наш один шар нейронної мережі, пропускаючи кожен електронний лист попередньо через BERT, що надзвичайно спрощує розв‘язання задачі класифікації.

РОЗДІЛ 2. ТЕОРЕТИЧНА МОДЕЛЬ

З використанням моделі BERT, можна значно швидше та простіше створити чат-бота, для відповідей на питання, аналоги яких вже були чаті. Тому я пропоную наступну теоретичну модель, що складається з кількох основних етапів: ідентифікація питання, знаходження подібного, ідентифікація відповіді на питання, перевірка відповіді. Розберемо кожен етап окремо.

2.1 Ідентифікація питання

Вочевидь, оскільки наш чат-бот має надавати відповіді на питання замість людини, все починається з розпізнавання чи є повідомлення питанням. Можна, звісно, було б обійтись звичайною перевіркою наявності знака питання, але постає проблема риторичних запитань чи відсутності знака питання, як такого, через неуважність користувача. Тому тут найбільш вдалим розв‘язком, на мою думку, є побудова нейронної мережі класифікатору з використанням BERT, для класифікації чи є повідомлення питанням, чи ні. Втім наявність знаку питання теж має бути вагомим сигналом.

В якості матеріалу для навчання мережі можна використати доволі популярний датасет SQuAD (The Stanford Question Answering Dataset). На ньому можна попередньо навчити класифікатор та використовувати вже готову модель.

Якщо при перевірці повідомлення ми класифікували його, як питання, ми запам’ятовуємо це повідомлення, як питання, зберігаючи його ідентифікатор.

2.2 Специфічна тематика

У наступних двох пунктах ми будемо перевіряти схожість питань та відповідей між собою. Але оскільки чати зачасту мають специфічну тематику, то можна припускати, що з високою ймовірністю в повідомленнях будуть часто вживатись слова цієї ж спецефічної тематики. Тоді логічно припускати, що в

подібних повідомленнях зачасту мають зустрічатися однакові специфічні слова для обраного чату. Більше того, для питання та відповіді на нього теж має справджуватись наведена властивість. На мою думку, цим варто скористатися для покращення працездатності остаточної моделі.

Залишається питання знаходження цих слів, де ми прибіґнемо до певних евристичних методів. Логічно припустити, що тематичні слова, притаманні обраному чату, мають зустрічатись у ньому частіше, ніж у загальних текстах. Можна було б взяти статистику слів та обрати найпопулярніші, але тоді переважну частину слів займуть часто вживані слова в побудові речення: займенники (I, this, it), артиклі (a, the), службові дієслова (is, can, will) та інші. Тому я пропоную спершу взяти статистику слів в обраному чаті, а потім відняти від неї статистику слів зі звичайних діалогів (тут можна скористатись тим же SQuAD датасетом), враховуючи співвідношення розмірів наборів слів. В такий спосіб можна легко відсіяти часто вживані слова в обраному чаті від часто вживаних слів загалом.

Серед отриманої статистики вживаності слід взяти топ max_dictionary_size (параметр) найуживаніших слів і вважати їх специфічними словами чату. У сукупності будемо називати ці слова словником чату.

2.3 Пошук подібних питань

Справедливо припускати, що в подібних питань має бути подібний зміст. Крім того, слова зі словнику чату мають орієнтовно однаковою мірою бути наявними та відсутніми в обох чатах. Тоді для пошуку подібних питань можна зробити наступні дії.

Як тільки ми натикаємось на нове питання, ми починаємо пошук серед останніх question_search_num (параметр) питань у базі. Для кожного питання будується його інтерпретація у вигляді двох векторів: вектору змісту питання, знайденого за допомогою BERT, та вектору використаних слів зі словнику розміру dictionary_size з 0 та 1. Далі ми окремо порівнюємо відповідні вектори поточного та старого питань, шляхом пропуску попередньо нормалізованих векторів через

скалярний добуток. Як відомо з математики, скалярний добуток двох векторів рівний добутку їх довжин на косинус кута між ними. Відповідно, знайдені нами величини будіть рівні косинусам кутів між відповідними векторами, а тому чим більше значення косинусу, тим вектори подібніші один одному. Такий підхід називається косинусна подібність [4] та часто використовується в аналізі даних.

Отримавши два значення косинусу подібності векторів змісту та векторів слів зі словника поточного та старого питань, нам залишається лише перемножити їх на вагові коефіцієнти question_similarity_bert_weight і 1-question_similarity_bert_weight відповідно, та просумувати. Так цим параметром ми задаємо наскільки впливовим є вектор змісту від BERT у порівнянні з вектором слів зі словнику. У результаті отримуємо шукане значення подібності питань.

Так, серед усіх перебраних питань ми залишаємо лише ті, чиє значення подібності з поточним питанням є вищим за деяке порогове значення подібності question_similarity_threshold (параметр). Саме ці питання ми вважаємо подібними нашому та відповіді саме на деякі з цих питань ми і запропонуємо нашому користувачу. Пошук декількох питань робиться з міркувань можливості неповної відповіді на поставлене запитання лише однією минулою відповіддю - нерідко трапляються ситуації, коли для вирішення питання треба перечитати декілька різних джерел.

2.4 Пошук відповіді

Знайшовши подібні поточному питання, нам треба знайти відповіді на них. Так, якщо ми вже опрацьовували деяке питання, то мали б зберегти ідентифікатор відповідь на нього. Якщо ж ідентифікатор відповідь відсутній, ми починаємо пошук відповіді.

Спершу розберемось, що ми вважаємо за відповідь. Частіше за все у чатах відповідь йде відразу після питання або десь неподалік. Крім того, в чатах тех підтримок та їм подібним, повідомлення від модераторів чату зазвичай є відповідями на питання, а тому їх варто розглядати окремо від загальної маси

повідомлень. При цьому, як вже зазначалось у відповіді ймовірно мають повторюватись слова зі словнику чату, які були у питанні, а також тематика питання та відповіді мають бути подібним.

Підсумувавши, пошук відповіді на питання можна робити наступним чином. Перебирати ми будемо лише ті повідомлення, які не є питаннями. При цьому до розгляду потрапляють max_moderator_answers (параметр) перших повідомлень від модераторів та max_user_answers (параметр) перших повідомлень від звичайних користувачів, які йдуть після питання, для якого шукається відповідь. Також, якщо це передбачено інтерфейсом чату, слід приділяти особливу увагу повідомленням, що “реплаять” розглянуте питання, а тому також будемо розглядати max_replies повідомлень-реплаїв. Для всіх розглянутих повідомлень вираховується значення подібності до питання аналогічно тому, як рахується значення подібності двох питань. Єдина відмінність має заключались у ваговому коефіцієнті answer_similarity_bert_weight, який ми використаємо для сумування косинусних подібностей векторів змісту та векторів слів зі словнику. Він має бути відмінним від question_similarity_bert_weight, оскільки тепер ми порівнюємо вже не два питання, а питання та відповідь, а тому загальний зміст повідомлень може більше відрізнятись. Проте варто зазначити, що ці два параметри скоріш за все мають бути близькими за значенням. Знайшовши значення подібності з питанням для кожного розглянутого повідомлення, множимо це значення на деякі підсилюючі коефіцієнти, в залежності від того, чи було повідомлення надіслане адміністратором та чи було воно реплаєм до питання, та маємо значення впевненості у відповіді. З отриманих значень обираємо найбільше. Якщо воно більше за деякий поріг answer_threshold (параметр), то відповідне повідомлення і будемо вважати відповіддю на питання, і збережемо його ідентифікатор разом зі значенням впевненості у відповіді. Якщо ж значення буде меншим, то вважаємо, що відповіді на це питання ще не було.

Тут можуть бути два варіанти. Перший варіант - це коли питання ще нове та на нього не встигли відповісти. Про це буде свідчити те, що не було ще max_moderator_answers чи max_user_answers повідомлень від модераторів чи

звичайних користувачів після питання, або що не було ще max_replies реплаїв на питання. У такому випадку це питання поки пропускається. Другий варіант - це вже досить старе питання (протилежний випадок до першого), і тоді ми вважаємо, що на питання так ніхто і не відповів, а тому ми видаляємо це питання зі списку питань.

2.5 Відповідь користувачу

Якщо існує хоча б одне подібне питання до нового і для хоча б одного з цих подібних питань є відповідь, чат-бот має сформувати та відправити відповідь користувачу. Для цього визначається топ top_answers_max_size (параметр) найкращих відповідей. Це робиться шляхом перемноження значення подібності між новим питанням і питанням зі списку подібних питань та значенням впевненості у відповідях на ці питання, та подальшим вибором відповідей, яким відповідають найбільші добутки. Цей топ відповідей і формують основу повідомлення-відповіді у вигляді списку посилання на відповідні повідомлення. Також бот просить користувача повідомити, чи була ця відповідь задовільною та корисною для нього, пропонуючи натиснути одну з двох кнопок.

Якщо користувач відповідає ствердно, то ми одразу додаємо ідентифікатор свіжо сформованої відповід, як відповідь на поточне питання з впевненістю 1. У протилежному випадку - бот кличе модераторів, щоб вони відповіли користувачу.

2.6 Фонове покращення

В даній конфігурації ми двічі припускаємо, що подібні повідомлення будуть мати подібні вектори змістів та слів зі словника, а сама подібність визначається через скалярний добуток. І хоч це доволі ефективний та простий спосіб порівняння, він не є найкращим. Подібність, особливо в специфічній темі, є надто розмитим питанням і чіткі математичні формули ймовірно будуть не кращими. Значно краще вирішити це питання можуть нейронні мережі на цих наборах

векторів, але для їх навчання не існує датасетів, бо для кожного чату він має бути свій.

Як варіант виходу з останнього я пропоную вчити нейронну мережу на фоні. Спершу ми будемо припускати, що вище зазначений алгоритм дій пошуку відповідей є ідеально правильним і для кожного нового питання будемо дивитись, яку найкращу відповідь дає цей алгоритм, і використовувати ці дані для навчання фонових нейронних мереж. Так з часом система дійдемо до того, що фонові нейронні мережі зможуть замістити чіткі алгоритми знаходження подібних повідомлень і даватимуть такі ж відповіді. З цього моменту ми перестаємо використовувати чіткі алгоритми, заміщаючи їх фоновими нейронними мережами.

Крім того, у сформованих відповідях ми додаємо ще одну кнопку, яка вказуватиме, що знайдені повідомлення не є задовільними відповідями, але дійсно в цих частинах чату розглядались потрібні користувачу питання. Це потрібно для подальшого навчання нейронних мереж в ході їх використання. Як результат - ми матимемо чат-бота, який буде натренований для надання відповідей у обраному конкретному чаті.

Використовуючи саме такий підхід, ми одночасно матимемо чат-бота, який одразу готовий виконувати свої обов‘язки, але при цьому має перспективу краще відповідати на поставлені запитання у майбутньому.

2.7 Константи

Всі зазначені в цьому розділі константи, а саме: max_dictionary_size, question_search_num, question_similarity_bert_weight, question_similarity_threshold, max_moderator_answers, max_user_answers, max_replies, answer_similarity_bert_weight, answer_threshold, top_answers_max_size — параметри, які слід корегувати у ході побудови чат-боту, а також підлаштовувати під конкретні чати в залежності від їх особливостей.

РОЗДІЛ 3. ДЕТАЛІ РЕАЛІЗАЦІЇ

В цьому розділі будуть розглянуті деякі специфіки реалізації та основні частини програмного коду. Повний код наведено у додатку А, а також його можна знайти за посиланням: https://github.com/albashuk/QAChatBot.

Схематично весь проект можна зобразити наступним чином (рисунок 8):

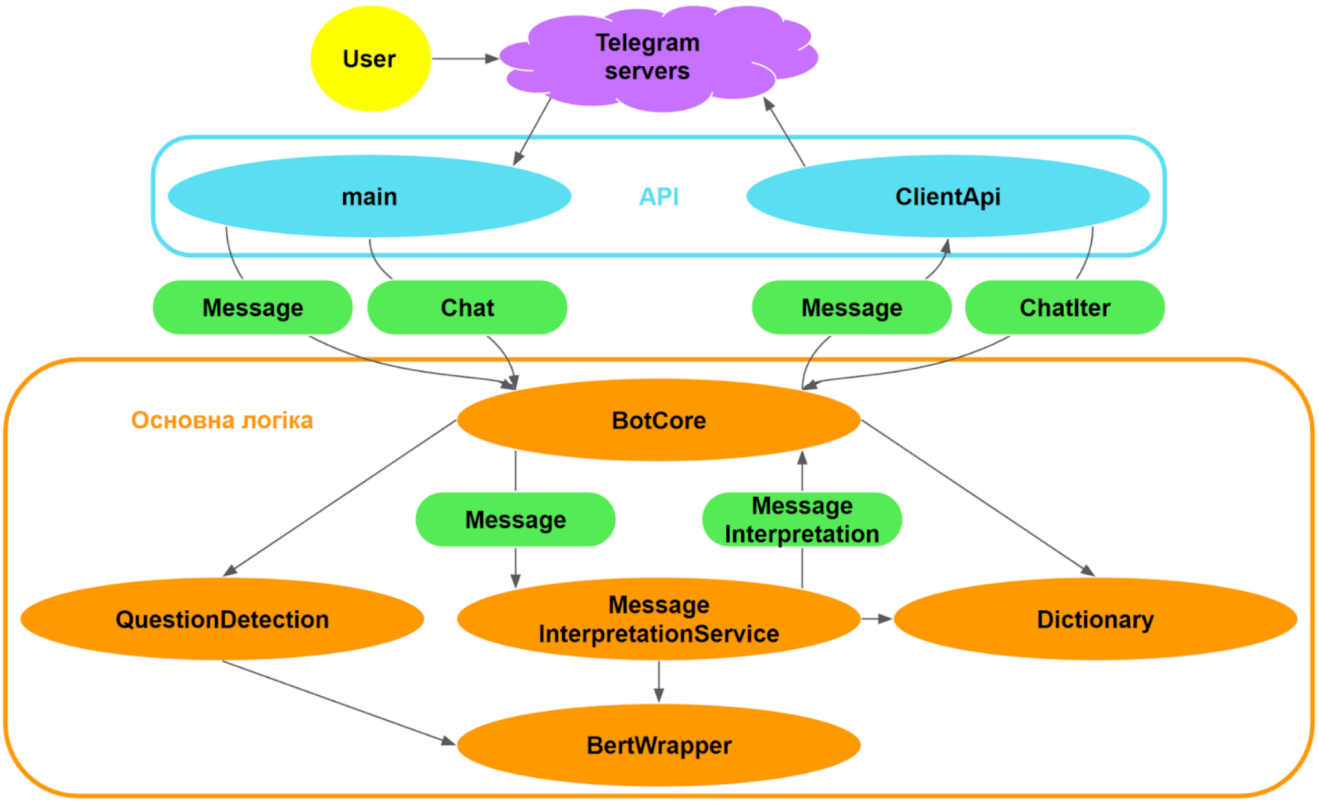


Рисунок 8 – Схема чат-боту

Бот розроблявся так, щоб максимально розділити саму логіку роботи бота та API платформи чату. В поточній реалізації бот створений мовою Python для платформи Telegram, але завдяки такому розділенню його можна легко інтегрувати й на інші платформи.

Сам програмний код бота можна розділити на окремі складові, серед яких можна виділити три основні: логіка роботи бота, API платформи та проміжні класи для групування інформації. Також є окремий клас properties, в якому прописані всі параметри, та клас parser, через який відбуваються всі розбиття

тексту на слова. У цьому розділі буде розглянута перша складова, а саме логіка роботи бота, як його ядра. Складова логіки роботи складається з наступних класів: BertWrapper, QuestionDetection, MessageInterpretationService, Dictionary та BotCore. Розглянемо їх детальніше.

3.1 BertWrapper

BertWrapper - допоміжний клас, що, як можна здогадатись із назви, виконує векторизацію набору рядків завдяки одному з різновидів моделі BERT. Цей клас використовує вже навчені моделі для токенізації та обробки рядків, та повертає векторне представлення загального змісту для кожного з рядків. Змінюючи значення параметра *properties.bert.version*, можна вибрати бажану версію BERT.

```
class BertWrapper:
    def __init__(self, device) -> None:
        self.__device = device
        self.__tokenizer =
BertTokenizerFast.from_pretrained(properties.bert.version)
        self.__bert =
AutoModel.from_pretrained(properties.bert.version).to(device)
        self.__bert.eval()
        for param in self.__bert.parameters():
            param.requires_grad = False

    def __call__(self, text, max_padding: bool = False):
        sent_ids, mask = self.__sentToIds(text, max_padding)
        return self.__bert(sent_ids, mask)

    def __sentToIds(self, texts, max_padding: bool):
        args = {
            'batch_text_or_text_pairs': texts,
            'truncation': True,
            'return_token_type_ids': False}
        if max_padding:
            args['max_length'] = properties.bert.token_max_seq_len
            args['padding'] = 'max_length'
        tokens = self.__tokenizer.batch_encode_plus(**args)

        return torch.tensor(tokens['input_ids']).to(self.__device),
torch.tensor(tokens['attention_mask']).to(self.__device)
```

3.2 QuestionDetection

В модулі QuestionDetection реалізовано логіку вирішення, чи є речення питанням, чи ні. Для цього речення переводиться у векторне представлення за допомогою BERT. Після цього йде повнозв'язний шар, що переводить це векторне представлення у два сигнали, попередньо пропустивши їх через функцію Softmax. Залежно від того, який сигнал більший, і визначається, чи є речення питанням чи ні:

```
class QuestionDetection():

    class __Module(nn.Module):

        class __ModuleEnd(nn.Module):
            def __init__(self) -> None:
                super().__init__()

                self.__drop = nn.Dropout(properties.dropout_prob)
                self.__ln = nn.Linear(properties.bert.out_size, 2)
                self.__softmax = nn.Softmax(dim=1)

            def forward(self, x):
                x = self.__drop(x)
                x = self.__ln(x)
                x = self.__softmax(x)

                return x
    …
    def isQuestion(self, sentence) -> bool:
        self.__module.eval()
        has_question_mark = 1.0 if "?" in sentence else -1.0
        question_prob = self.__module([sentence])[0]
        return question_prob[0].item() <= question_prob[1].item() +
has_question_mark * properties.question_mark_weight
```

Модель була навчена на датасеті SQuAD.

3.3 MessageInterpretationService

Модуль MessageInterpretationService відповідає за побудову векторного представлення повідомлень у чаті, об'єднуючи вектор змісту всього повідомлення, отриманого від BertWrapper, та вектор використаних у повідомленні слів зі словнику.

```
class MessageInterpretationService:
    …
    def toInterpretation(self, message: str, dictionary: Dictionary)
-> MessageInterpretation:
        bert_vec = self.__bert([message])[1]
        dict_vec = self.__dictionaryUse(message, dictionary)

        return MessageInterpretation(bert_vec, dict_vec,
dictionary.getVersion())
```

Також цей модуль для двох інтерпретацій повідомлень визначає їх подібність за допомогою косинусної подібності:

```
class MessageInterpretationService:
    …
    def similarity(self,
                   interp1: MessageInterpretation,
                   interp2: MessageInterpretation,
                   bert_multiplier: float) -> float:
        bert_cos = self.__bertCosSim(interp1.bert_vec,
interp2.bert_vec)
        if interp1.dict_vec is None and interp2.dict_vec is None:
            return bert_cos
        else:
            dict_cos = self.__dictCosSim(interp1.dict_vec,
interp2.dict_vec)
            return (bert_multiplier * bert_cos + (1 -
bert_multiplier) * dict_cos).item()
```

3.4 Dictionary

Модуль Dictionary відповідає за створення словнику чату, шляхом збору статистики вживання слів та подальшого виділення специфічних слів. Для збору

статистики вживаності слів використовується метод increaseWordCurrency, який збільшує лічильник вживаності слова:

```
class Dictionary:
    …
    def increaseWordCurrency(self, word: str):
        word = wordToStem(word)
        if self.__update_enabled:
            if self.__words_currency.get(word) is None:
                self.__words_currency[word] = 0
            self.__words_currency[word] += 1
            self.__words_count += 1
```

Як тільки вирішується, що вже було вжито достатню кількість слів, щоб вважати поточний словник застарілим, словник можна оновити за допомогою методу update. Цей метод від нормалізованої статистики використаних слів у чаті віднімає нормалізовану статистику вживаності загальних слів, помножену на деякий ваговий коефіцієнт, і в отриманій різниці обирається топ з не більше max_dictionary_size слів, чиї значення в цій різниці більші деякого порогу. Обрані слова і утворюють словник нової версії, і в такий спосіб можна завжди мати актуальний для чату словник:

```
class Dictionary:
    …
    def update(self,
               max_size: int = properties.dictionary.default_size,
               common_words_weight: float =
properties.dictionary.default_common_words_weight,
               threshold: float =
properties.dictionary.default_update_threshold):
        if self.__update_enabled:
            if len(self.__common_words_usage) != 0:
                words_usage = []
                for word in self.__words_currency:
                    common_words_usage =
self.__common_words_usage[word] \
                        if self.__common_words_usage.get(word) is
not None \
                        else 0
                    word_usage = self.__words_currency[word] /
self.__words_count \
                                 - common_words_usage *
```

```
common_words_weight
                    if word_usage > threshold:
                        words_usage.append((word_usage, word))
                if len(words_usage) > 0:
                    words_usage.sort(reverse=True)
                    self.__dictionary = {w[1]: i for i, w in
enumerate(words_usage[:max_size])}
                    self.__dictionary_version += 1
```

В якості бази для статистики вживаності загальних слів використовуються тексти з датасету SQuAD.

3.5 BotCore

Основний модуль зветься BotCore, і він відповідає за обробку нових повідомлень та початкову ініціалізацію для чатів аналогічно до теоретичної. Проте, через поширені технічні обмеження чатів, особливо для ботів, через які перегляд історії чату є неможливим, остаточна реалізація бота виконує ті самі дії, тільки в іншому порядку. Замість знаходження відповідей на питання, коли в цьому виникає потреба, бот в режимі реального часу перевіряє, чи могло нове повідомлення бути відповіддю на деяке з минулих питань та зберігає його за потреби, як відповідь:

```
class BotCore:
   …
   async def messageProcessing(self, message: Message,
with_answering: bool = True):
       chat = self.__chatById(message.id.chat_id)
       user_type = "moderator" if await
self.__isFromModerator(message) else "user"
       …
       if self.__isQuestion(message):
           …
       else:
           if message.reply_id is not None:
               reply = chat.questions.get(message.reply_id)
               if reply is not None and isinstance(reply,
QuestionSummary):
                   reply.replies += 1
                   if reply.replies <= properties.max_replies:
```

```
                        self.__answerChecking(reply, message,
self.__messageWeight(user_type, True))

            for question in chat.questions_queue[user_type]:
                self.__answerChecking(question, message,
self.__messageWeight(user_type))
```

Така послідовність дій також є вигіднішою і з точки зору продуктивності, оскільки завантажує бота при обробці повідомлень, які не є питанням, на відмінну від його простою в теоретичній конфігурації, та зменшує складність побудову відповіді на поставлене користувачем питання, чим значно прошвидшує відповідь бота користувачу.

Ще одним покращенням продуктивності є збереження інтерпретації питань разом з питаннями, завдяки чому при перевірці подібності не треба постійно заново перераховувати інтерпретацію питання, як повідомлення.

Останньою складовою обробки повідомлень є оновлення статистик словнику словами з нового повідомлення, та оновлення самих слів словнику згідно до нових значень статистик раз на 1000 повідомлень, що забезпечує підтримку актуальності словника:

```
class BotCore:
    …
    async def messageProcessing(self, message: Message,
with_answering: bool = True):
        …
        self.__updateDictionary(message)
    …
    @classmethod
    def __updateDictionary(cls, message: Message):
        chat = cls.__chatById(message.id.chat_id)
        words = parseOnWords(message.message)
        for word in words:
            chat.dictionary.increaseWordCurrency(word)
        if (chat.message_count["user"] +
chat.message_count["moderator"]) % 1000 == 0:
            chat.dictionary.cleanCurrency()
            chat.dictionary.update()
```

При цьому, відбувається постійний контроль версій інтерпретацій повідомлень, і в разі виявлення застарілої інтерпретації повідомлення, це інтерпретація оновлюється згідного до останньої версії словнику.

Цього здебільшого достатньо для функціонування бота. Проте для того, щоб словник набув практичності треба час. Тому, на випадок, якщо для бота все ж є можливість переглядати історію чату, він здатний ініціалізувати список питань з відповідями на них, а також словник чату, аналогічно опрацювавши певну кількість останніх повідомлень чату:

```
class BotCore:
    …
    async def initChat(self, chat_id: Chat.Id):
        if self.__chats.get(chat_id) is not None:
            self.__log.warning(f"Chat ({chat_id.value()}) is already
init")
            return
        self.__chats[chat_id] = Chat(chat_id, Dictionary(set(),
True, True))

        if not self.__clientApi.hasAccessToChatHistory(chat_id):
            self.__log.warning(f"Client hasn't access to chat
({chat_id.value()}) history")
            return

        try:
            chatIter = await self.__clientApi.buildIter(chat_id,
properties.history_limit)
            while True:
                message = await chatIter.next()
                await self.messageProcessing(message, False)
        except StopAsyncIteration:
            pass
```

Конкретно у випадку платформи Telegram, вона забороняє ботам можливість перегляду історії чату. Проте, якщо надати боту можливість авторизуватися через профіль справжнього користувача, це обмеження зникає і в бота з’являється доступ до перегляду історії чату.

3.6 Приклади використання

Для реалізованої моделі було зімітовано дві ситуації, які можна спостерігати на рисунку 9 та на рисунку 10.

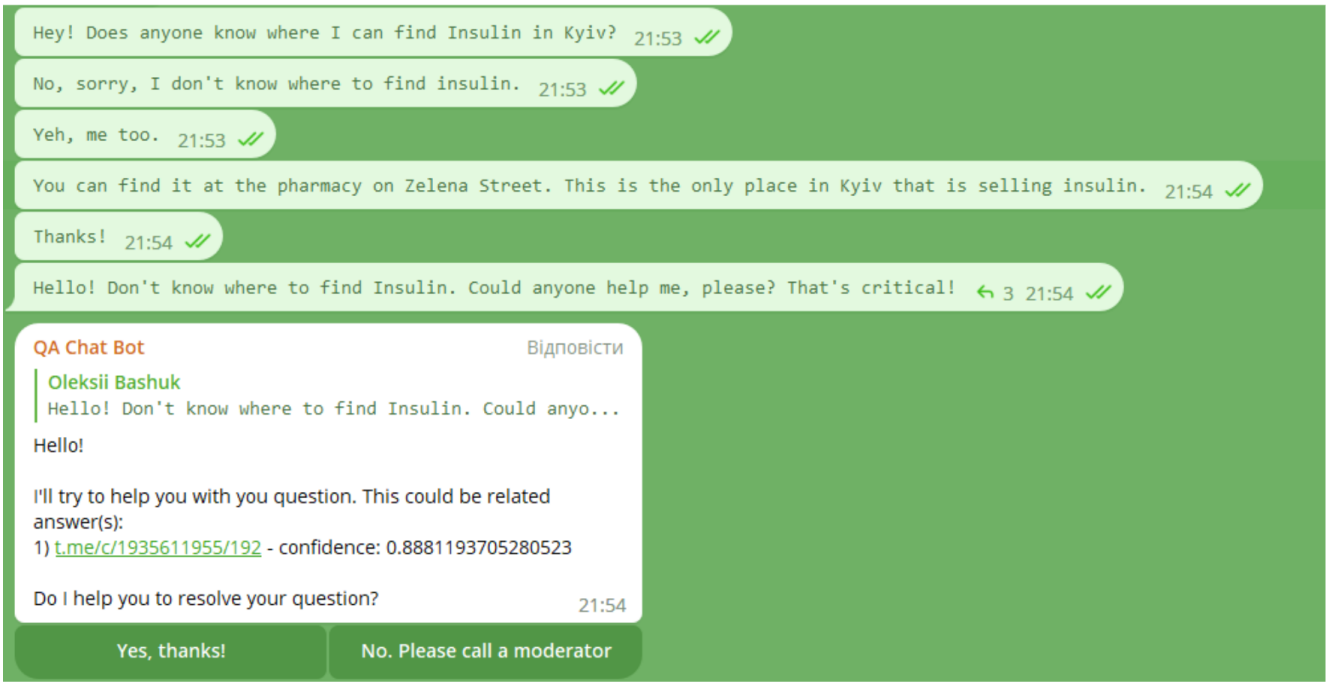


Рисунок 9 – Приклад зімітованого чату обміну ліків в Києві

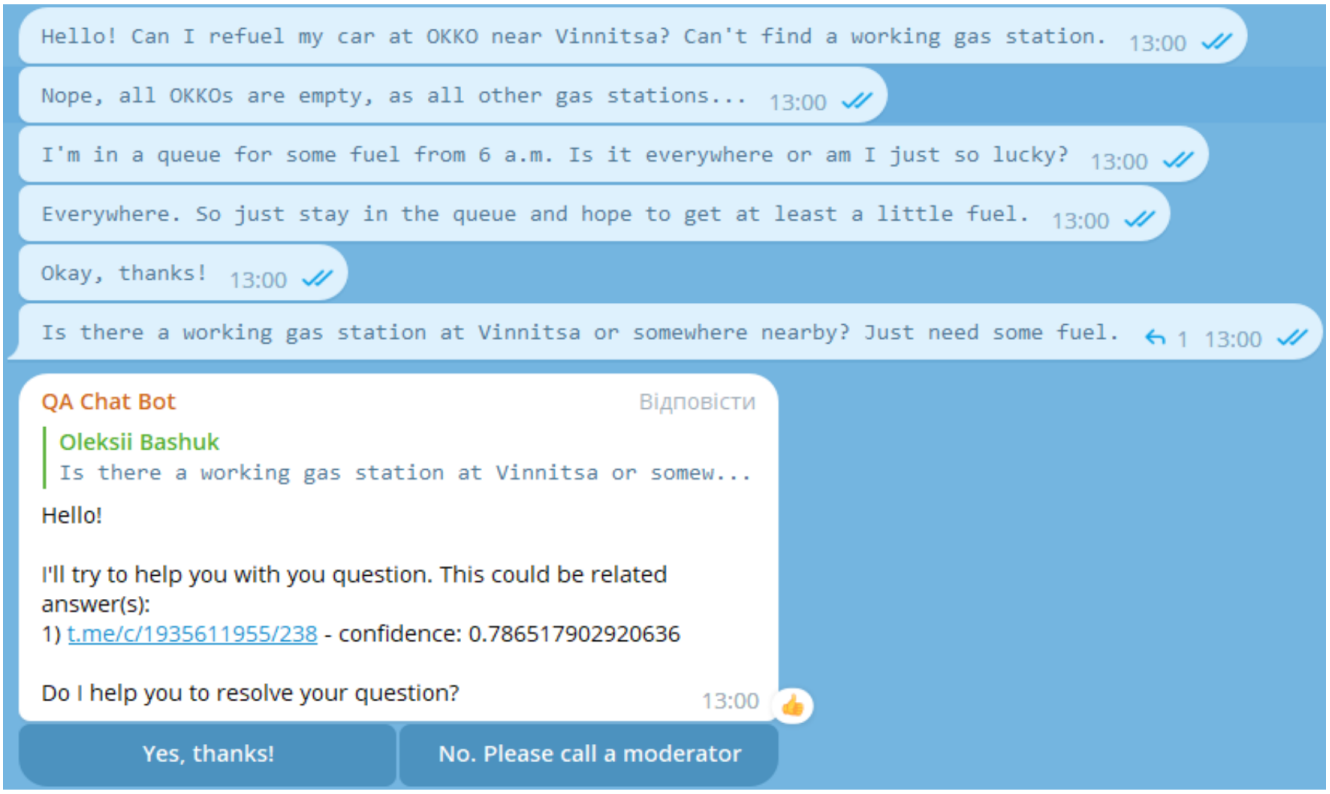


Рисунок 10 – Приклад зімітованого чату з тематикою пошуку бензину в Україні

3.7 Пропозиції подальшого розвитку

Розроблена модель виконує всі поставлені перед нею завдання, та має і простір для подальшого розвитку. Серед теоретичними покращеннями моделі є:

- використання баз даних, що є необхідним при масштабуванні боту на велику кількість чатів;
- кластеризація питань за їх подібністю;
- використання більшої кількості сигналів для встановлення відповіді на питання, що має підвищити загальну якість знаходження потрібної відповіді, а отже і якості роботи боту загалом;
- використання фонових нейронних мереж згідно до підрозділу 2.6, що також має покращувати якість згенерованих відповідей на питання з плином часу;
- використання технологій GPT для генерації повідомлення-відповіді на базі знайдених відповідей, щоб покращити досвід користувача при контакті з чат-ботом.

ВИСНОВОК

В цій роботі були висвітлені основи нейронних мереж, а також було розібрано, за яким принципом працює запропонована компанією Google модель BERT. В ході розбору, було показано, як саме модель пов‘язує між собою слова, для розуміння значення кожного окремого слова в залежності від інших, а також, як модель будує розуміння загального змісту на цих зв‘язках.

Крім того, із розумінням, як саме працює BERT, було побудовано теоретичну модель чат-боту для відповідей на питання, яка активно використовує BERT, та на її основі було розроблено конкретну реалізацію мовою Python для чатів платформи Telegram. Модель побудована так, щоб максимально ефективно знаходити подібні питання та відповіді на них, формувати відповіді користувачу та реагувати на реакцію користувача в залежності від того, чи сподобалась йому сформована відповідь, чи ні. Сама ж реалізація спроектова таким чином, щоб максимально розділити API платформи та логіку поведінки бота, а тому інтеграція бота на інші платформи є можливою та простою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IBM Cloud Education. Neural Networks. Доступно за: https://www.ibm.com/cloud/learn/neural-networks

2. WikipediA. BERT (language model). Доступно за: https://en.m.wikipedia.org/wiki/BERT_(language_model)

3. J. Alammar. The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) [Blog post]. Доступно за: http://jalammar.github.io/illustrated-bert/

4. D. Jurafsky, J. H. Martin. Speech and Language Processing. Chatbots and Dialogue Systems. Доступно за: https://web.stanford.edu/~jurafsky/slp3/24.pdf

5. J. Alammar. The Illustrated Transformer [Blog post]. Доступно за: https://jalammar.github.io/illustrated-transformer/

ДОДАТОК А

Повний код реалізації чат-бота

В цьому додатку наведений повний код реалізації чат-боту, який також можна знайти за посиланням: https://github.com/albashuk/QAChatBot.

Код файлу ***BertWrapper.py***:

```
from properties import properties
from transformers import AutoModel, BertTokenizerFast

import torch


class BertWrapper:
    def __init__(self, device) -> None:
        self.__device = device
        self.__tokenizer =
BertTokenizerFast.from_pretrained(properties.bert.version)
        self.__bert =
AutoModel.from_pretrained(properties.bert.version).to(device)
        self.__bert.eval()
        for param in self.__bert.parameters():
            param.requires_grad = False

    def __call__(self, text, max_padding: bool = False):
        sent_ids, mask = self.__sentToIds(text, max_padding)
        return self.__bert(sent_ids, mask)

    def __sentToIds(self, texts, max_padding: bool):
        args = {
            'batch_text_or_text_pairs': texts,
            'truncation': True,
            'return_token_type_ids': False
        }
        if max_padding:
            args['max_length'] = properties.bert.token_max_seq_len
            args['padding'] = 'max_length'
        tokens = self.__tokenizer.batch_encode_plus(**args)

        return torch.tensor(tokens['input_ids']).to(self.__device),
torch.tensor(tokens['attention_mask']).to(self.__device)
```

Код файлу ***BotCore.py***:

```
import logging

import torch

from BertWrapper import BertWrapper
from Chat import Chat
from ClientApi import ClientApi
from Dictionary import Dictionary
from Message import Message
from MessageInterpretation import MessageInterpretation
from MessageInterpretationService import
MessageInterpretationService
from QuestionDetection import QuestionDetection
from QuestionSummary import QuestionSummary
from parser import parseOnWords, parseOnSentences
from properties import properties


class BotCore:
    # logs
    __log = logging.getLogger('__name__')

    # modules
    __device = torch.device("cuda:0" if torch.cuda.is_available()
else "cpu")
    __bert = BertWrapper(__device)
    __questionDetection = QuestionDetection(__device, __bert)
    __messageInterpretationService =
MessageInterpretationService(__device, __bert)

    # data
    __chats = {}

    def __init__(self, clientApi: ClientApi) -> None:
        self.__clientApi = clientApi

    async def messageProcessing(self, message: Message,
with_answering: bool = True):
        chat = self.__chatById(message.id.chat_id)
        user_type = "moderator" if await
self.__isFromModerator(message) else "user"
        chat.message_count[user_type] += 1
        while len(chat.questions_queue[user_type]) > 0 \
                and not
```

```
chat.questions_queue[user_type][0].waitingForAnswer(chat.message_co
unt[user_type], user_type):
            chat.questions_queue[user_type].popleft()

        self.__updateDictionary(message)

        if self.__isQuestion(message):
            question = QuestionSummary(message.id,
                                       message.message,

chat.message_count["user"],

chat.message_count["moderator"])
            chat.questions_queue["user"].append(question)
            chat.questions_queue["moderator"].append(question)
            chat.questions[message.id] = question

            if with_answering:
                answers = []
                for oldQuestionId in list(chat.questions)[:-1]:
                    oldQuestion = chat.questions[oldQuestionId]
                    if oldQuestion.answer_id is not None:
                        similarity =
self.__questionSimilarity(question, oldQuestion)
                        if similarity >=
properties.question_similarity_threshold:
                            answers.append((similarity *
oldQuestion.answer_confidence, oldQuestion.answer_id))
                    else:
                        if not oldQuestion.waitingForReply() \
                                and not
oldQuestion.waitingForAnswer(chat.message_count["user"], "user") \
                                and not
oldQuestion.waitingForAnswer(chat.message_count["moderator"],
"moderator"):
                            chat.questions.pop(oldQuestionId)

                top_answers =
sorted(answers)[:properties.top_answers_max_size]
                return self.__clientApi.buildRespond(top_answers) if
len(top_answers) != 0 else None
        else:
            if message.reply_id is not None:
                reply = chat.questions.get(message.reply_id)
                if reply is not None and isinstance(reply,
```

```
QuestionSummary):
                    reply.replies += 1
                    if reply.replies <= properties.max_replies:
                        self.__answerChecking(reply, message,
self.__messageWeight(user_type, True))

            for question in chat.questions_queue[user_type]:
                self.__answerChecking(question, message,
self.__messageWeight(user_type))

    async def acceptGeneratedAnswer(self, answer_id: Message.Id,
question_id: Message.Id):
        chat = self.__chatById(answer_id.chat_id)
        question = chat.questions[question_id]
        question.answer_id = answer_id
        question.answer_confidence = 1

    async def initChat(self, chat_id: Chat.Id):
        if self.__chats.get(chat_id) is not None:
            self.__log.warning(f"Chat ({chat_id.value()}) is already
init")
            return
        self.__chats[chat_id] = Chat(chat_id, Dictionary(set(),
True, True))

        if not self.__clientApi.hasAccessToChatHistory(chat_id):
            self.__log.warning(f"Client hasn't access to chat
({chat_id.value()}) history")
            return

        try:
            chatIter = await self.__clientApi.buildIter(chat_id,
properties.history_limit)
            while True:
                message = await chatIter.next()
                await self.messageProcessing(message, False)
        except StopAsyncIteration:
            pass

    async def __isFromModerator(self, message: Message) -> bool:
        if message.is_from_moderator is None:
            message.is_from_moderator = await
self.__clientApi.isFromModerator(message)
        return message.is_from_moderator
```

```
    @classmethod
    def __chatById(cls, chat_id: Chat.Id):
        if cls.__chats.get(chat_id) is None:
            cls.__chats[chat_id] = Chat(chat_id, Dictionary(set(),
True, True))
        return cls.__chats[chat_id]

    @classmethod
    def __isQuestion(cls, message: Message) -> bool:
        if message.is_question is None:
            message.is_question = False
            sentences = parseOnSentences(message.message)
            for sentence in sentences:
                message.is_question =
cls.__questionDetection.isQuestion(sentence)
                if message.is_question is True:
                    break
        return message.is_question

    @classmethod
    def __questionSimilarity(cls, question1: QuestionSummary,
question2: QuestionSummary):
        return
cls.__messageInterpretationService.similarity(cls.__messageInterpre
tation(question1),

cls.__messageInterpretation(question2),

properties.question_similarity_bert_weight)

    @classmethod
    def __answerChecking(cls, question: QuestionSummary, answer:
Message, weight: float = 0.0):
        similarity =
cls.__messageInterpretationService.similarity(cls.__messageInterpre
tation(question),

cls.__messageInterpretation(answer),

properties.answer_similarity_bert_weight)
        answer_confidence = cls.__answerConfidence(similarity,
weight)
        if similarity >= properties.qa_similarity_threshold and
answer_confidence >= properties.answer_threshold:
            if question.answer_id is None or
```

```
question.answer_confidence < answer_confidence:
                question.answer_id = answer.id
                question.answer_confidence = answer_confidence

    @classmethod
    def __messageInterpretation(cls, message: Message |
QuestionSummary) -> MessageInterpretation:
        chat = cls.__chatById(message.id.chat_id)
        if message.interpretation is None or
message.interpretation.dict_ver != chat.dictionary.getVersion():
            message.interpretation =
cls.__messageInterpretationService.toInterpretation(message.message
,

chat.dictionary)
        return message.interpretation

    @classmethod
    def __updateDictionary(cls, message: Message):
        chat = cls.__chatById(message.id.chat_id)
        words = parseOnWords(message.message)
        for word in words:
            chat.dictionary.increaseWordCurrency(word)
        if (chat.message_count["user"] +
chat.message_count["moderator"]) % 1000 == 0:
            chat.dictionary.cleanCurrency()
            chat.dictionary.update()

    @staticmethod
    def __messageWeight(user_type: str, reply: bool = False) ->
float:
        if reply:
            return properties.user_weight[user_type] *
properties.reply_weight
        else:
            return properties.user_weight[user_type]

    @staticmethod
    def __answerConfidence(similarity: float, weight: float) ->
float:
        return 0 if weight == 0.0 else 1.0 - (1.0 - similarity) *
(1.0 / weight)
```

Код файлу ***Chat.py***:

```
from __future__ import annotations

from collections import deque

from telethon.tl.types import TypePeer, PeerUser, PeerChat,
PeerChannel

from Dictionary import Dictionary


class Chat:
    class Id:
        def __init__(self, peed_id: 'TypePeer') -> None:
            self.peer_id = peed_id

        def __hash__(self):
            match self.peer_id:
                case PeerUser():
                    return hash((0, self.peer_id.user_id))
                case PeerChat():
                    return hash((1, self.peer_id.chat_id))
                case PeerChannel():
                    return hash((2, self.peer_id.channel_id))

        def __eq__(self, other: Chat.Id):
            return self.peer_id == other.peer_id

        def __str__(self):
            return str(self.peer_id)

        def value(self) -> int:
            match self.peer_id:
                case PeerUser():
                    return self.peer_id.user_id
                case PeerChat():
                    return self.peer_id.chat_id
                case PeerChannel():
                    return self.peer_id.channel_id

    def __init__(self, id: Id, dictionary: Dictionary = None) ->
None:
        self.id = id
        self.dictionary = dictionary
        self.message_count = {"user": 0, "moderator": 0}
```

```
        self.questions_queue = {"user": deque(), "moderator":
deque()}
        self.questions = {}
```

Код файлу ***ChatIter.py***:

```
from telethon import TelegramClient, hints

from Message import Message


class ChatIter:
    def __init__(self,
                 client: TelegramClient,
                 entity: 'hints.EntityLike',
                 limit: int,
                 start_id: int = 0,
                 downward: bool = False) -> None:
        self.__iter = aiter(client.iter_messages(entity=entity,
                                                 limit=limit,
                                                 offset_id=start_id,
                                                 reverse=downward))

    async def next(self) -> Message:
        return Message.fromTelethonMessage(await anext(self.__iter))
```

Код файлу ***ClientApi.py***:

```
import logging

from telethon import TelegramClient
from telethon.tl.functions.channels import GetParticipantRequest
from telethon.tl.types import ChannelParticipantAdmin,
ChannelParticipantCreator

from Chat import Chat
from ChatIter import ChatIter
from Message import Message


class ClientApi:
    __log = logging.getLogger('__name__')

    def __init__(self, client: TelegramClient) -> None:
        self.__client = client
        self.__is_bot = None
```

```
    async def isBot(self) -> bool:
        if self.__is_bot is None:
            self.__is_bot = await self.__client.is_bot()
        return self.__is_bot

    async def hasAccessToChatHistory(self, chat_id: Chat.Id) ->
bool:
        return not await self.isBot()

    async def buildIter(self,
                        chat_id: Chat.Id,
                        limit: int,
                        start_message_id: int = 0,
                        downward: bool = False) -> ChatIter | None:
        if not await self.hasAccessToChatHistory(chat_id):
            self.__log.error(f"Client hasn't access to chat
({chat_id.value()})_ history")
            return None
        else:
            return ChatIter(self.__client, chat_id.peer_id, limit,
start_message_id, downward)

    async def isFromModerator(self, message: Message) -> bool:
        participant = await
self.__client(GetParticipantRequest(channel=message.id.chat_id.valu
e(),

participant=message.from_id))
        return isinstance(participant.participant,
ChannelParticipantAdmin | ChannelParticipantCreator)

    @staticmethod
    def buildRespond(answers) -> str:
        respond = "Hello!\n\n" \
                  + "I'll try to help you with you question. This
could be related answer(s):\n"
        for i, answer in enumerate(answers):
            respond += f"{i + 1}) t.me/c/{str(answer[1])} -
confidence: {answer[0]}\n"
        respond += "\n"
        respond += "Do I help you to resolve your question?"
        return respond
```

Код файлу ***Dictionary.py***:

```
from parser import parseOnWords, wordToStem
from properties import properties


class Dictionary:
    def __init__(self,
                 dictionary: set = None,
                 update_enabled: bool = False,
                 configure_default_common_words_usage: bool = False)
-> None:
        self.__dictionary = {} if dictionary is None else
{wordToStem(w): i for i, w in enumerate(dictionary)}
        self.__dictionary_version = 0
        self.__update_enabled = update_enabled
        self.__words_currency = {}
        self.__words_count = 0
        self.__common_words_usage = {}
        if configure_default_common_words_usage:
            self.configCommonWordsUsage()

    def __str__(self):
        return str(self.__dictionary) + "\n" +
str(self.__words_currency) + "\n"

    def saveToFile(self, path):
        with open(path ,'w') as f:
            for word in self.__dictionary:
                f.write(str(word) + "\n")

    def setFromFile(self, path, update_enabled: bool = False):
        dictionary = set(parseOnWords(open(path).read()))
        self.__dictionary = {w.lower(): i for i, w in
enumerate(dictionary)}
        self.__update_enabled = update_enabled
        self.__dictionary_version += 1

    def set(self, dictionary: set, update_enabled: bool = False):
        self.__dictionary = {w.lower(): i for i, w in
enumerate(dictionary)}
        self.__update_enabled = update_enabled
        self.__dictionary_version += 1

    def getVersion(self) -> int:
        return self.__dictionary_version
```

```
    def size(self) -> int:
        return len(self.__dictionary)

    def index(self, word: str) -> bool:
        word = wordToStem(word)
        return self.__dictionary.get(word)

    def setUpdateMode(self, update_enabled: bool):
        self.__update_enabled = update_enabled

    def increaseWordCurrency(self, word: str):
        word = wordToStem(word)
        if self.__update_enabled:
            if self.__words_currency.get(word) is None:
                self.__words_currency[word] = 0
            self.__words_currency[word] += 1
            self.__words_count += 1

    def cleanCurrency(self, threshold: int =
properties.dictionary.default_clean_threshold):
        if self.__update_enabled:
            for word in list(self.__words_currency):
                if self.__words_currency[word] <= threshold:
                    self.__words_count -=
self.__words_currency[word]
                    self.__words_currency.pop(word)

    def update(self,
               max_size: int = properties.dictionary.default_size,
               common_words_weight: float =
properties.dictionary.default_common_words_weight,
               threshold: float =
properties.dictionary.default_update_threshold):
        if self.__update_enabled:
            if len(self.__common_words_usage) != 0:
                words_usage = []
                for word in self.__words_currency:
                    common_words_usage =
self.__common_words_usage[word] \
                        if self.__common_words_usage.get(word) is
not None \
                        else 0
                    word_usage = self.__words_currency[word] /
self.__words_count \
```

```
                                    - common_words_usage *
common_words_weight
                    if word_usage > threshold:
                        words_usage.append((word_usage, word))
                if len(words_usage) > 0:
                    words_usage.sort(reverse=True)
                    self.__dictionary = {w[1]: i for i, w in
enumerate(words_usage[:max_size])}
                    self.__dictionary_version += 1

    def configCommonWordsUsage(self, path="long-text.txt"):
        self.__dictionary_version += 1
        self.__common_words_usage = {}

        words = parseOnWords(open(path, "r",
encoding="utf-8").read())
        for word in words:
            word = wordToStem(word)
            if self.__common_words_usage.get(word) is None:
                self.__common_words_usage[word] = 0
            self.__common_words_usage[word] += 1

        for word in self.__common_words_usage:
            self.__common_words_usage[word] /= len(words)
```

Код файлу ***main.py***:

```
from telethon import TelegramClient, events
from telethon.tl.types import PeerChannel, ReplyInlineMarkup,
KeyboardButtonRow, KeyboardButtonCallback

from BotCore import BotCore
from Chat import Chat
from ClientApi import ClientApi
from Message import Message
from TelegramConfig import *

client = TelegramClient("not bot", api_id, api_hash).start()
bot = TelegramClient("bot", api_id,
api_hash).start(bot_token=api_token)

botCore = BotCore(ClientApi(client))


@bot.on(events.NewMessage(pattern="/init"))
async def init(event):
```

```
    chat_id = Chat.Id(event.peer_id)
    await botCore.initChat(chat_id)


@bot.on(events.NewMessage())
async def message_processing(event):
    if isinstance(event.peer_id, PeerChannel):
        if event.message.message.startswith("/init"):
            return

        message = Message.fromTelethonMessage(event.message)
        response = await botCore.messageProcessing(message)
        if response is not None:
            inline_buttons = ReplyInlineMarkup(
                [
                    KeyboardButtonRow(
                        [
                            KeyboardButtonCallback(
                                text="Yes, thanks!",
                                data=b'Yes'
                            ),
                            KeyboardButtonCallback(
                                text="No. Please call a moderator",
                                data=b'No'
                            )
                        ]
                    )
                ]
            )
            await event.respond(message=response,
reply_to=event.message.id, buttons=inline_buttons)


@bot.on(events.CallbackQuery(data=b'Yes'))
async def yes(event):
    await event.respond(message="You're welcome!",
reply_to=event.message_id)
    tMessage = await event.get_message()
    message = Message.fromTelethonMessage(tMessage)
    await botCore.acceptGeneratedAnswer(message.id,
message.reply_id)


@bot.on(events.CallbackQuery(data=b'No'))
async def no(event):
```

```
    tMessage = await event.get_message()
    await event.respond(message="MODERATOR!!!",
reply_to=tMessage.reply_to.reply_to_msg_id)


def main():
    bot.run_until_disconnected()


if __name__ == "__main__":
    main()
```

Код файлу ***Message.py***:

```
from __future__ import annotations

import telethon
from telethon.tl.types import TypePeer

from Chat import Chat
from MessageInterpretation import MessageInterpretation


class Message:
    class Id:
        def __init__(self, chat_id: Chat.Id, message_id: int) ->
None:
            self.chat_id = chat_id
            self.message_id = message_id

        def __hash__(self):
            return hash((self.chat_id, self.message_id))

        def __eq__(self, other: Message.Id):
            return self.chat_id == other.chat_id and self.message_id
== other.message_id

        def __str__(self):
            return f"{self.chat_id.value()}/{self.message_id}"

    def __init__(self,
                 id: Id,
                 message: str,
                 from_id: 'TypePeer',
                 reply_id: Id = None,
                 is_from_moderator: bool = None,
```

```
                is_question: bool = None,
                interpretation: MessageInterpretation = None) ->
None:
        self.id = id
        self.message = message
        self.from_id = from_id
        self.reply_id = reply_id
        self.is_from_moderator = is_from_moderator
        self.is_question = is_question
        self.interpretation = interpretation

    @classmethod
    def fromTelethonMessage(cls, tMessage:
telethon.tl.custom.message.Message):
        reply_id = None \
            if tMessage.reply_to is None \
            else Message.Id(Chat.Id(tMessage.peer_id),
tMessage.reply_to.reply_to_msg_id)
        return Message(Message.Id(Chat.Id(tMessage.peer_id),
tMessage.id), tMessage.message, tMessage.from_id, reply_id)
```

Код файлу ***MessageInterpretation.py***:

```
class MessageInterpretation:
    def __init__(self, bert_vec, dict_vec, dict_ver) -> None:
        self.bert_vec = bert_vec
        self.dict_vec = dict_vec
        self.dict_ver = dict_ver
```

Код файлу ***MessageInterpretationService.py***:

```
import math

from BertWrapper import BertWrapper
from Dictionary import Dictionary

import string
import torch

from MessageInterpretation import MessageInterpretation
from parser import parseOnWords


class MessageInterpretationService:
    def __init__(self, device, bert: BertWrapper) -> None:
        self.__cos = torch.nn.CosineSimilarity(dim=1, eps=1e-6)
```

```
        self.__device = device
        self.__bert = bert

    def similarity(self,
                   interp1: MessageInterpretation,
                   interp2: MessageInterpretation,
                   bert_multiplier: float) -> float:
        bert_cos = self.__bertCosSim(interp1.bert_vec,
interp2.bert_vec)
        if interp1.dict_vec is None and interp2.dict_vec is None:
            return bert_cos
        else:
            dict_cos = self.__dictCosSim(interp1.dict_vec,
interp2.dict_vec)
            return (bert_multiplier * bert_cos + (1 -
bert_multiplier) * dict_cos).item()

    def toInterpretation(self, message: str, dictionary: Dictionary)
-> MessageInterpretation:
        bert_vec = self.__bert([message])[1]
        dict_vec = self.__dictionaryUse(message, dictionary)

        return MessageInterpretation(bert_vec, dict_vec,
dictionary.getVersion())

    def __dictionaryUse(self, msg: str, dictionary: Dictionary):
        words = self.__parseMsgOnWords(msg)
        if dictionary.size() == 0:
            dict_vec = None
        else:
            dict_vec = []
            for word in words:
                index = dictionary.index(word)
                if index is not None:
                    dict_vec.append(index)
        return dict_vec

    def __bertCosSim(self, vec1, vec2):
        return self.__cos(vec1, vec2)

    def __dictCosSim(self, dict1, dict2):
        if len(dict1) == 0 or len(dict2) == 0:
            return 0.5 if len(dict1) == 0 and len(dict2) == 0 else
0.0
        return len(set(dict1).intersection(dict2)) /
```

```
math.sqrt(len(dict1) * len(dict2))

    @staticmethod
    def __parseMsgOnWords(msg: str):
        return set(parseOnWords(msg))
```

Код файлу ***parser.py***:

```
import string
import nltk

nltk.download('punkt')
stemmer = nltk.PorterStemmer()


def wordToStem(word: str) -> str:
    if len(word) >= 4 and word[-3] == "'":
        word = word[:-3]
    elif len(word) >= 3 and word[-2] == "'":
        word = word[:-2]
    else:
        word = word
    return stemmer.stem(word)


def __isLink(word: str) -> bool:
    word = word.strip(string.punctuation)
    return word.startswith("https://") or word.startswith("http://")


def parseOnWords(text: str) -> list:
    return [wordToStem(word.strip(string.punctuation))
            for word_ in text.split()
            if not __isLink(word_)
            for word in word_.split("/")
            if len(wordToStem(word.strip(string.punctuation))) > 0]


def parseOnSentences(text: str) -> list:
    return nltk.tokenize.sent_tokenize(text)
```

Код файлу ***properties.py***:

```
class properties:
    class bert:
        version = "bert-base-uncased"
        out_size = 768
        token_max_seq_len = 25

    dropout_prob = 0.1
    learning_rate = 0.1
    epochs = 10

    class dictionary:
        default_size = 100
        default_clean_threshold = 5
        default_update_threshold = 0.0
        default_common_words_weight = 10

    max_answers = {"user": 50, "moderator": 10}
    max_replies = 10

    user_weight = {"user": 0.5, "moderator": 1.0}
    reply_weight = 2.0

    question_mark_weight = 0.5

    question_similarity_bert_weight = 0.7
    question_similarity_threshold = 0.8

    answer_similarity_bert_weight = 0.5
    qa_similarity_threshold = 0.7
    answer_threshold = 0.8

    top_answers_max_size = 5

    history_limit = 10000
```

Код файлу ***QuestionDetDataset.py***:

```
import json
import random
from torch.utils.data import Dataset

class QuestionDetDataset:
    class Data(Dataset):
        def __init__(self, sentences) -> None:
```

```
            self.sentences = sentences

        def __len__(self):
            return len(self.sentences)

        def __getitem__(self, index):
            return self.sentences[index]

    def getDataset(name) -> Data:
        match name:
            case "SQuAD":
                return
QuestionDetDataset.__SQuAD('train-v2.0.json'),
QuestionDetDataset.__SQuAD('dev-v2.0.json')
            case _:
                print("No dataset selected")
                return None

    def buildDataset(sentences):
        return QuestionDetDataset.Data(sentences)

    def __SQuAD(path):
        with open(path, 'r') as file:
            data = json.load(file)

            questions = []
            answers = []
            for block in data["data"]:
                for paragraph in block["paragraphs"]:
                    for qa in paragraph["qas"]:
                        questions.append([qa["question"], 1])
                        if qa["is_impossible"] == False:

answers.append([qa["answers"][0]["text"], 0])

            sentences = questions + answers
            random.shuffle(sentences)

            return QuestionDetDataset.Data(sentences)
```

Код файлу ***QuestionDetection.py***:

```
from properties import properties
from BertWrapper import BertWrapper

import pathlib
import datetime
import torch
import torch.nn as nn


class QuestionDetection():

   class __Module(nn.Module):

       class __ModuleEnd(nn.Module):
           def __init__(self) -> None:
               super().__init__()

               self.__drop = nn.Dropout(properties.dropout_prob)
               self.__ln = nn.Linear(properties.bert.out_size, 2)
               self.__softmax = nn.Softmax(dim=1)

           def forward(self, x):
               x = self.__drop(x)
               x = self.__ln(x)
               x = self.__softmax(x)

               return x

       def __init__(self, bert: BertWrapper, path = None) -> None:
           super().__init__()

           self.__bert = bert
           self.__moduleEnd = self.__ModuleEnd()
           if (path != None):
               self.load(path)

       def forward(self, sentences):
           x = self.__bert(sentences, max_padding=True)[1]
           x = self.__moduleEnd(x)

           return x

       def save(self, path):
           torch.save(self.__moduleEnd.state_dict(), path)
```

```
        def load(self, path):
            self.__moduleEnd.load_state_dict(torch.load(path))

    def __init__(self, device, bert: BertWrapper, path = None) ->
None:
        self.__device = device
        self.__module = self.__Module(bert, path).to(device)

    def isQuestion(self, sentence) -> bool:
        self.__module.eval()
        has_question_mark = 1.0 if "?" in sentence else -1.0
        question_prob = self.__module([sentence])[0]
        return question_prob[0].item() <= question_prob[1].item() +
has_question_mark * properties.question_mark_weight

    def train(self, train_dataset, valid_dataset, train_batch_size,
valid_batch_size, criterion, learning_rate, epochs):
        self.__module.train()

        train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=train_batch_size)
        valid_loader =
torch.utils.data.DataLoader(dataset=valid_dataset,
batch_size=valid_batch_size)
        optimizer = torch.optim.Adam(self.__module.parameters(), lr
= learning_rate)

        correctness_rate = []
        for epoch in range(epochs):
            for x, y in train_loader:
                optimizer.zero_grad()

                _y = torch.zeros(len(y), 2).to(self.__device)
                for i in range(len(y)):
                    _y[i][y[i]] = 1

                yhat = self.__module(x)
                loss = criterion(_y, yhat)
                loss.backward()
                optimizer.step()
            print("Trained: ", 100 * (epoch + 1) / epochs, "%",
end="\r")
```

```
            correct = 0
            for x, y in valid_loader:
                _, yhat = torch.max(self.__module(x), 1)
                yhat.to(self.__device)
                y = torch.tensor(y).to(self.__device)
                correct += (y == yhat).sum().item()
            correctness_rate.append(100 * (correct /
len(valid_dataset)))
            print("Validated: ", 100 * (epoch + 1) / epochs, "%",
end="\r")

        self.__module.eval()
        self.saveModuleToFile()
        return correctness_rate

    def saveModuleToFile(self, path = "default", filename =
"default"):
        save_to_last_version = (path == "default" and  filename ==
"default")

        if (path == "default"):
            path = "modules/QuestionDetection/versions/"
        pathlib.Path(path).mkdir(parents=True, exist_ok=True)

        if (filename == "default"):
            dt = datetime.datetime.now()
            filename = dt.strftime("%y-%m-%d-%H-%M-%S") + ".pt"

        self.__module.save(path + filename)
        if (save_to_last_version):
            self.__module.save("modules/QuestionDetection/last.pt")

    def loadModuleFromFile(self, path = "default"):
        if (path == "default"):
            path = "modules/QuestionDetection/last.pt"
        self.__module.load(path)
```

Код файлу ***QuestionSummary.py***:

```
from __future__ import annotations

from Message import Message
from MessageInterpretation import MessageInterpretation
from properties import properties


class QuestionSummary:
   def __init__(self,
                id: Message.Id,
                message: str,
                user_messages_before_question: int,
                moderator_messages_before_question: int,
                replies: int = 0,
                answer_id: Message.Id = None,
                answer_confidence: float = None,
                interpretation: MessageInterpretation = None) ->
None:
       self.id = id
       self.message = message
       self.messages_before_question = {"user":
user_messages_before_question,
                                        "moderator":
moderator_messages_before_question}
       self.replies = replies
       self.answer_id = answer_id
       self.answer_confidence = answer_confidence
       self.interpretation = interpretation

   def waitingForAnswer(self, messages: int, user_type: "str") ->
bool:
       return messages - self.messages_before_question[user_type]
<= properties.max_answers[user_type]

   def waitingForReply(self) -> bool:
       return self.replies <= properties.max_replies
```