МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ ОБ'ЄКТІВ ЛІНГВІСТИЧНОЇ ПРЕДМЕТНОЇ ГАЛУЗІ

(3-Й КУРС)

МАТЕРІАЛИ ТА РЕКОМЕНДАЦІЇ

до виконання лабораторних робіт

Автор:

Микола КОСТІКОВ,
к.т.н., доц.,
доцент каф. інформаційних
систем та технологій

КИЇВ КНУТШ 2026

ЗМІСТ

ЛАБОРАТОРНА РОБОТА 1.

Базові завдання автоматичного опрацювання тексту

Сучасні технології комп'ютерної лінгвістики, зокрема для різноманітного опрацювання текстів, передбачають використання мов програмування високого рівня, а також відповідних бібліотек і модулів для них. Необхідні засоби розроблено для багатьох об'єктно-орієнтованих і функціональних мов програмування, серед яких Python, C# та інші. Тим не менше, в останні роки одним із лідерів у галузях опрацювання природної мови та роботи зі штучним інтелектом лишається саме ***Python*** [6]. Для цієї мови створено цілий ряд модулів, які широко використовуються для розв'язання відповідних завдань.

Що стосується середовищ розроблення (IDE), для Python доступні різні засоби, серед яких PyCharm, PyDev, NetBeans, Visual Studio Code, IDLE тощо. В цьому курсі приклади роботи реалізовано через середовище PyCharm, розроблене компанією JetBrains. Воно має як повноцінну професійну (Professional), так і безкоштовну (Community) версії. Також є можливість використання академічних ліцензій для здобувачів при вивченні програмування та роботі над навчальними проєктами.

Для роботи з консоллю в Python використовують 2 стандартні функції — **print()** та **input().** Перша виводить на екран довільний текст або дані інших типів, у тому числі списки, об'єкти тощо. Якщо хочемо вивести рядок, його слід узяти в одинарні або подвійні лапки. Для виведення інших типів даних іноді слід явним чином конвертувати їх у рядок через функцію **str().** Можна також виводити будь-які змінні, їх пишуть у **print()** без лапок:

```
print('Hello, world!')
print("Another line of text")
print(str(date))
a = 1
print(a)
```

Декілька значень, навіть різних типів, можна записати в **print()** через кому, тоді вони виведуться в консоль через пробіл:

```
print('Hello', a)
# результат: Hello 1
```

Також можна використати операцію *конкатенації* — з'єднання, склеювання двох і більше рядків у один через знак **«+».** При цьому слід стежити, аби всі складові обов'язково мали рядковий тип даних. Натомість рядки в одинарних і подвійних лапках можуть поєднуватись без проблем. Наступні 4 рядки коду однаково виведуть те саме слово «*hello*»:

```
print('hello')
print("hello")
```

```
print("hell" + "o")
print("hell" + 'o')
```

Крім того, є можливість у одному операторі **print()** здійснити перенесення рядка або вставити табуляцію (відступ, зазвичай довжиною у 8 пробілів) через використання спеціальних символів — «**\n**» (від англ. «*new line*» — новий рядок) та «**\t**» («*tabulation*»):

```
print('a\tb')
print("c\nd")
```

Результат виглядатиме так:

```
a    b
c
d
```

Якщо ж нам слід вивести послідовність «**\n**», «**\t**» чи інші подібні як є, щоби вони не сприймалися інтерпретатором Python як спеціальні символи, можна скористатися дослівним виведенням рядка через префікс «**r**» (від англ. «*raw*» — «сирий», «необроблений»), який ставиться перед лапками:

```
print(r'w\tf')
# результат: w\tf
```

Введення можливе через функцію **input().** При цьому її можна використати або без параметрів (тоді вона просто чекатиме на введення користувача), або з параметром рядкового типу — конкретним запитом до користувача, наприклад:

```
f = input()
h = input('Введіть значення h:')
```

Важливо, що перед функцією **input()** слід писати назву змінної, якій хочемо присвоїти результат введення користувача, інакше воно не збережеться.

Для опрацювання рядків Python має багато різних функцій. Це зокрема такі:

- **.find()** — знайти індекс першого входження підрядка;
- **.count()** — порахувати кількість входжень підрядка;
- **.replace()** — замінити один символ / підрядок на інший;
- **.split()** — розбити рядок на список підрядків за пробілами;
    - **.split('_')** — розбити за символом-розділювачем (тут — «_»);
- **.strip()** — видалити всі пробіли та відступи на початку та в кінці;
- **.lower()** — зробити всі літери малими;
- **.upper()** — зробити всі літери великими;
- **.startswith()** — чи рядок починається з символа / підрядка в дужках;
- **.endswith()** — чи рядок закінчується на символ / підрядок у дужках;
- **[m]** — взяти символ рядка за індексом **m** (починаючи з 0);
- **[-1]** — останній символ рядка (аналогічно інші від’ємні індекси);

- **[m:n]** — взяти символи з індексу **m** (включно) по **n** (не включно);
- **[m:]** — взяти символи з індексу **m** до кінця;
- **[:n]** — взяти символи з початку до індексу **n**;

Усі вищенаведені функції пишуться після назви рядка, який опрацьовуємо. Крім того, є функція **join(),** яка застосовується до списку рядків та з'єднує його в один суцільний, використовуючи довільний розділювач, наприклад:

```
text = ['a', 'b', 'c']
print('_'.join(text))
# результат: a_b_c
```

Тут слід також звернути увагу на дві типові помилки в програмуванні, яких іноді припускаються через незнання або неуважність, особливо на початку вивчення функцій роботи з рядками та іншими даними.

По-перше, аби наочно побачити результат будь-якого опрацювання рядків, його слід вивести на екран через **print()** або іншим чином. Інакше опрацювання рядка відбудеться в пам'яті комп'ютера, але його результат не буде видимим.

По-друге, якщо не присвоїти результат опрацювання рядковій змінній (новій або тій самій), він не збережеться, і на наступних кроках програми буде показуватись і опрацьовуватись той самий початковий рядок.

Для типових завдань, які використовуватимуться у програмі неодноразово, зручно створювати власні функції. В Python це робиться наступним чином:

```
def change_a_to_b(text):
    new_text = text.replace('a', 'b')
    return new_text
```

У прикладі вище функція **change_a_to_b()** бере на вхід змінну **text,** яка повинна мати рядковий тип, і опрацьовує її, заміняючи літеру «**a**» на «**b**» та зберігаючи результат у новій змінній **new_text.** Після цього новостворена змінна з результатом заміни повертається функцією назовні.

Тепер можемо викликати цю функцію та побачити її результат у консолі:

```
line = 'language'
print(change_a_to_b(line))
# результат: lbngubge
```

Дуже важливим моментом при роботі з рядковими даними та текстами у програмуванні є кодування символів. Якщо його налаштовано неправильно, то для багатьох мов, включно з українською, можливе некоректне виведення тексту або окремих літер. Наприклад, замість слова «*Привіт*» можемо побачити таке:

```
Прив?т
Прив_т
??????
_
```

Річ у тому, що базовим набором символів, який підтримується всюди ще з 1960-х років, є стандартна латинська абетка (26 літер), яка нині використовується в англійській мові. Будь-які інші символи латинської абетки (в тому числі спеціальні німецькі, французькі, іспанські та інші літери) є додатковими відносно цього основного набору і потребують особливого кодування для коректного збереження та відображення в електронних системах. Цікаво, що буквально всі сучасні абетки європейських мов, містять принаймні декілька літер, що не входять до базової латинки. Це модифікації стандартних літер (діакритика: **ä, ö, ü, å** тощо), наголоси над ними (наприклад, у італійській: **à, ú** і т.д.) та окремі літери, яких узагалі немає у звичайній латинській абетці (ісландська мова: **ð, þ, æ**).

З кирилицею, грецькою та іншими абетками все ще складніше, адже в них набір символів повністю відрізняється від латиниці. При цьому в кирилиці теж є так звана базова абетка (на основі російської та болгарської). Тож літери інших мов (українська, білоруська, сербська, македонська, монгольська тощо), яких немає в основному наборі символів, теж можуть спричиняти труднощі при роботі з текстом. Для української це літери **ґ, є, і, ї,** а також іноді апостроф.

Задля розв'язання проблем із кодуванням почали створювати нові стандарти Windows та ISO, які охоплювали групи мов із подібними абетками. Одне з нових кодувань дозволяло працювати відразу з декількома центральноєвропейськими мовами, інше — зі скандинавськими мовами тощо. Проте якщо певний проєкт вимагав підтримки відразу і польської, і шведської мови (тобто з різних груп), то працювало в підсумку лише щось одне. З плином часу, глобалізацією світу, появою та поширенням інтернету ця проблема ставала все більш актуальною.

Подолати ці труднощі був покликаний стандарт Unicode, який почали розробляти в 1988 році. Кодування цієї групи стали охоплювати літери відразу багатьох абеток. Це цілий набір кодувань, серед яких UTF-7, UTF-8, UTF-16, UTF-32 та їхні варіанти. Між собою вони відрізняються кількістю байтів, що виділяються на кодування одного символа, послідовністю знаків у таблиці символів, а також правилами кодування.

Найкращим варіантом зараз є UTF-8, який підтримує не лише розширену латиницю та кирилицю, а й інші абетки, включно з ієрогліфами для східних мов, а також сучасні емоджі. Тож якщо проєкт передбачає роботу відразу з декількома різними мовами, що мають відмінні абетки, саме цей стандарт дозволить уникнути труднощів.

Щодо сумісності з UTF-8 у сучасних технологіях, то нині Python, C# та багато інших мов програмування вже відразу за замовчуванням підтримують це кодування в повному обсязі. Таким чином, їх можна без проблем використовувати для опрацювання текстових даних безліччю мов світу.

Важливо, що при програмуванні слід розрізняти 2 кодування — введення та виведення (*input / output encoding*). Якщо налаштувати лише одне з них, то з іншим усе одно можливі проблеми. Наприклад, текст правильно виводитиметься на екран, однак буде некоректно записаний до текстового файлу, бази даних тощо, і навпаки.

Для перевірки кодування зручно використовувати *панграми* — текст, який містить відразу всі літери абетки принаймні по одному разу. Панграми також

застосовують для демонстрації друкованого та рукописного вигляду літер певної абетки, для перевірки та вибору шрифтів у дизайні друкованих та електронних видань, для перевірки коректності передачі даних певною мовою тощо.

Існує багато панграм для природних мов. Однією з найпоширеніших для англійської є фраза «*The quick brown fox jumps over a lazy dog*». Саме це речення зазвичай використовують для демонстрації накреслення шрифтів латинкою.

Для української є декілька відомих варіантів, серед яких: *«Гей, хлопці, не вспію — на ґанку ваша файна їжа знищується бурундучком», «В Бахчисараї фельд'єґер зумів одягнути ящірці жовтий капюшон!», «Жебракують філософи при ґанку церкви в Гадячі, ще й шатро їхнє п'яне знаємо».* Як бачимо, останні дві панграми містять не лише всі літери української абетки, а й апостроф.

Транслітерація використовується для передачі літер однієї мови літерами іншої на письмі. Транскрипцію натомість застосовують для передачі звучання однієї мови засобами іншої. Для транслітерації зазвичай використовують абетку іншої природної мови, а для транскрипції також існують додаткові засоби — наприклад, міжнародний фонетичний алфавіт (IPA).

І транслітерацію, і транскрипцію застосовують:

- для запису імен та прізвищ іншомовного походження;
- при перекладі географічних назв, вказівників, мап тощо;
- для запису назв компаній, торгових марок, зокрема в рекламі;
- для навчання іноземних мов (особливо на початковому етапі);
- у мовах із двома абетками (наприклад, сербська);
- при проблемах із кодуванням абетки — наприклад, кирилиці:
  - у старій чи іноземній техніці (касові апарати, табло);
  - у веб-системах (назви сайтів та адреси електронної пошти);
  - у мобільному зв'язку (SMS латинкою можуть бути довшими);
  - у програмуванні (назви змінних, функцій).

Цікаво, що коли йдеться про географічні назви, то крім транслітерації та транскрипції також є підхід із використанням перекладу змісту з іншої мови без передачі написання чи звучання оригіналу. Існують різні погляди за та проти перекладу. З одного боку, переклад набагато ясніший для людини, яка не володіє мовою — наприклад, англомовний турист зрозуміє назву зупинки «*Railway Station*» значно краще, ніж «*Vokzalna*». З іншого боку, транслітерація «*Vokzalna*» та звукові оголошення в транспорті саме такої назви полегшать спілкування з місцевими мешканцями, які звикли до української назви «*Вокзальна*» і не завжди володіють англійською, аби зі свого боку зрозуміти варіант «*Railway Station*». У результаті в київському метрополітені зараз використовується транслітерація, а в наземному транспорті — переклад. Також на різних мапах можемо побачити як назву «*North Bridge*», так і «*Pivnichnyi Bridge*».

Важливою особливістю і транслітерації, і транскрипції є те, що вони не завжди підлягають зворотному відтворенню. Наприклад, у мові фарсі, грузинській та інших є досить подібні між собою звуки, які важко розрізнити в українській чи англійській. І в наших абетках не вистачає окремих літер, аби передати їх усі по-різному. В таких випадках дві чи більше різних літер із оригіналу доводиться

передавати тією самою літерою іншої мови. Тож при зворотному відтворенні може виникнути питання, яка саме літера насправді була в початковому тексті.

Для транслітерації традиційно існують державні та інші стандарти. Щодо запису українських літер латиницею, ще в 1971 році з'явився стандарт ГОСТ, який дозволяв передавати українські імена, назви та інше. Проте він мав ряд недоліків. По-перше, була відсутня літера «**ґ**», а літера «**г**» за аналогією з російською передавалась як «**g**», що не відповідає її звучанню в українській мові. По-друге, для деяких літер використовувалась діакритика (**ž, š** і навіть **ŝ**), що вимагає наявності спеціальної клавіатури чи додаткової розкладки, а також може бути не зрозумілим для іншомовних людей. Крім того, така транслітерація не знімає проблем із кодуванням, які було викладено вище.

У підсумку після багатьох спроб удосконалити ці правила в 2010 році було створено та затверджено новий стандарт, який початково використовувався для офіційного запису імен і прізвищ громадян у закордонних паспортах, але відтоді фактично застосовується і для багатьох інших цілей. Він теж має свої недоліки, проте за рахунок відсутності додаткових літер (використовується лише базова латиниця, як і в англійській) немає проблем із кодуванням, а також із розумінням звучання літер — принаймні, для людей, що володіють англійською. Особливістю цього стандарту є також те, що м'який знак і апостроф не відтворюються взагалі, а для буквосполучення «**зг**» у правилах транслітерації є виняток: воно передається як «**zgh**», аби уникнути збігу з «**zh**», що передає українську літеру «**ж**».

Слід також сказати, що згідно з Цивільним кодексом України, громадяни мають право на відмінності від офіційного стандарту транслітерації при написанні своїх імен та прізвищ, зокрема через національні традиції.

Крім того, до 2010 року в Україні вже було досить багато офіційних документів, наукових публікацій тощо, де фігурувала інша транслітерація, за старими стандартами. З цієї та інших причин деякі люди залишили попередній варіант написання своїх імен та прізвищ, тож навіть зараз сучасний стандарт транслітерації використовується не у 100% випадків.

Також варто зазначити, що при перекладі офіційних документів не англійською, а іншими мовами (польською, німецькою, іспанською тощо) часом використовуються окремі правила транслітерації, що враховують особливості саме цих мов. Ці правила можуть бути затвердженими як стандарти на різних рівнях і можуть мати різні сфери застосування. Такі стандарти спрощують читання та розуміння імен, прізвищ і назв однією мовою носіями іншої. Їх можна знайти для різних пар мов у світі.

Література

Див. джерела №№ 1, 2, 3, 6.

ЛАБОРАТОРНА РОБОТА 2.

Аналіз подібности слів і тексту

Одним із найбільш поширених інструментів мови Python для комп'ютерної лінгвістики та опрацювання природної мови є NLTK [1; 7].

NLTK [7] (від англ. *Natural Language Toolkit* — набір інструментів для природної мови) є набором бібліотек для Python, призначених для опрацювання природної мови, роботи з текстами, корпусами та іншими лексичними ресурсами, зокрема словниками тощо. Вона розробляється з 2001 року і за цей час увібрала в себе величезну кількість корисних та ефективних інструментів.

Серед розробників NLTK — такі фахівці, як Steven Bird, Ewan Klein та Edward Loper. У 2009 р. вони видали підручник із використання цієї бібліотеки — «*Natural Language Processing with Python*», який відтоді став класичним виданням щодо опрацювання природної мови в цілому. Книга містить масу наочних прикладів щодо застосування окремих методів, наявних у NLTK. Безкоштовна вебверсія цього видання з теорією та зразками коду доступна в інтернеті на офіційному сайті самої бібліотеки [1].

Аби встановити та використовувати NLTK у середовищі PyCharm, можна натиснути на пункт «*Configure Python Interpreter*» біля коліщатка налаштувань у вікні коду (праворуч згори або знизу). Далі слід обрати зі списку «*Interpreter Settings...*», і у вікні, що відкриється, клацнути на іконку «+» або натиснути Ctrl+N для завантаження нових модулів. Ввівши у полі пошуку NLTK, можна побачити цей модуль у списку, за необхідності праворуч обрати необхідну версію та встановити її, клацнувши на кнопку «*Install*». Якщо завантаження буде успішним, з'явиться зелена підсвітка, і після цього бібліотеку можна використовувати в коді.

Для посилання на NLTK слід виконати імпорт:

```
import nltk
```

Після цього можна застосовувати в коді майже всі засоби бібліотеки. Проте якщо передбачається робота з корпусами текстів чи іншими ресурсами, їх слід встановити окремо. Річ у тому, що вони не включені до комплектації самої NLTK через завеликі обсяги даних. Отже, завантажити необхідні саме Вам компоненти слід вручну, викликавши в коді наступну функцію:

```
nltk.download()
```

Після запуску такої програми має автоматично відкритись вікно «*NLTK Downloader*». У ньому через графічний інтерфейс можна обрати потрібну вкладку (збірки; корпуси; моделі; все разом) і рядки з окремими компонентами, після чого натиснути кнопку «*Download*» і встановити їх собі на комп'ютер.

Зверніть увагу, що на цьому етапі в цьому ж вікні також можна відразу обрати диск і папку, куди саме будуть завантажуватись усі ці ресурси. За замовчуванням це папка з назвою «*nltk_data*», яка створюється в корені диска C:\,

D:\ або іншого доступного для програми. При виборі розташування цієї папки майте на увазі, що деякі з корпусів є досить об'ємними, а якщо встановити все разом, то в сумі вони можуть зайняти декілька гігабайт. Тож переконайтеся, що у відповідному місці є необхідний обсяг вільного простору.

Після використання в коді команди **nltk.download()** не забудьте закоментувати або видалити її, якщо при подальших запусках програми не плануєте довантажувати інші ресурси.

Для перевірки, чи все встановилось правильно, можна провести такий тест:

```
from nltk.corpus import gutenberg

words = gutenberg.words('chesterton-thursday.txt')
print(len(words), 'words found')
```

Ця програма повинна видати в консоль результат на кшталт «*69213 words found*» — за умови, що попередньо було завантажено корпус Gutenberg із вкладки «*Corpora*». Вищенаведений код звертається до текстового файлу, який містить текст книги Честертона «Людина, що була Четвергом», у зазначеному корпусі, підраховує кількість слів у ньому та виводить отримане число на екран.

Отже, якщо все це пройшло успішно, надалі можна аналогічно посилатись у коді на інші корпуси та ресурси NLTK. За назвою ресурсу інтерпретатор автоматично буде знаходити розташування потрібних папок і файлів на комп'ютері, викликати та використовувати їх при виконанні програми.

Семантичні мережі, або ***Semantic Web*** (дослівно — «семантична павутина») є технологією, яка являє собою надбудову над звичайним World Wide Web, тобто інтернетом як всесвітньою мережею обміну даними.

Якщо звернутись до історії розвитку IT та мереж, то спершу було створено апаратні та програмні засоби для взаємодії комп'ютерів між собою на локальному рівні. Після цього мережі поступово розростались і згодом стали охоплювати всю земну кулю. Розширилися сфери їх застосування та зросла кількість користувачів. З'явилися вебсайти, браузери та пошукові машини.

Однак пошук інформації в інтернеті лишався досить складним завданням доти, доки це відбувалося шляхом використання класичних методів пошуку даних у текстах, а саме за повною збіжністю символьних рядків. Справді, коли стоїть загальне завдання знайти інформацію за певною темою, то доволі важко передбачити, які саме конкретні слова та формулювання буде використано в документах, що відповідають заданій тематиці.

Якщо проводити пошук за одним ключовим словом, завдання спрощується. Проте навіть одне слово може зустрічатись у текстах у різних відмінках чи інших граматичних формах, що ускладнює процес його знаходження. Крім того, існують різні варіанти написання, скорочені форми та абревіатури, а також цілі синонімічні ряди для того самого поняття. В таких випадках дослівний, буквальний пошук видаватиме лише обмежений набір результатів порівняно із зазвичай значно більшим обсягом документів, які насправді можуть бути релевантними, тобто відповідними по суті до поставленої мети пошуку.

Аби розв'язати ці проблеми, з'явилися семантичні мережі, які доповнюють дані додатковими — метаданими, що описують наявні. Головною тут є семантика, тобто зміст, значення слів. Якщо розмітити в текстах усі слова та їхні форми саме їхніми значеннями, стає можливим пошук не за тотожністю рядків, а за суттю того, про що йдеться в цих текстах. Таким чином полегшується машинне опрацювання даних і з'являється змога видавати у відповідь на пошук не лише повні збіжності по тексту, а й усі можливі форми слів, варіації, синоніми тощо.

Однією з ключових технологій семантичних мереж є словник WordNet. Це лексична база даних, яка містить інформацію про семантику слів. Цей словник було розроблено саме для англійської мови, проте пізніше з'явилися спроби реалізувати щось подібне і для інших природних мов. Такі спроби мали різний успіх. Зокрема деякі дослідження щодо створення WordNet для української були започатковані в 2009 році в університеті «Львівська політехніка».

Що стосується використання оригінального WordNet для англійської, він за потреби завантажується аналогічно до інших корпусів у NLTK. Після цього можемо імпортувати та використовувати в коді всі ресурси та функції цього словника. Між іншим, одним зі зручних способів посилання на модулі в Python є створення коротких псевдонімів, як у наступному прикладі:

```
from nltk.corpus import wordnet as wn
```

Тож надалі в коді посилатимемось на WordNet саме за коротким іменем **wn.**

Словник WordNet зберігає значення слів і відношення між ними. При цьому слід пам'ятати, що слова не є тотожними їхнім семантичним значенням. І це є ключовою особливістю WordNet і подібних засобів.

Дійсно, одне й те саме слово може мати більш ніж десяток різних значень і їхніх відтінків. У цьому можна переконатися, відкривши будь-який тлумачний словник. Це стосується багатьох природних мов, зокрема й української. Скажімо, слово «коса» може означати як зачіску, так і знаряддя праці чи піщаний півострів. Такі слова називають *омонімами*, і подібних прикладів безліч.

В англійській, із якою працює WordNet, ця характеристика проявляється ще більш яскраво. Адже те саме англійське слово часто може означати навіть різні частини мови. Наприклад, «*round*» може виступати як іменником, так і прикметником, дієсловом, прислівником і навіть прийменником. Своєю чергою, «*round*» як іменник теж має декілька значень, і т.д.

Із іншого боку, бувають і протилежні ситуації, коли одне й те саме значення може бути виражене різними словами — *синонімами*. Крім того, одне слово може мати декілька альтернативних варіантів написання, скорочення тощо. Все це є різними способами запису того самого поняття, що не дозволяє нам ототожнити слово та його значення.

Як вихід із цієї ситуації, автори WordNet прийняли рішення взяти за основну одиницю у своєму словнику *синсет* (від англ. *synset* ← *synonym set* — набір синонімів). Кожен синсет охоплює всі слова-синоніми, які мають тотожне (на думку розробників WordNet) значення. Таке групування всіх синонімів у один набір дозволяє розв'язати наявну багатозначність і невідповідність між

написанням слів і їхньою суттю. Синсет однозначно вказує саме на одне окреме поняття, один конкретний відтінок значення.

Скажімо, якщо взяти англійське слово «*car*», то у WordNet воно посилається відразу на декілька різних синсетів із відповідними назвами «*car.n.01*», «*car.n.02*», «*car.n.03*» тощо. Тут «*n*» означає «*noun*» — іменник (аналогічно є позначення і для інших частин мови), а далі йде порядковий номер значення. При цьому кожен синсет означає щось своє: «автомобіль», «вагон» і т.д.

Із іншого боку, на кожен із перелічених синсетів можуть посилатись і інші слова. Наприклад, до значення «*car.n.01*» («автомобіль») прив'язані також слова «*auto*», «*automobile*», «*machine*», «*motorcar*» — тобто всі, які можуть позначати те ж саме поняття (принаймні в одному зі своїх значень).

Таким чином, пріоритетним для WordNet є семантичне значення, а не написання слів. Тож саме для синсетів будується ієрархія понять, визначаються відношення між ними (батьківські та дочірні поняття, синоніми, антоніми тощо).

Витягнути всі значення слова за його написанням можна, скориставшись функцією **synsets():**

```
car_meanings = wn.synsets('car')
```

Вищенаведений код збереже всі синсети, пов'язані зі словом «*car*», у змінну *car_meanings*. У результаті там опиниться список із декількох значень. У подальшому можна проводити їх опрацювання, проходячи по списку циклом чи беручи з нього окремі значення за індексами абощо.

Звернувшись до окремого синсета за його повною назвою (на кшталт «*car.n.01*»), можна застосувати до нього наступні функції:

- **name()** — отримати назву синсета;
- **lemma_names()** — знайти назви лем (початкові форми слів) синсету;
- **definition()** — показати визначення (тлумачення) синсета;
- **examples()** — навести приклади вживання слова в цьому значенні.

До прикладу, такий код:

```
print(wn.synset('car.n.01').definition())
```

Має повернути пояснення, що саме означає слово «*car*» у першому значенні іменника («автомобіль»).

Проходити по всіх синсетах окремого слова зручно через цикли, наприклад:

```
for synset in wn.synsets('java', wn.NOUN):
    print(synset.name() + ':', synset.definition())
```

Такий код видасть нам назви синсетів англійського слова «java», кожну з нового рядка, при цьому через двокрапку буде наведено визначення (тлумачення) відповідного значення. Зверніть увагу, що тут за допомогою додаткового опціонального аргумента **wn.NOUN** ми фільтруємо синсети, беручи серед усіх

можливих результатів лише значення іменників. Аналогічно можна зазначати й інші частини мови.

За результатами виконання вищенаведеної програми ми можемо довідатися, що першим у словнику WordNet іде значення «острів Ява», другим — кава з цього острова, а третім — об'єктно-орієнтована мова програмування Java. Характерно, що ці синсети мають наступні назви:

- **java.n.01**
- **coffee.n.01**
- **java.n.03**

Звідси бачимо, що у словнику відсутня назва «*java.n.02*» для другого значення — сорту кави. Натомість цей синсет перенаправляється на перше значення слова «*coffee*». Отже, на думку розробників, кава з острова Ява не є достатньо важливим поняттям сама по собі. А її відмінності від будь-якої іншої кави не настільки суттєві, аби виділяти під це поняття окремий синсет. Через це вони об'єднали друге значення «*java*» та перше значення «*coffee*» в один спільний синсет, прив'язаний до слова «*coffee*», адже для нього значення «напій» є основним, а не другим. Тим часом усі можливі сорти кави, що позначаються іншими словами (за наявності), будуть скеровані до нього.

Тим не менше, третє значення «*java*» має порядковий номер синсета 3, а не 2, оскільки друге значення вже зайняте і позначає каву. Мова програмування в цій послідовності йде третьою, через що і називається «*java.n.03*». Слід пам'ятати про цю особливість у нумерації синсетів: деякі номери можуть бути пропущені.

Наступний код може допомогти нам знайти синоніми або альтернативні варіанти написання того самого значення:

```
for synset in wn.synsets('java'):
    print(synset.lemma_names())
```

Запустивши цю програму, ми побачимо такий результат:

```
['Java']
['coffee', 'java']
['Java']
```

Тут можна звернути увагу на наступні особливості. Перш за все, острів і мова програмування не мають синонімів чи інших варіантів написання. Натомість каву позначають два слова — «*coffee*» та «*java*». Крім того, залежно від значення, слово «*java*» може писатись як із малої, так і з великої літери. Проте це не впливає на формування синсетів. І загальні, і власні назви групуються в один список значень.

Розглянемо деякі відношення між синсетами, які визначають зв'язки між ними. Це зокрема *гіпероніми* (щось більше, загальніше) та *гіпоніми* (щось менше, конкретніше). Їх можна витягнути для кожного синсета, застосувавши до нього функції **hypernyms()** і **hyponyms()** аналогічно до **lemma_names()** у прикладі вище. Результатом виконання кожної функції буде список синсетів.

Пройшовши по всіх трьох значеннях слова «*java*», отримаємо наступний результат по гіперонімах:

```
[]
[Synset('beverage.n.01')]
[Synset('object-oriented_programming_language.n.01')]
```

Отже, для мови Java більш загальним, родовим (батьківським) поняттям у ієрархії словника WordNet є «об'єктно-орієнтована мова програмування», а для кави — «напій». Справді, нині є багато мов ООП, серед яких Java є одним частковим випадком, тобто дочірнім поняттям. Так само є безліч напоїв, одним із прикладів яких є кава. Що ж до острова, оскільки Ява не є типом земної поверхні (на відміну від загальних слів «острів», «півострів», «материк» тощо), а просто одним конкретним островом, єдиним у своєму роді, то для нього гіперонімів не визначено взагалі. Теоретично можна було би прив'язати до слова «острів» як дочірні поняття назви всіх можливих островів на Землі, але у WordNet цього робити не стали. Натомість для слова «острів» («*island*») видовими поняттями є «бар'єрний острів» і лише кілька окремих груп островів.

Тепер візьмемо гіпоніми по трьох значеннях слова «*java*»:

```
[]
[Synset('cafe_au_lait.n.01'), Synset('cafe_noir.n.01'),
Synset('cafe_royale.n.01'), Synset('cappuccino.n.01'),
Synset('coffee_substitute.n.01'),
Synset('decaffeinated_coffee.n.01'), Synset('drip_coffee.n.01'),
Synset('espresso.n.01'), Synset('iced_coffee.n.01'),
Synset('instant_coffee.n.01'), Synset('irish_coffee.n.01'),
Synset('mocha.n.03'), Synset('turkish_coffee.n.01')]
[]
```

Результати свідчать про те, що дочірніх (видових, конкретніших) понять до острова Ява та мови програмування у WordNet не визначено, тоді як для кави є цілий ряд підвидів. Серед них — капучіно, еспресо, мока, ірландська та турецька кава тощо. З такою ієрархією можна посперечатись, однак вона відображає точку зору авторів словника. В іншому засобі відношення між цими самими поняттями могли би бути інакшими.

Загалом в ієрархії WordNet ми можемо пройти від більшості понять як униз до найбільш конкретних речей, так і вгору до найбільших абстракцій. На найвищому рівні знаходиться слово «*entity*» — «сутність», відносно якого всі інші поняття в ієрархії є видовими (прямо чи опосередковано).

Глибина в цій ієрархії визначається кількістю рівнів, які треба пройти від корінного поняття «*entity*» донизу, аби дістатися заданого. Глибину синсета можна отримати через функцію **min_depth()** — наприклад, таким чином:

```
tea = wn.synset('tea.n.01')
print('tea:', tea.min_depth())
```

Результатом для чаю («*tea*»), як і для кави («*coffee*») та інших їхніх сестринських понять буде число 6. Для підвидів кави глибина становить 7 і більше, натомість ідучи вгору, знайдемо слово «напій» («*beverage*») із глибиною 5, їжу («*food*») зі значенням 4 і т.д. аж до слова «сутність» («*entity*»), яке має глибину 0.

Аналізуючи глибину понять і їхні родо-видові відношення між собою, можна спостерегти деякі неочевидні речі. Скажімо, слово «*juice*» («сік») у WordNet не є нащадком слова «*beverage*» («напій»), натомість відноситься до їжі загалом («*food*»). Усі ці виявлені особливості дають розуміння про суб'єктивність при класифікації понять у світі та можливість різних підходів до цього питання. Причому це проявляється навіть у межах однієї мови (в нашому випадку англійської), не кажучи про інші природні мови, в кожній із яких картину світу може бути відображено зовсім інакше.

Як ми вже побачили, при опрацюванні знань, виражених у текстовій формі, є багато нюансів, які залежать від особливостей конкретної мови та суб'єктивного погляду того, хто висловлює або інтерпретує ці знання.

Звісно, так само суб'єктивним буде і питання семантичної подібності будь-яких двох понять між собою. Тим не менше, в рамках певної визначеної ієрархії на кшталт WordNet це все ж можна спробувати обчислити математично. Для розв'язання цієї задачі ми можемо врахувати абсолютне та відносне розташування понять у ієрархії.

Є цілий ряд різних методів для визначення семантичної подібності. Проте більшість із них беруть до уваги ті самі два основні показники — спільний гіперонім для обох понять, а також глибину цих понять у ієрархічній структурі словника.

Щодо спільного гіпероніма, у WordNet для його знаходження є окрема функція — **lowest_common_hypernyms().** Наприклад, для кави та чаю це буде:

```
coffee = wn.synset('coffee.n.01')
tea = wn.synset('tea.n.01')
print(coffee.lowest_common_hypernyms(tea))
```

Результатом буде синсет «*beverage.n.01*».

Проте не обов'язково шукати гіпероніми та обчислювати глибину ієрархії вручну. Для знаходження семантичної подібності можна також скористатися готовими функціями, вбудованими у WordNet.

Одним із найпростіших варіантів є застосування методу *Path Distance Similarity*. Його можна викликати для вищезазначених синсетів *coffee* та *tea* так:

```
print(coffee.path_similarity(tea))
```

Цей код дасть нам значення 0,333, тобто 1/3. Отже, сестринські поняття (в яких є спільний предок рівнем вище) вважаються подібними на 33%. Провівши серію експериментів, можна виявити, що предок і нащадок матимуть значення 0,5 (подібність 50%), а порівняння поняття із самим собою видасть результат 1,0

(тобто 100% подібність). Натомість беручи більш віддалені поняття, будемо отримувати все менші частки одиниці: 0,25 (1/4), 0,2 (1/5) і т.д., наближаючись до нуля. Таким чином обчислює близькість метод *Path Distance Similarity,* який дає результати за шкалою від 0 до 1.

Ще одним популярним методом є *Wu–Palmer Similarity,* названий на честь його розробників — Жибяо Ву та Марти Палмер. Обчислити подібність із його допомогою можна наступним чином:

```
print(coffee.wup_similarity(tea))
```

Якщо за *Path Distance Similarity* для цих синсетів ми мали всього 0,333, то тут уже буде 0,888. І хоча в методі Ву—Палмер використовується та сама шкала від 0 до 1, та до уваги береться не лише покрокова близькість між поняттями, а також і їхня глибина в ієрархії. Як наслідок, і результат виходить настільки суттєво вищим для пари «кава» і «чай». Адже ці поняття самі по собі вже є доволі конкретними, а не загальними. Тож вони вважаються між собою подібнішими, ніж інші два сестринські поняття, які є абстрактнішими та розташовані в ієрархії вище. Скажімо, для понять «організм» і «жива клітина», які за *Path Distance Similarity* дають так само 0,333, за методом *Wu–Palmer Similarity* отримаємо результат 0,833 через те, що вони перебувають у WordNet кількома рівнями вище.

Наступним розглянемо метод *Leacock Chodorow Similarity,* який теж названо за прізвищами авторів — Клаудії Лікок і Мартіна Ходорова. В цьому підході так само враховується і відстань між поняттями, і їхня глибина, та спосіб обчислення кінцевого результату є відмінним. Для розрахунку значення близькості в цьому методі береться формула:

$$-log(p/2d),$$

де p — найкоротший шлях між поняттями;
d — глибина таксономії.

Очевидно, що при таких розрахунках шкала вже не лежатиме в межах від 0 до 1. Як приклад, для тієї самої пари «кава» та «чай» результат складе 2,539. Звідси робимо висновок, що обчислення подібності можна проводити різними способами, проте результати за різними методами не є зіставними між собою через особливості розрахунків і шкал.

Серед інших підходів до визначення близькості понять можна згадати такі методи, як *Resnik Similarity, Jiang–Conrath Similarity, Lin Similarity* тощо. Всі вони мають свої формули розрахунку та дають різні результати. Крім шляху між семантичними значеннями та глибини в ієрархії, вони використовують додаткові параметри. Одним із них є показник *Information Content* — величина, що обчислюється на основі використання слів у корпусі текстів, а також має вагові множники для різних значень.

Методи визначення семантичної подібності є корисними для пошуку інформації за значенням. Це саме той інтелектуальний пошук, про який було згадано вище і який став доступним завдяки використанню семантичних мереж.

Наприклад, якщо людина введе в пошуковик «транспорт», їй можуть показати не лише документи, що містять саме це конкретне слово, а й також будь-які інші, в яких згадуються дочірні поняття: «автомобілі», «літаки», «човни» тощо.

Крім семантичної близькості, певну цінність також становить і визначення подібності слів за їхнім написанням. Це використовується вже для інших задач, зокрема для автоматичної перевірки правопису. Одним із популярних донині методів тут лишається відстань редагування («*edit distance*»). Цей термін також відомий як відстань Левенштейна — за прізвищем математика, який запропонував такий підхід у 1965 році. Згодом інший дослідник, Фредерік Дамерау, доповнив цей метод, унаслідок чого з'явилась його модифікація, відома як відстань Дамерау—Левенштейна.

Класична **відстань редагування,** або відстань власне Левенштейна, являє собою міру різниці двох рядків між собою. Вона обчислюється як мінімальна кількість операцій, необхідних для перетворення одного рядка на інший. При цьому допускаються такі операції, як:

- вставка символа;
- видалення символа;
- заміна одного символа на інший.

Щодо модифікації Дамерау, в його версії за одну операцію вважається також перестановка (зміна послідовності двох сусідніх символів). У класичній відстані Левенштейна для цього вимагалося би 2 операції заміни або 1 видалення + 1 вставка.

Застосування відстані редагування може бути корисним у таких сферах:

- виправлення помилок у тексті:
  - для перевірки правопису;
  - для коригування OCR (сканованих / сфотографованих текстів);
- порівняння послідовностей символів:
  - для версій текстів, програмного коду тощо;
  - для послідовностей ланцюжків у біології (ДНК, РНК і т.д.);
- запобігання шахрайству (пошук фейкових торгових марок і адрес);
- оцінювання взаємної зрозумілості подібних мов.

Слід також зауважити, що перевірка правопису потрібна в багатьох сферах життя, а помилки в написанні слів можуть бути спричинені цілим рядом чинників. Крім очевидних проблем із технічними одруками та неграмотністю, такі помилки можуть бути викликані особливостями окремої мови і специфікою самих текстів. Зокрема є мови, в яких правила читання літер є досить простими й однозначними. Завдяки цьому написання слів не викликає серйозних труднощів навіть у тих, хто лише починає вивчати мову. Натомість у інших мовах зі складним правописом деякі питання є проблематичними навіть для носіїв. Тож залежно від конкретної мови актуальність перевірки написання слів може бути дещо вищою або нижчою.

Крім того, незалежно від мови, особливі труднощі традиційно викликають власні назви та імена. Можна згадати чимало відомих людей, наприклад, у США, які мали походження з інших країн і відповідно слов'янські, німецькі, французькі та інші прізвища. З плином часу ті самі прізвища, які раніше читалися за

правилами мов оригіналу, стали вимовлятися ближче до правил англійської. Таким чином Палагнюк став «Поланіком», Ваховські — «Вачовськими», Маслов — «Маслоу», Хомський — «Чомскі» тощо. Тим не менше, в офіційних документах зберігається початкове написання імен та прізвищ, через що для них немає однозначної відповідності між літерами та звуками. Натомість є безліч варіантів, як той самий звук може бути записаний літерами. Це може викликати додаткові труднощі при збереженні, пошуку та опрацюванні текстових даних.

Перейдемо до технічної реалізації обрахунку відстані редагування. Якщо маємо рядок X довжиною n символів і рядок Y довжиною m символів, то відстань редагування між ними $D(n, m)$ можна обчислити покроково, щоразу при відмінностях додаючи 1 за кожну необхідну операцію, описану вище. Для крайніх випадків, коли один із рядків є порожнім, $D(n, m) = m$ (якщо порожнім є X) або $D(n, m) = n$ (якщо порожнім є Y).

Розрахувавши відстань редагування для англійських слів «*elephant*» («слон») і «*relevant*» («відповідний»), можемо отримати значення 3. Справді, для перетворення одного рядка на інший чи навпаки мінімально знадобиться три операції. Наприклад, видалити зі слова «*relevant*» першу літеру *r*, замінити *v* на *p*, а далі вставити літеру *h*. Така відстань для двох слів по 8 літер кожне є відносно невеликою. І пишуться, і читаються вони досить схоже.

Очевидно, що якби ми порівняли цю пару слів за їхніми семантичними значеннями, подібність була би значно меншою, адже це різні частини мови, які позначають зовсім різні поняття. Тим не менше, саме близькість за написанням є визначальною для перевірки правопису, коли у разі орфографічної помилки чи випадкового одруку ми можемо знайти подібні слова для заміни неправильного.

Станом на сьогодні відстань Левенштейна та Дамерау—Левенштейна не є єдиним наявним методом для визначення подібності написання слів. Серед альтернатив можна також згадати наступні:

- Needleman–Wunch;
- Smith–Waterman;
- Monge–Elkan;
- Jaro;
- Jaro–Winkler.

Якщо порівняти їхню ефективність, побачимо, що нині всі вони дають кращі результати за класичний метод Левенштейна. Проте його доволі часто використовують і досі, оскільки його перевагами є простота реалізації алгоритму та висока ефективність для коротких рядків. Тим не менше, значним недоліком цього методу є складність обчислення, що приблизно дорівнює добутку довжин двох рядків. Таким чином, доцільно використовувати відстань Левенштейна і Дамерау—Левенштейна для випадків, коли порівнювані рядки є короткими — наприклад, для окремих слів, а не довгих текстів.

Розглянемо приклад практичного застосування відстані редагування для перевірки правопису. Поставимо задачу наступним чином:

- користувач вводить певний текст;
- програма має перевірити кожне слово на наявність у її словнику;

- якщо такого слова немає, треба підібрати зі словника *N* найближчих (найбільш подібних) слів, аби запропонувати їх на заміну.

Перш за все, нам буде потрібен словник із еталонними словами, які система визначатиме як написані правильно. Далі ми будемо попарно порівнювати слово, введене користувачем, із усіма словами, наявними в нашому словнику. Для кожної пари будемо розраховувати відстань редагування.

Якщо для якоїсь пари отримаємо значення 0, це означатиме, що введене слово наявне у словнику, тож ми можемо його прийняти та йти далі. Якщо ж відстань 0 відсутня, нам слід відсортувати отримані значення відстаней за зростанням. Слова, що матимуть відстань 1 від введеного користувачем, будуть найближчими до нього, адже вимагатимуть лише однієї операції для приведення некоректного (згідно зі словником) слова до еталону. Слова з відстанню 2 будуть менш імовірними претендентами на заміну, і т.д. Якщо ми маємо видати користувачам 3, 5 чи будь-яку іншу кількість пропозицій щодо потенційної заміни неправильного слова на правильне, обмежимо відсортований список претендентів саме цією кількістю.

Можемо перевірити описаний підхід, узявши словник із 1000 найчастіше вживаних слів англійської мови. Це досить небагато, проте в ньому вже будуть міститися такі слова, як «*they*», «*them*», «*their*», «*there*» тощо. Отже, якщо користувач введе щось із цього списку, його слово буде знайдено. Натомість якщо буде введено щось на кшталт «*theire*», результатом підбору 5 найбільш імовірних варіантів на заміну буде такий список слів і їхніх відстаней редагування:

- their (1);
- there (1);
- here (2);
- these (2);
- third (2).

Як бачимо, чим більше ми віддаляємось від введеного слова, тим менш правдоподібним стає припущення про те, що людина помилилась при наборі тексту та насправді хотіла ввести слова з відстанню редагування 2, 3, і т.д. Тож, можливо, слід обмежувати пропозиції (підказки) не просто певною кількістю варіантів, а саме деяким значенням відстані редагування — скажімо, не більше 1 або 2.

Література

Див. джерела №№ 1, 2, 3, 4, 6, 7.

ЛАБОРАТОРНА РОБОТА 3.

Автоматична класифікація тексту (сентимент-аналіз)

Класифікація тексту — це пошук шаблонів у текстових даних для розбиття окремих текстів, речень, слів тощо на певні категорії. До задач автоматичної класифікації належать зокрема наступні:

- визначення роду слова;
- розмітка слів за частинами мови (*PoS-tagging*);
- класифікація текстів за змістом:
  - поділ на теми;
  - визначення спаму;
- сентимент-аналіз;
- (інші).

Сентимент-аналіз, або аналіз тональності тексту, полягає у маркуванні текстів залежно від їхньої емоційної забарвленості. Найпростішим випадком є поділ на дві категорії — позитивні та негативні. При більш докладному аналізі можна також виділити нейтральні тексти, а для позитивних і негативних провести градацію за певною шкалою (наприклад, +2, +1, 0, –1, –2 або 1, 2, ..., 10). Одним із застосувань автоматичної класифікації такого роду є оцінювання відгуків на товари, послуги та інше.

В основі сентимент-аналізу зазвичай лежать методи штучного інтелекту, зокрема машинного навчання. Спершу створюється тренувальний набір, у якому кожен текст промарковано відповідними позначками (наприклад, позитивний чи негативний). Цей набір передається моделі машинного навчання для тренування. При цьому модель із допомогою штучного інтелекту самостійно визначає певні закономірності в текстах, які дозволяють віднести їх до тієї чи іншої категорії.

На другому етапі береться тестовий набір, для якого теж відомі позначки, однак цей набір передається моделі на аналіз уже без них. Модель намагається промаркувати кожен текст самостійно, визначаючи відповідний клас на основі попереднього навчання. Зіставивши отримані висновки моделі з реальними позначками, можна охарактеризувати точність моделі.

Розглянемо наступну задачу. Дано тренувальний набір, що містить 2000 відгуків англійською мовою на різні фільми. Набір є збалансованим і містить по 1000 позитивних і негативних відгуків. Використовуючи ці дані та певну модель машинного навчання, слід навчити її визначати тональність відгуку лише за його текстом. Після перевірки на тестовому наборі треба визначити точність наявних результатів.

Варіантом розв’язання цієї задачі є наступний алгоритм. Спершу зберемо всі слова з усіх наявних відгуків. Потім знайдемо найбільш частотні слова, тобто ті, які зустрічаються в цьому корпусі текстів найчастіше. Для кожного слова, починаючи з найчастотніших, визначимо, в якій категорії відгуків (позитивні чи негативні) воно зустрічається частіше і наскільки. Надалі при аналізі нових

текстів без маркувань будемо рахувати, слів із якої категорії в ньому міститься більше.

В наступному прикладі коду імпортується корпус текстів **movie_reviews,** який містить вищезгадані відгуки, розподілені по папках *pos* і *neg*. Також тут імпортується модуль **random,** який дозволяє перемішувати тексти випадковим чином для формування тренувальних і тестових наборів даних.

```
from nltk.corpus import movie_reviews
import random
```

Сформуємо список усіх наявних слів у нижньому регістрі (аби на аналіз не впливала позиція слова на початку чи в середині / кінці речення). Слова сортуються за спаданням частотності через функцію **FreqDist().** На екран виводяться 15 найбільш уживаних слів у заданому корпусі відгуків:

```
all_words = []
for w in movie_reviews.words():
    all_words.append(w.lower())
all_words = nltk.FreqDist(all_words)
print(all_words.most_common(15))
```

У результаті можемо отримати приблизно такий список:

```
[(',', 77717), ('the', 76529), ('.', 65876), ('a', 38106), ('and',
35576), ('of', 34123), ('to', 31937), ("'", 30585), ('is', 25195),
('in', 21822), ('s', 18513), ('"', 17612), ('it', 16107), ('that',
15924), ('-', 15595)]
```

Як бачимо, до списку ввійшли не лише повноцінні слова, а й короткі форми на кшталт «*’s*», а також розділові знаки. Це відбувається через те, що функція **words()** не просто шукає в тексті слова, а розбиває його на складові за пробілами. Таке розбиття називається токенізацією, і за бажання можна в подальшому відфільтрувати отриманий результат, видаливши зайві елементи списку (токени).

Між іншим, тут можна зауважити ще одну особливість, яка стосується саме Python. У консоль майже всі токени виведено в одинарних лапках, однак самий знак «’» виділено подвійними.

Список *all_words,* утворений вище, можна використати для перевірки кількості вживань певного слова в цілому наборі текстів. Для прикладу довідаємося, скільки разів у відгуках зустрічаються слова «*stupid*» і «*excellent*»:

```
print(all_words['stupid'])
print(all_words['excellent'])
```

Після виконання програми побачимо, що перше слово вживається 253 рази, а друге — 184. Можна висувати різні версії, чому так сталось. Однією з версій може бути те, що негативна реакція на фільм (яка випливає з ужитої лексики) частіше викликає бажання лишити відгук, ніж позитивна. Втім, аби підтвердити

чи спростувати таку гіпотезу, слід узяти весь обсяг відгуків на певному ресурсі (скажімо, IMDB, звідки і було наповнено корпус **movie_reviews** у NLTK) та порівняти реальну кількість позитивних і негативних.

Іншою та більш імовірною версією є те, що саме по собі слово «*stupid*» загалом є більш уживаним у англійській мові, ніж «*excellent*». Аби перевірити це, можна скористатися певним частотним словником. Що ж до роботи з нашим корпусом, тут можна спробувати пошукати інші позитивно забарвлені слова. Зробивши це, виявимо, що «*good*» зустрічається в цьому наборі відгуків 2411 разів, «*great*» — 1148, а «*nice*» — 344, що є вищими показниками, ніж у слова «*stupid*». Отже, скоріш за все, слово «*excellent*» просто не належить до найчастіше вживаної лексики в англійській.

Тепер перейдемо до роботи з машинним навчанням. Передусім сформуємо список файлів із відгуками, розбитих за категоріями (позитивні та негативні) та перемішаємо його випадковим чином:

```
doc = [(list(movie_reviews.words(fileid)), category)
      for category in movie_reviews.categories()
      for fileid in movie_reviews.fileids(category)]
random.shuffle(doc)
```

Потім створимо список *word_features*, що міститиме перші 3000 найбільш частотних елементів із *all_words*. А також оголосимо функцію *find_features()* наступного змісту:

```
word_features = list(all_words.keys())[:3000]
def find_features(d):
  words = set(d)
  features = {}
  for w in word_features:
    features[w] = (w in words)
  return features
```

Ця функція братиме на вхід слова з певного тексту (наприклад, файлу з окремим відгуком) і формуватиме з них множину (без повторів). Далі створюється словник *features,* який наповнюється ключами — всіма 3000 словами з *word_features*, та відповідними їм значеннями — наявністю чи відсутністю кожного з цих слів у поданому на вхід наборі слів (із відгуку абощо). На вихід ця функція повертає утворений і наповнений словник *features*.

Можемо потестувати роботу створеної функції так:

```
print(find_features(movie_reviews.words('neg/cv000_29416.txt')))
```

Цей код візьме файл *cv000_29416.txt* із папки *neg* (негативні відгуки) і перевірить, які з 3000 найчастотніших слів у ньому містяться, а які — ні. При цьому на екран буде виведено всі 3000 слів із мітками «*True*» або «*False*». Тож на

майбутнє було б добре вдосконалити цей код і відображати лише ті слова, які зустрілись, нехтуючи мітками «*False*», яких зазвичай буде переважна більшість.

Далі перейдемо до машинного навчання. Спершу треба розбити наявні в нас 2000 відгуків на тренувальний і тестовий набори. Для тренувального візьмемо перші 1900 відгуків, для тестового залишимо решту 100. Варто пам'ятати, що на початку програми ми перемішали відгуки через функцію **random.shuffle().** Тож при кожному наступному запуску коду конкретний вміст тренувального та тестового наборів буде іншим.

```
featuresets = [(find_features(rev), category)
                for (rev, category) in doc]
training_set = featuresets[:1900]
testing_set = featuresets[1900:]
```

Безпосередньо машинне навчання можна проводити з використанням різних методів. Одним із найпростіших і поширених класичних методів є наївний Баєсів алгоритм, що ґрунтується на теоремі Баєса. Наївним його називають через те, що для спрощення задачі він припускає, що всі риси, наявні у вибірці, є незалежними між собою. Таким чином, при визначенні приналежності об'єкта до певного класу наявність кожної риси рахується окремо.

Подібне спрощення впливає на точність результатів: у середньому наївний Баєсів алгоритм дає точність від 60% до 90%, і вийти за ці межі досить важко. Однак за рахунок нехтування взаємозалежностями цей алгоритм є простим для побудови та легко масштабується. Тож його можна використовувати для простих задач, де не вимагається особлива точність.

Наївний Баєсів алгоритм є вбудованим у NLTK. Викликати його та навчити модель на нашому тестовому наборі з його допомогою можна наступним чином:

```
classifier = nltk.NaiveBayesClassifier.train(training_set)
```

Після навчання проведемо перевірку моделі на тестовому наборі та виведемо на екран точність отриманих результатів у відсотках:

```
print('Naive Bayes Algorithm accuracy percent:',
(nltk.classify.accuracy(classifier, testing_set))*100)
```

Результат може бути, наприклад, таким:

```
Naive Bayes Algorithm accuracy percent: 75.0
```

Враховуючи вищеописані особливості алгоритму, якщо ми отримали 75%, це можна вважати досить непоганим результатом. Також пам'ятайте, що оскільки відгуки щоразу перемішуються і вміст тренувального та тестового набору є різним, так само різним буде і значення точності. Скласти більш об'єктивне враження про точність цього (чи іншого) алгоритму можна, провівши серію експериментів та обчисливши середнє арифметичне отриманих значень.

До слова, для класифікації можна скористатися й іншими алгоритмами машинного навчання. Порівнявши їхню точність із наївним Баєсовим алгоритмом, можна виявити, що деякі з них дають кращі результати. Зокрема в NLTK є модулі з іншими класифікаторами.

Крім того, є цілий ряд інших бібліотек для Python, які можна підключити до проєкту. Наприклад, у ***Scikit-learn*** доступні такі алгоритми, як ***MultinomialNB, GaussianNB*** та ***BernoulliNB*** — вдосконалені модифікації наївного Баєса (NB). Серед інших поширених методів — ***SVC, LinearSVC, NuSVC, SGDClassifier, LogisticRegression*** тощо.

Іще один зручний інструмент — функція **show_most_informative_features().** Вона дозволяє побачити саме ті риси в наборі, які є найбільш виразними і найбільш чітко показують приналежність кожного об’єкта до того чи іншого класу. В нашому випадку з відгуками це слова, які значно частіше зустрічаються у позитивних, ніж у негативних, і навпаки.

Викличемо цю функцію, аби побачити 5 найбільш інформативних слів:

```
classifier.show_most_informative_features(5)
```

Результат може бути наступним:

```
Most Informative Features
  astounding = True   pos : neg  =  11.7 : 1.0
  incoherent = True   neg : pos  =   9.6 : 1.0
    predator = True   neg : pos  =   8.3 : 1.0
     wasting = True   neg : pos  =   8.3 : 1.0
breathtaking = True   pos : neg  =   7.5 : 1.0
```

Як бачимо, співвідношення ***pos*** **:** ***neg*** для найбільш інформативного слова «*astounding*» («вражаючий», «приголомшливий») становить 11,7 до 1. Це означає, що воно зустрілось у позитивних відгуках у 11,7 разів частіше, ніж у негативних. Цього можна було очікувати, враховуючи позитивне забарвлення самого слова.

В той же час, необхідно звернути увагу на те, що серед інформативних слів є також і ті, що зустрілись у негативних відгуках частіше. Скажімо, «*incoherent*» має співвідношення *neg* **:** *pos* зі значенням 9,6. Слід розуміти, що інформативність є абстрактною характеристикою, яка показує виразність загалом — незалежно від того, до якого саме класу віднесено той чи інший об’єкт.

Література

Див. джерела №№ 1, 2, 3, 4, 5, 6, 7, 9.

ЛАБОРАТОРНА РОБОТА 4.

Взаємодія з API для лінгвістичних завдань

Автоматичне збирання текстів використовується для різних цілей. Це пошук інформації в інтернеті за певним запитом; створення корпусів текстів окремих авторів чи організацій або збірок за деякою тематикою; наповнення тренувальних наборів для машинного навчання з метою подальшої генерації текстів; збирання та структурування статистичних даних (метеоумови, курси валют, торгівля тощо) із текстів для подальшої візуалізації, інтелектуального аналізу даних і т.д.

Останнім часом серед інших актуальними є також проєкти, в яких із допомогою методів комп'ютерної лінгвістики робляться спроби автоматичного визначення та маркування мови ворожнечі та токсичних текстів (наприклад, у соцмережах), а також ідентифікації авторства анонімних текстів завдяки стилеметрії та аналізу характерних рис тексту.

Крім того, збирання текстів із відкритих джерел активно використовується в OSINT-аналітиці (*open-source intelligence*). Можна не лише збирати дані самі по собі, а й проводити дослідження для пошуку першоджерела новини чи іншого тексту. Це можливо зробити за рахунок аналізу часу та вмісту публікацій, відстежуючи місця витоків даних, ланцюжки поширення дезінформації тощо.

Із технічного боку збирання текстів із вебджерел можливе різними шляхами. Нині одними з поширених підходів є робота з API та вебскрапінг.

API (від англ. *Application Programming Interface*) — інтерфейс, тобто сполучна ланка, для зв'язку програм між собою. Зазвичай цими програмами є клієнтський додаток і певний сервіс, розміщений на вебсервері. Серед відомих API слід назвати інтерфейси, створені популярними соцмережами, месенджерами, сайтами різного профілю: ***Google, Facebook, Instagram, Twitter (X), Last.fm, OpenWeather, ENTSO-E*** тощо.

Одним зі зручних способів збирання великих обсягів текстової інформації з відкритих джерел є використання Telegram API. Самий месенджер Telegram розроблявся ще з 2013 року, в Україні набув значного поширення з 2017 року. Станом на квітень 2026 року цей засіб мав 1 млрд. активних користувачів у світі щомісяця, серед яких 500 млн. — щодня.

Серед особливостей месенджера:

- хмарне зберігання даних (що є нині стандартом для сучасних великих сервісів);
- кросплатформність (його можна встановити на комп'ютер навіть без наявності мобільного додатка, а також є веб-версія, що працює через браузер майже з будь-якого пристрою);
- інтерфейс багатьма світовими мовами;
- можливість наскрізного шифрування в режимі таємних чатів (E2EE — *end-to-end encryption*, при якому ключі зберігаються лише на клієнтських пристроях, а не на серверах);

- відкритий код (якщо не довіряєте офіційній версії, що поширюється як файл .exe / .apk, можна проглянути вихідний код і скомпілювати перевірену версію додатка самостійно);
- можливість не лише обміну повідомленнями з текстом і медіа, а й збереження файлів, ведення блогів у каналах, дискусій у коментарях і загальних чатах, використання ботів для різних потреб тощо.

Що стосується каналів, вони в Telegram є двох типів:

- публічні (доступні за коротким нікнеймом, їх можна знайти за ним або назвою через загальний пошук у месенджері, таких каналів можна створити до 10 в безкоштовній версії та до 20 у платній);
- приватні (не мають нікнейма та доступні для підписників лише за постійним чи тимчасовим запрошенням від власника, таких каналів можна створити безліч, як із підписниками, так і без них).

Характерно, що якщо не поширювати лінк на приватний канал, він стане приватним у повному сенсі цього слова, адже до нього не матиме доступу ніхто, крім самого власника. Завдяки цьому такі канали можна використовувати для зручної організації власних даних (нотаток, медіафайлів, документів, посилань, збережених постів і повідомлень і т.п.), на кшталт діалогу «Saved Messages». Однак не варто забувати, що все це зберігається на серверах без шифрування, а отже, не слід розміщати в таких сховищах будь-чиї чутливі персональні дані.

Що ж до публічної інформації, яка оприлюднюється у вільному доступі через публічні канали, ми можемо збирати її шляхом створення власних додатків, які будуть отримувати доступ до Telegram API. Для цього слід зареєструвати свій додаток і отримати параметри доступу. Після цього можна буде працювати як із власним обліковим записом (профілем, чатами, особистими повідомленнями), так і з каналами.

При цьому важливо розрізняти Telegram API і Telegram Bot API, що є двома різними способами зв'язку з месенджером. Для використання кожного з них потрібна окрема реєстрація. Telegram Bot API застосовується виключно для створення ботів і роботи з ними. Натомість для роботи з власним профілем, каналами тощо нам потрібен звичайний Telegram API, що використовує TDLib — *Telegram Database Library* — для доступу до всіх необхідних нам даних.

Із докладною інформацією про API можна ознайомитись на офіційному ресурсі для розробників: **https://core.telegram.org.**

Для роботи з API через сучасні мови програмування створено багато зручних інструментів. Це спеціальні бібліотеки та модулі для Python, C# та інших мов. Зокрема для Python популярною бібліотекою є Telethon, що розробляється з 2016 року.

Станом на червень 2026 року найновішою версією Telethon є 1.44.0. Проте зверніть увагу, що приклади коду, які буде наведено нижче, створено на версії 0.18.0.3, що є сумісною зі старими версіями інтерпретатора Python аж до 3.5.0 включно. Це важливо для забезпечення можливості коректної роботи створених додатків навіть на комп'ютерах зі старими ОС і програмними засобами. Тим не менше, наведені приклади коду можна використати і в найновіших версіях

бібліотеки після невеликих правок по синтаксису (зокрема додання асинхронних операцій).

Так чи інакше, перед використанням будь-якої бібліотеки нам необхідно зареєструвати свій майбутній додаток на сторінці, призначеній для розробників: **https://my.telegram.org/apps.** Процес реєстрації досить простий, якщо порівняти з деякими іншими сучасними сервісами. Зокрема форма реєстрації займає всього лише одну невелику сторінку. Також перевірка даних проходить автоматично, і немає потреби листуватися з адміністраторами щодо призначення й особливостей Вашого додатку (що може знадобитися, наприклад, для роботи з X [Twitter]).

У формі реєстрації слід зазначити наступне:

- назву додатку (будь-яку);
- коротке ім’я (лише літери латинки та цифри, обсяг 5–32 символів);
- URL (слід лишити порожнім, якщо проєкт не є веб-додатком);
- платформу, для якої створюється додаток (зазвичай це Desktop);
- опис (тут бажано написати хоча би 15–20 слів англійською).

Важливо, що якщо опис буде закоротким, це призведе до помилки при реєстрації. Проте із загального повідомлення «ERROR» не буде зрозуміло, в чому проблема. Тим не менше, з практики найчастіше викликає цю помилку саме замалий обсяг опису додатку. Тож краще відразу помістіть туди достатньо тексту.

Якщо реєстрація успішна, на наступному екрані з’являться 2 параметри, які слід зберегти, адже саме за ними буде проходити з’єднання Вашого додатку з серверами Telegram і буде можливою робота з API:

- API ID (ідентифікатор, номер вашого додатку);
- API hash (пароль, кодове слово для доступу до даних).

Рештою наведеної технічної інформації можна знехтувати, натомість ID та hash можна відразу скопіювати до коду програми та зберегти їх у відповідних змінних цілочисельного та рядкового типів.

Перед тим, як встановлювати з’єднання та працювати з даними, варто наголосити на системі безпеки Telegram і труднощах, із якими можуть стикнутися розробники через недотримання певних базових правил роботи. Річ у тому, що задля безпеки користувачів і серверів будь-яка підозріла активність як людей, так і додатків, що працюють через API, призводить до блокування облікового запису, з якого вона була зафіксована. Зокрема Вам можуть обмежити доступ до акаунту приблизно на 22 години у випадку т. зв. переповнення (*FloodWaitError*).

Така ситуація стається, серед іншого, при 5-разовій невдалій авторизації (неправильно введено пароль, код логіну тощо) або при зміні параметрів сесії. Під зміною параметрів розуміється як зміна IP-адреси, з якої відбувається вхід, так і зміна пристрою, що може відстежуватись через MAC-адресу. Тож очевидно, що якщо заходити одночасно чи з невеликим інтервалом із різних пристроїв (телефон, планшет, ноутбук, десктопний комп’ютер) або адрес (через кабельний інтернет, Wi-Fi, інтернет від мобільного оператора, а також із використанням VPN та без), це може трактуватись як підозріла активність і розцінюватись як спроба зламу вашого облікового запису зловмисниками. Отже, при тестуванні додатків слід пам’ятати про можливі обмеження, уважно вводити свої дані доступу, а

також не забувати підтверджувати свою активність із інших пристроїв у разі запитів через офіційний додаток Telegram.

Безпосередньо для під'єднання до API нам, окрім API ID та API hash іще знадобиться ввести свій номер телефону чи нікнейм. Це взаємозамінні параметри, проте в разі під'єднання за нікнеймом система все одно запитає номер телефону для пітвердження, і його слід буде ввести в консоль. У разі з'єднання відразу за номером телефону процес авторизації буде на один крок коротшим. Номер слід вводити в міжнародному форматі, зі знаком «+» і кодом країни: наприклад, +380...

Отже, код для збереження параметрів доступу, створення та запуску клієнта може виглядати наступним чином:

```
api_id = ...
api_hash = '...'
username = '...' # або:
phone_number = '+380...'

from telethon import TelegramClient

client = TelegramClient(username, api_id, api_hash)
client.start()
```

Після з'єднання за номером телефону чи його введення в консоль на цей номер має прийти код підтвердження. Зазвичай він надсилається в офіційний додаток Telegram на телефоні чи комп'ютері. Після введення отриманого коду в консоль сесія зберігається як файл із розширенням **.session** у папці з Вашим проєктом Python. Якщо не чіпати цей файл, у подальшому клієнт у програмі має запускатись і працювати за збереженою сесією відразу, без жодних додаткових кодів підтвердження.

Щодо доступу до даних через Telegram API, слід розуміти, що основним об'єктом є загальне поняття «сутність» (*Entity*). Сутностями вважаються як звичайні користувачі, так і групи (загальні чати) та канали. Для взаємодії з будь-якою сутністю до неї слід звернутись через її ID або нікнейм. Найчастіше ми не знаємо ID, тож тут корисною буде функція **get_entity(),** яка за нікнеймом витягує посилання безпосередньо на сутність.

Нехай у нас є публічний канал із нікнеймом «*nazva_kanalu*». Отримаємо та збережемо посилання на нього до змінної *channel_entity:*

```
channel_name = 'nazva_kanalu'
channel_entity = client.get_entity(channel_name)
```

Після цього ми можемо приступати до роботи з сутністю (в нашому випадку з каналом), використовуючи збережене у змінній посилання на цей об'єкт.

Кожна сутність має набір властивостей, які можна зчитати. Це зокрема:

- номер (**ID**);
- нікнейм (**username**);
- назва (**title**);

- дата створення (**date**);
- посилання на фото (**photo**);
- додаткові параметри залежно від типу сутності:
  - чи є група надгрупою (**megagroup**);
  - кількість учасників (**participants_count**);
  - чи є доступ обмеженим (**restricted**);
  - чи є профіль перевіреним (**verified**);
  - чи поточний користувач є адміністратором (**admin_rights**);
  - чи поточний користувач є автором групи (**creator**).

Окрім отримання загальної інформації про сутність, можна також зчитати повідомлення (публікації, пости) з діалогу, групи чи каналу. Для цього призначені дві основні функції:

- **iter_messages();**
- **get_messages().**

Обидві вони працюють дуже подібно, але друга дозволяє отримати загальну кількість повідомлень.

Зчитування розглянемо на прикладі з **iter_messages():**

```
msgs = client.iter_messages(channel_entity, limit=3)
```

Цей код візьме з уже зазначеної вище сутності 3 останні повідомлення та запише їх до змінної *msgs*. Якщо бажаєте зчитувати повідомлення в прямому хронологічному порядку, від давніх до нових, необхідно додати параметр **reverse.**

Щодо кількості повідомлень, слід зазначити, що параметр **limit** у цій функції є опціональним. Якщо його не зазначати, програма спробує зчитати всі повідомлення аж до першого в діалозі (чаті, каналі). Тим не менше, з технічної точки зору це затратно по ресурсах і нераціонально, тож Telegram усе одно обмежить такі спроби самостійно. Залежно від характеристик запиту та джерела даних, записи видаватимуться в кількості по 100 або 3000 за один раз.

Більш того, досвід показує, що навіть коли виходить зчитати повідомлення порціями по 200 (загалом більше 100), хронологічний порядок починає збиватися. І хоча результати й приходять у повному обсязі, але записуються в неправильній послідовності. Тож із практики можна порадити вичитувати за 1 запит максимум по 100 повідомлень або постів, що допоможе уникнути подібних проблем.

Після зчитування першої порції записів можна взяти наступну з потрібним зміщенням, задавши у функції параметр **add_offset** або інші додаткові аргументи.

Щодо функції **get_messages(),** її зручно використовувати для підрахунку загальної кількості повідомлень (постів), що міститься в діалозі (або на каналі):

```
msgs = client.get_messages(channel_entity)
print(msgs.total)
```

При цьому буде підраховано всі опубліковані пости чи наявні повідомлення незалежно від того, скільки саме ми зчитуємо зараз через **get_messages().** Тобто навіть якщо виставити ліміт у 3 (чи скільки завгодно), властивість **.total** покаже

загальну кількість. Це число може бути корисним для планування подальшого зчитування записів, а також для порівняння діалогів або каналів між собою й оцінювання їхньої активності.

Зберігши потрібну нам кількість повідомлень у змінну *msgs* чи довільну іншу, надалі можемо опрацьовувати отримані записи, проходячи по них циклом і зчитуючи для кожного з них текст і метадані (характеристики повідомлення). Наприклад, аби вивести на екран дату публікації та текст поста з каналу, можна використати наступний код:

```
for msg in msgs:
    print(msg.date)
    print(msg.message)
    print()
```

Тут для кожного повідомлення беруться його властивості **.date** та **.message,** які містять необхідну нам інформацію. Слід звернути увагу, що після кожного повідомлення варто робити відступ принаймні в один рядок задля того, аби потім було легше читати результати та бачити межі між різними записами.

Надалі можна не тільки показувати зчитані повідомлення в консолі, а й записувати їх до текстових або інших файлів чи опрацьовувати отримані тексти взагалі будь-яким довільним чином. Для цього можна записати вміст властивості **.message** до рядкової змінної.

Проте важливо, що далеко не всі повідомлення в діалогах, групах і на каналах будуть містити текст. Для деяких постів властивість **.message** буде являти собою порожній рядок — наприклад, якщо опубліковано лише картинку, відео чи інший файл без жодних підписів. Що гірше, деякі пости взагалі не міститимуть властивості **.message** як такої (її значення буде **null, None** тощо), адже деякі повідомлення є службовими: канал створено, перейменовано, змінено зображення профіля, додано чи видалено учасників і т.д. У таких випадках намагання зчитати чи скопіювати вміст **.message** призведе до помилки. Тож при завантаженні слід перевіряти наявність цієї властивості та в разі її відсутності пропускати подібні пости.

Як іще один приклад роботи з текстовими та іншими даними розглянемо соціальну мережу ***Instagram,*** яка є однією з найпопулярніших у світі та зокрема і в Україні. Ця система розроблялась іще з 2010 року, і нині належить компанії Meta (США). Станом на початок 2026, цей засіб мав близько 3 млрд. активних користувачів у світі щомісяця, з яких біля 500 млн. — щодня.

Серед особливостей *Instagram:*

- хмарне зберігання даних (що є нині стандартом для сучасних великих сервісів);
- кросплатформність (ним можна користуватись як через офіційний мобільний додаток під Android та iOS, так і настільний додаток для Windows 10+, а також вебверсію, що працює через браузер майже з будь-якого пристрою);
- інтерфейс понад 30 мовами світу;

- можливість автоматичного та безкоштовного перекладу текстів публікацій (що також сприяє популярності засобу в різних точках світу);
- можливість публікувати фото, відео, тексти, лишати коментарі, поширювати публікації інших, а також спілкуватись і ділитись медіафайлами у вбудованому месенджері;
- закритий код, однак наявні часткові аналоги з відкритим кодом, у яких обмін даними реалізовано з допомогою API.

Що стосується сторінок, або профілів, вони в *Instagram* є двох типів:

- публічні (доступні до перегляду для всіх без підписки на профіль і навіть без акаунту та/або авторизації в мережі);
- приватні (вимагають підписки та схвалення заявки власником).

Як інформацію з публічних сторінок, так і з приватних (за умови підписки) можна збирати шляхом створення власних додатків, які будуть отримувати доступ до *Instagram API.* Для цього можна або зареєструватись і отримати параметри доступу на офіційній сторінці для розробників, або скористатися бібліотеками для певної мови програмування, яких нині досить багато. Після цього можна буде працювати як із власним обліковим записом у соцмережі (профілем, підписками, особистими повідомленнями, збереженими постами тощо), так і з інформацією з інших сторінок, до перегляду яких ми маємо доступ.

Серед можливостей взаємодії з *Instagram* слід виділити наступні:

- отримання загальної інформації про профіль (опис тощо);
- отримання доступу до постів (кількість, посилання на них і т.д.)
- скачування публікацій (файли зображень, відео, текстів підписів у форматі .txt і метаданих постів у форматі JSON).

Із докладною інформацією про версії *Instagram API* та їхні можливості можна ознайомитись на офіційному ресурсі для розробників за посиланням: **https://developers.facebook.com/products/instagram/apis.**

Саме завдяки офіційному API за останні роки було розроблено безліч сторонніх засобів, що дозволяють працювати з соцмережею поза офіційними додатками та вебверсією. Це різноманітні консольні програми, мобільні додатки та розширення для браузерів, у яких функції *Instagram* або розширені (додано можливість скачувати публікації та історії, запланувати та автоматизувати власні пости, аналізувати статистику взаємодії тощо), або ж частково урізані (відфільтровано стрічку новин; усунуто рекламу; прибрано деякі функції заради економії пам’яті і т.д.).

Щодо роботи з API через сучасні мови програмування, то для цього теж уже створено багато зручних інструментів. Це спеціальні бібліотеки та модулі для Python, C# та інших мов. Зокрема для Python популярною бібліотекою є ***Instaloader,*** що розробляється з 2016 року. Станом на весну 2026 р. найновішою версією цього модуля є 4.15.1. Поширеною та зручною альтернативою нині є також ***Instagrapi,*** перші версії якої почали виходити в 2020 році.

Слід зазначити, що залежно від сервісу, доступ до якого ми хочемо отримати через API, процес налаштування та підготовки до роботи може бути простішим або складнішим, причому досить суттєво.

Якщо взяти за зразок соцмережу ***X [Twitter],*** то це один із прикладів складної реалізації. По-перше, тут обов'язково потрібна реєстрація та створення облікового запису. По-друге, при цьому треба не лише відповісти на довгий список запитань, а й підтвердити свою адресу електронної пошти та навіть телефонний номер. Більш того, якщо ви плануєте передавати вміст скачаних постів або похідну інформацію будь-якій державній установі або організації, пов'язаній із нею (наприклад, якщо ви розробляєте додаток для ДСНС чи іншої служби), вам доведеться не один тиждень листуватися з адміністрацією щодо призначення й особливостей вашого додатку, адже ця соцмережа дуже неохоче ділиться своїми даними. Крім цього, на початок 2026 р. функції скачування текстів та іншого були доступні лише у платних підписках, тоді як безкоштовна надавала змогу лише автоматизувати публікацію власних постів.

Що стосується інших соцмереж, месенджерів і сайтів зі статистичною інформацією, часто процедура реєстрації теж обов'язкова, проте вона є значно простішою. Наприклад, вимагається лише заповнити коротку форму розробника з декількома запитаннями (назва й тип додатку, сфера використання, іноді також короткий опис та галочка згоди з умовами надання послуг). Після цього перевірка даних проходить автоматично, і на електронну пошту або миттєво, або протягом пари годин приходить лист із параметрами доступу до API (це може бути Id та пароль, хеш або токен). Іноді доводиться зачекати від кількох хвилин до однієї доби для активації доступу, але зазвичай усе працює відразу після реєстрації.

Також більшість API є безкоштовними, хоча тут є чимало винятків, особливо коли вам необхідно викачувати значні обсяги даних або робити велику кількість запитів на секунду, хвилину, годину, добу і т.д. Це передусім стосується випадків, коли створювані вами додатки будуть витягувати дані з певного сервісу не лише для вас у рамках особистого використання, а для багатьох кінцевих користувачів. Скажімо, обмеження в 60 запитів на хвилину (з безкоштовної підписки сайту OpenWeather), яких цілком достатньо при експериментах із API для навчання чи особистих цілей, можуть стати вже суттєвою перешкодою для безперебійної роботи мобільного додатку, розрахованого на щоденний показ погодних умов для десятків тисяч людей.

Повертаючись до соцмережі *Instagram* і бібліотек *Instaloader, Instagrapi* та інших аналогів, це навпаки, один із найпростіших шляхів отримання даних через API. Адже ним можливо скористатись не лише без реєстрації акаунту розробника, а й узагалі без облікового запису в самій соцмережі, тобто анонімно.

Тим не менше, перед спробами встановлення з'єднання та роботи з даними варто наголосити на системі безпеки *Instagram* та інших популярних вебсервісів. Досить часто це викликає труднощі, із якими можуть стикнутися розробники через недотримання певних базових правил роботи. Річ у тому, що задля безпеки самих користувачів і серверів будь-яка підозріла активність як людей, так і додатків, що працюють через API, призводить до блокування облікового запису, з якого вона була зафіксована. Зокрема Вам можуть обмежити доступ до акаунту

від кількох хвилин і до декількох діб у випадку т. зв. переповнення (*FloodWaitError*).

Таке може статися, серед іншого, при багатократній невдалій авторизації (неправильно введено пароль, код логіну тощо) або при зміні параметрів сесії. Під зміною параметрів розуміється як зміна IP-адреси, з якої відбувається вхід, так і зміна пристрою, що може відстежуватись через MAC-адресу. Тож очевидно, що якщо заходити одночасно чи з невеликим інтервалом із різних пристроїв (телефон, планшет, ноутбук, настільний комп'ютер) або адрес (через кабельний інтернет, Wi-Fi, інтернет від мобільного оператора, а також із використанням VPN та без), це може трактуватись як підозріла активність і розцінюватись як спроба зламу вашого облікового запису зловмисниками. Отже, при тестуванні додатків слід пам'ятати про можливі обмеження, уважно вводити свої дані доступу, а також не забувати підтверджувати свою активність із інших пристроїв у разі запитів через офіційний додаток *Instagram*.

Розглянемо приклад простого запиту до публічного профіля в мережі з використанням мови Python і бібліотеки *Instaloader:*

```
import instaloader

loader = instaloader.Instaloader()
user = '...' # username

profile = instaloader.Profile.from_username
(loader.context, user)
print('Username:', profile.username)
print('User ID:', profile.userid)
print('Bio:', profile.biography)
print('Media count:', profile.mediacount)
print('Private profile:', profile.is_private)
```

У цьому фрагменті ми імпортуємо бібліотеку, створюємо клієнта *loader* для завантаження даних, обираємо профіль *user*, який нас цікавить, і отримуємо посилання безпосередньо на сторінку цього користувача у змінній *profile*. Після цього можна звертатись до змінної, вичитуючи з неї потрібні нам атрибути.

Тут слід сказати, що кожен профіль у *Instagram* має набір властивостей:

- номер (**userid**);
- нікнейм (**username**), видимий як назва та адреса сторінки;
- опис (**biography**);
- режим доступу (**is_private**);
- кількість постів (**mediacount**);
- посилання на фото;
- підписники (**followers**);
- підписки (**followees**);
- та інше.

Вищенаведений код вичитає 5 зазначених властивостей зі сторінки за назвою, збереженою у змінній *user*, і покаже їх у консолі. Підписники та підписки

доступні лише після авторизації в мережі, а при подібному анонімному запиті доступ до цієї інформації буде обмежено.

Наступним кроком спробуємо отримати доступ до постів — що також можливо анонімно, поки ми працюємо з публічними профілями. Наприклад:

```
print('Posts:\n')
posts = profile.get_posts()
for post in posts:
      print('Date:', post.date_utc)
      print('Caption:', post.caption)
      print('Media count:', post.mediacount)
      print()
```

Цей код спершу отримує посилання на всі пости сторінки у змінну *posts*, а далі проходить по них циклом *for* і для кожного пише в консолі дату публікації, текст (властивість **caption**) і кількість медіафайлів у цьому пості. Між постами продруковується один порожній рядок як розділювач для зручності читання.

Важливо зазначити: якщо читати публікації таким чином, то при отриманні даних не проводиться жодна фільтрація постів. Тож навіть коли певна сторінка має тисячу чи більше опублікованих записів, цикл *for* пройде по них усіх без винятку. Це не завжди добре, адже:

- зазвичай це зайві дані, які нам не потрібні;
- це може займати деякий час і витрачати наш трафік;
- при перевищенні певних лімітів даних нас можуть заблокувати.

Конкретно з методом **get_posts()** це не так критично, як із деякими іншими, адже він лише отримує перелік постів і метаданих, не скачуючи їх. Тим не менше, було б корисно мати змогу обмежити кількість скачаних постів довільним чином.

Однак незручність полягає в тому, що звичайне звернення до окремих елементів за індексом через **[n]** і розрізання (англ. *slicing*) через **[m:n]**, яке доступне в Python для структур даних на кшталт списків, тут не працює. Річ у тому, що отриманий перелік постів є не списком, а об’єктом типу **NodeIterator,** елементи якого можна проходити лише покроково від першого до останнього по одному за раз. Тож звернення за індексом призводить до помилки типів даних.

Розв’язати цю проблему можна за рахунок використання допоміжних засобів із бібліотеки ***islice.*** Розглянемо три шляхи виходу з ситуації:

```
from itertools import islice

limit = 3

for post in islice(posts, 0, limit):
      print(post.date_utc)
      print(post.caption)
# АБО:
for index, post in zip(range(limit), posts):
      print('Post no.', index+1)
      print(post.date_utc)
```

```
        print(post.caption)
# АБО:
for index, post in enumerate(islice(posts, limit)):
        print('Post no.', index+1)
        print(post.date_utc)
        print(post.caption)
```

Другий і третій варіанти дозволяють не лише обмежити числом *limit* кількість постів, по якій ми проходимо, а й за потреби пронумерувати їх.

Як бачимо, навіть без скачування самих постів ми маємо доступ до їхніх текстів, дат і часу публікації. Зберігши потрібну нам кількість повідомлень у рядкову змінну, текстовий файл чи будь-куди інде, надалі можна опрацьовувати отримані тексти довільним чином, наприклад, витягуючи з них потрібну числову чи іншу інформацію, проводячи інтелектуальний аналіз даних тощо.

Важливо, що при збиранні текстової інформації з API, зокрема при роботі з месенджерами, далеко не всі повідомлення в діалогах, групах і на каналах будуть містити текст. Наприклад, для деяких повідомлень властивість **.message** (або аналогічна залежно від джерела та API) може являти собою порожній рядок — наприклад, якщо в діалозі чи на каналі було розміщено лише картинку, відео чи інший файл без жодних підписів. Що гірше, деякі повідомлення та пости взагалі не містять властивості **.message** як такої (при витягуванні даних її значення буде **null, None** тощо), адже деякі повідомлення є службовими: канал створено, перейменовано, змінено зображення профіля, додано чи видалено учасників і т.д. У таких випадках намагання зчитати чи скопіювати вміст **.message** призведе до помилки. Тож при завантаженні слід перевіряти наявність цієї властивості та в разі її відсутності пропускати подібні повідомлення.

Що ж до завантажування медіафайлів через бібліотеку *Instaloader*, тут нам доступні такі варіанти. По-перше, можна скачати лише зображення профілю та метадані (опис сторінки) у форматі JSON таким чином:

```
loader.download_profile(user, profile_pic_only=True)
```

При цьому, якщо задати значення другого параметра рівним «*False*», то така функція скачає відразу всі пости без жодних обмежень, аналогічно до **get_posts().** Але й тут так само є змога відфільтрувати завантажувані дані. Це навіть простіше, адже функція **download_profile()** має ще один вбудований опціональний параметр **post_filter.** Сюди можна прописати обмеження на пости, наприклад, за датою їх публікації:

```
import datetime

loader.filename_pattern = '{owner_username}_{date_utc}_UTC'
filter = lambda post: post.date_utc >= datetime.datetime
(2026, 5, 1)
profile = loader.download_profile(user, post_filter=filter)
```

Цей фрагмент коду спершу задає шаблон для імен файлів, які буде скачано в папку з нашим проєктом, далі визначає умови фільтру — пости не раніше 1 травня 2026 року, а потім застосовує цей фільтр до завантаження даних профіля.

Ще раз звернемо увагу, що всі вищенаведені дії проводились нами без реєстрації нашого додатку і без логіну під власним акаунтом. Тож багато можливостей по роботі з даними доступні не лише анонімно, а й навіть за відсутності будь-якого облікового запису в *Instagram.*

Однак у випадку роботи з власним профілем, для доступу до закритих сторінок (де вимагається підписка), а також задля зменшення ймовірності бану й підвищення лімітів на скачування даних буває корисно авторизуватись. У цьому разі вам знадобиться реєстрація в *Instagram.* Маючи обліковий запис, можна доповнити вищенаведений код, вставивши на початку наступне:

```
me = '...'
pw = '...'
loader.login(me, pw)
```

Тепер за рахунок змінної *me*, яка повинна містити ваш нікнейм, і *pw* (відповідно пароль) клієнт не просто запитуватиме доступ до даних, а робитиме це з-під вашого профіля. Це розширить набір доступних функцій, а також надасть права на роботу із закритими сторінками, на які ви підписані.

Література

Див. джерела №№ 6, 8.

ЛАБОРАТОРНА РОБОТА 5.

Автоматична генерація тексту та створення чатботів

Автоматична генерація текстів є важливим завданням у багатьох сферах сучасного життя. Класичним підходом до розв'язання цієї задачі є використання ланцюгів Маркова і похідних від нього моделей.

Ланцюг Маркова, запропонований математиком А. Марковим іще в 1907–13 рр. як «ланцюг залежних подій» — це ймовірнісна модель, яка описує послідовність можливих подій. При цьому будується орієнтований граф взаємозв'язків, де вузлами є можливі події (стани), а ребрами — ймовірність переходу між ними. Сума ймовірностей повинна дорівнювати одиниці (100%) для всіх виходів із кожного вузла. Граф може містити петлі, тобто ребра, що сполучають вузол із самим собою.

Події, які моделюються ланцюгами Маркова, вважаються випадковими, а кожна подія в них залежить від попередньої. Саме в цьому полягає ключова відмінність такого підходу від схеми Бернуллі чи наївного Баєсового алгоритму, які нехтують можливими взаємозалежностями заради спрощення моделі.

Залежно від того, чи враховується лише один попередній стан (подія), чи більше, виділяють відповідно моделі 1-го порядку, 2-го та інших. Чим більшим є порядок *N* моделі Маркова, тим точнішими є її результати, але так само зростає і обчислювальна складність. Отже, для розв'язання завдань традиційно доводиться шукати баланс між простотою моделі та її точністю.

У лінгвістиці ланцюги Маркова активно використовуються для завдань статистичного машинного перекладу. Зокрема на них базуються алгоритми Google Translate і його аналогів. Також ланцюги Маркова застосовують для виправлення помилок і автоматичної генерації тексту.

Окремо слід сказати про виправлення помилок, можливе за допомогою цієї моделі. Річ у тому, що часом усі слова в реченні можуть бути написані правильно, проте в цілому речення є некоректним. Це відбувається через граматичні помилки або одруки, внаслідок яких одне слово перетворюється на інше, теж коректне, але яке не підходить за контекстом. Наприклад: «*I need to notified the bank of this problem*» (замість «*notified*» тут мало бути «*notify*»). Підходи на кшталт відстані редагування тут не допоможуть, адже вони опрацьовують кожне слово окремо. І лише моделі Маркова, що враховують контекст і залежності між попереднім і наступним словом, здатні виявити помилки такого роду.

Щодо генерації текстів, завдання може стояти наступним чином. Є корпус текстів певного автора або текстів, написаних у певному стилі. Необхідно згенерувати подібний текст на основі наявних даних.

Для реалізації можна спершу проаналізувати залежності між сусідніми словами в наявних текстах. Якщо в них неодноразово зустрічається слово «*hi*», і в 30% випадків за ним іде слово «*everyone*», в 20% — «*there*», а в 50% — крапка, це можна подати у вигляді наступної таблиці:

Табл. 1. *Залежності для слова «hi»*

	everyone	**there**	**.**
hi	0,3	0,2	0,5

Якщо ми надалі проаналізуємо, що йде за словами «*everyone*», «*there*» тощо, граф буде розширюватись, а в певні моменти може також зациклитись, адже після слова «*everyone*» може йти «*says*», а після «*says*» — «*hi*», з якого ми почали аналіз. У будь-якому разі, обсяг інформації в таблиці буде також збільшуватись. Аби відобразити в ній усі можливі пари слів (оскільки теоретично після будь-якого одного слова може йти будь-яке інше), нам доведеться сформувати квадратну двовимірну матрицю переходів («*transition matrix*»). У ній кількість рядків буде дорівнювати кількості стовпців, і в них міститиметься перелік унікальних слів, які зустрічаються в заданому наборі текстів. Натомість кожна комірка всередині зберігатиме в собі ймовірність p_{ij} переходу від слова i до слова j. Ось як може виглядати приклад такої матриці переходів для набору з 9 унікальних слів:

Табл. 2. *Матриця переходів для набору з 9 унікальних слів*

0,2	0	0,1	0	0,1	0,1	0	0,3	0,2
1	0	0	0	0	0	0	0	0
0,5	0	0,1	0,3	0	0,1	0	0	0
0	0	0,4	0,2	0,4	0	0	0	0
0	0	0	0,25	0,25	0,5	0	0	0
0,2	0,1	0	0,1	0	0,2	0,2	0,1	0,1
0	0	0,2	0	0	0,2	0	0,6	0
0,2	0	0,2	0,4	0	0	0	0,2	0
0	0	0,75	0	0	0,25	0	0	0

Як бачимо, сума ймовірностей по кожному рядку дорівнює 1, як цього й вимагає модель Маркова. Використовуючи цю матрицю переходів, у подальшому можливо генерувати текст згідно з виявленими в ній залежностями. Тобто якщо в реальному наборі текстів після слова i з імовірністю p_{ij} йде слово j, так само часто будемо ставити його після i наступним і у згенерованих нами текстах.

Перейдемо до одного з можливих варіантів реалізації алгоритму генерації тексту на основі ланцюгів Маркова за допомогою мови Python. Спершу створимо словник, у якому ключами будуть поточні стани (слово i), а значеннями — наступні стани (слово j). Далі напишемо функцію генерації наступного стану із заданих альтернатив за ймовірностями, зазначеними у цьому словнику.

Ось як може виглядати функція, що створює модель Маркова за поданими текстами:

```
from collections import defaultdict

def markov_chain(text):
    words = text.split()
    my_dict = defaultdict(list)
```

```
    for current_word, next_word in zip(words[0:-1], words[1:]):
        my_dict[current_word].append(next_word)
    my_dict = dict(my_dict)
    return my_dict
```

Далі перевіримо роботу цієї функції на певному короткому тексті *test_text:*

```
test_dict = markov_chain(test_text)
for word in test_dict:
    print(word, test_dict[word])
```

Після такого виклику функції для поданого тексту буде згенеровано ланцюги Маркова, а обчислені залежності виведуться на екран. Результат може бути наступним (тут наведено лише фрагмент виведення):

```
I ['am', 'am.', 'do']
Hi ['everyone.', 'everybody.', 'there!']
Everyone ['says', 'says']
```

Отже, для кожного слова було збережено всі можливі слова-наступники. Зверніть увагу, що оскільки розбиття тексту на слова проводилося за пробілами через звичайну функцію **.split(),** то всі розділові знаки, що йшли після слів, лишилися при них і у словнику. Це може бути як недоліком, так і перевагою.

Незручністю такого підходу є те, що якщо нам не потрібна пунктуація з оригінальних текстів, її доведеться вичищати окремо. Також у нас дублюються ключі з усіма можливими розділовими знаками після них.

Натомість із іншого боку це не заважає нам генерувати тексти за такими залежностями, а навіть навпаки: після крапки чи знаку оклику завжди буде автоматично генеруватися слово з великої літери, а після коми чи пробілу — з малої, адже саме так і було в реальних текстах.

Також цікаво, що серед значень є взагалі повні дублі (наприклад, «says»). Так сталось через те, що в алгоритмі ми лише доповнюємо значення новими через функцію **.append(),** не перевіряючи, чи такі вже були записані. І це теж можна оцінити як негативно, так і позитивно.

Недоліком є те, що наявність таких дублів марнує ресурси, зокрема пам’ять для їх збереження, а також надалі витрачає час на проходження довшого списку значень.

Однак перевага полягає в тому, що за рахунок запису всіх без винятків слів-наступників у нас зберігається повна статистика щодо частоти кожного випадку. Справді, якщо в подальшому генерувати для ключа наступне слово, беручи варіанти зі списку значень випадковим чином, алгоритм потраплятиме на той чи інший варіант пропорційно до оригінальної статистики. Якщо після слова *i* 9 разів зустрілось j_1 і лише 1 раз — j_2, то й генератор у 90% випадків поставить після *i* саме j_1, і т.д.

Таким чином, при цьому підході ми не надто раціонально використовуємо пам’ять через наявність дублів, однак зате не мусимо обчислювати та зберігати

матрицю переходів, на що теж був би потрібен час. При цьому більшість комірок у таких матрицях у будь-якому разі містять нулі, адже після кожного конкретного слова *i* в реальності може зустрітись далеко не кожне слово *j*.

Перейдемо до останнього етапу роботи — власне генерації речень. Вона може бути реалізована, наприклад, так:

```
import random

def generate_sentence(chain, word_count):
    cur_word = random.choice(list(chain.keys()))
    sentence = cur_word.capitalize()
    for i in range(word_count-1):
        next_word = random.choice(chain[cur_word])
        sentence += ' ' + next_word
        cur_word = next_word
    sentence += '.'
    return sentence
```

Функція приймає на вхід вищеописаний словник, що містить ланцюги Маркова (аргумент *chain*), а також необхідну кількість слів для генерації речення (*word_count*). На початку ми випадковим чином беремо перше слово речення з ключів словника та пишемо його з великої літери.

Наступне слово береться теж випадковим чином уже зі значень у словнику для цього ключа. Між словами ставиться пробіл, а наступне слово приймається за поточне, після чого цей цикл повторюється стільки разів, щоби речення містило потрібну нам кількість слів. Речення завершується крапкою та повертається на вихід функції.

Перевіримо роботу функції на ланцюгах Маркова, отриманих у прикладі вище:

```
print(generate_sentence(test_dict, 8))
```

Результат може виглядати так:

```
Everything is going on here? Tell me. I.
```

Як видно, з розділовими знаками тут усе в порядку через особливості розбиття речення на слова, описані вище. Також, якщо проаналізувати кожну пару слів, вона є цілком схожою на те, що можна зустріти в реальному мовленні.

Тим не менше, щойно ми почнемо дивитися відразу на три чи більше сусідні слова, виявиться, що результат є вже не таким коректним і правдоподібним. Причина криється в тому, що ми використали модель Маркова 1-го порядку, яка враховує лише зв’язки між парами суміжних слів, але не більше. Лише при використанні складніших моделей нам вдалося б отримати результат, більш наближений до реальних текстів, написаних людьми.

Можна частково покращити генерацію речень за рахунок використання більших обсягів текстів, які приймаються на вхід при створенні моделі. Скажімо, якщо ми візьмемо текстовий файл із книгою «Аліса у Дивокраї» та побудуємо ланцюги Маркова для нього, результат може вийти, наприклад, таким:

```
Duchess! Oh my dear! I like a telescope.'
```

Як бачимо, лексика стала багатшою, можна зустріти деякі мовні звороти, однак також можливі й деякі граматичні помилки, не кажучи про відсутність послідовності в пунктуації тощо.

Таким чином, вхідний текст і його обсяг впливає на отримані результати, проте, на жаль, не знімає фундаментальних обмежень, викликаних вимушеною спрощеністю нашої моделі.

Тим часом нині генерація текстів може бути значно якіснішою з LLM.

LLM (від англ. *large language models* — великі мовні моделі) — це моделі мови на основі нейромереж, навчені на великих обсягах текстової інформації.

GPT (від англ. *generative pre-trained transformer* — генеративний попередньо навчений трансформер) — набір мовних моделей, які розроблялись компанією OpenAI [9] із 2015 р. і випускались із 2018 р. У 2022 р. було запущено чатбот ***ChatGPT*** на основі моделі GPT 3.5. Цей тип мовних моделей використовує глибинне навчання (англ. *deep learning*) та має архітектуру типу «трансформер».

Модель GPT початково навчається на корпусах тексту великого обсягу. Її основна мета — навчитись передбачати наступне слово на основі попередніх слів у тексті. Цим вона подібна на модель Маркова, однак ланцюги Маркова зазвичай обмежені лише невеликою кількістю попередніх слів.

Тим часом GPT (залежно від версії) може враховувати контекст і в кілька тисяч, і навіть до 1 млн. попередніх слів. Оскільки зазвичай вебсторінки, книги та інші джерела є коротшими, то фактично цієї кількості слів достатньо, аби покрити взагалі весь обсяг тексту і врахувати контекст усього документа повністю.

Історично кожна наступна версія GPT у рази, а іноді й на порядки перевищувала попередні за кількістю параметрів і обсягом тексту для навчання. Для порівняння наведемо лише деякі доступні у відкритих джерелах показники:

- GPT-2 (2019) — 1,5 млрд. параметрів;
- GPT-3 (2020) — 175 млрд. параметрів;
- GPT-4 (2023) — 1,8 трлн. параметрів;
- GPT-5 (2025) — від 5 до 8 трлн. параметрів (за різними оцінками).

За обсягами даних при навчанні:

- GPT-2 (2019) — 40 ГБ тексту (8 млн. вебсторінок);
- GPT-3 (2020) — 45 ТБ тексту.

У більш сучасних версіях акцент змістився від масштабування обсягу даних і збільшення кількості параметрів до вдосконалення самої архітектури системи, тож роль кількісних показників стала не такою важливою для якості роботи.

Серед особливостей і переваг моделі GPT слід відзначити наступні.

1. Набір даних дуже різноманітний (при навчанні модель тренується на вебджерелах абсолютно різної тематики та стилів), завдяки чому GPT може

«спілкуватись» на довільні теми, імітуючи різні стилі мовлення та інші особливості, притаманні реальним текстам природною мовою.

2. Модель може сама підлаштовуватися під зміст і стиль заданого тексту. Це теж можливо завдяки обсягу та розмаїтості вхідних даних.

3. Обсяги даних також дозволяють GPT показувати кращі результати за інші, спеціальні моделі, створені для окремих вузьких галузей. Хоча конкуренти могли мати багато вбудованих правил і тривалу історію розроблення й навчання на текстах конкретної галузі знань, GPT знову виграє за рахунок обсягів даних. Їх настільки багато, що навіть 1% може значно перевищувати весь розмір окремої моделі, яка наповнювалася текстами, відібраними чітко за вузькою сферою. Крім того, за рахунок загального обсягу GPT в цілому краще будує речення та відповідає на майже будь-які запитання.

4. Застосування машинного навчання не вимагає спеціальних правил чи моделей для різних завдань або галузей знань. При тренуванні моделі достатньо мати «сирі» дані, на яких вона вчиться сама, використовуючи методи штучного інтелекту. Тож і досягти ліпших результатів у певному сенсі простіше.

Разом із тим, моделі GPT мають і свої недоліки:

- у згенерованих текстах іноді зустрічаються повтори;
- відповіді можуть бути недосконалими і навіть нелогічними та суперечливими, адже в GPT немає моделі світу як такої, а лише модель мови, яка дозволяє генерувати тексти, подібні на справжні;
- GPT може швидко і недоречно перемикатись між різними темами;
- поки що не всі теми достатньо добре репрезентовані в наборах даних, на яких тренуються моделі, тож якість відповідей у цих сферах може бути гіршою.

Станом на зараз GPT використовується вже не лише для генерації (синтезу) текстів або спілкування в діалозі з користувачами, а й для аналізу текстів і пошуку відповідей на запитання, виведення логічних висновків із тексту (англ. *RTE — recognizing text entailments*), анотування та реферування документів тощо. Нині GPT також може зробити машинний переклад між різними природними мовами, що раніше теж вимагало створення спеціальних систем і моделей. Крім того, GPT уже вміє й генерувати зображення, відео, різні типи документів і т.д.

Одним із традиційних показників, за якими ще донедавна часто оцінювали роботу подібних моделей, є якість розуміння тексту. Зокрема GPT і аналоги тестували на наборі даних *TriviaQA,* який містить сотні тисяч питань і відповідей на різні теми. Якщо модель GPT-3 показувала в таких завданнях точність 60–70%, то якість GPT-4 уже зросла до 75–85%. І це вже цілком порівнювано з рівнем розуміння тексту людьми, який може складати близько 75–90%.

Тим не менше, станом на зараз результати виконання завдань по *TriviaQA* уже не вважаються таким важливим мірилом якості мовних моделей. Передусім це сталось через витоки даних: сучасні моделі вже бачили багато з тестових завдань, що лежать у відкритому доступі, і могли вчитися на них.

Також моделі можуть просто запам’ятовувати певні факти, що часто зустрічаються в текстах, у дослівному вигляді. Тож при відповіді на запитання неможливо точно визначити, чи модель обчислила відповідь і зробила висновок

на основі певної логіки, чи просто відтворила її, скопіювавши потрібний текст. Крім того, якість сучасних моделей часто відрізняється вже лише на 1–2%, і їх стає все важче порівнювати.

У будь-якому разі, нині моделі GPT і аналоги стали частиною нашого повсякденного життя, отримавши інтерфейси для спілкування зі звичайними користувачами з усього світу. Зокрема серед найбільш поширених ШІ-помічників на основі великих мовних моделей можна згадати:

- ChatGPT (від компанії OpenAI);
- Gemini (Google);
- Claude (Anthropic);
- Copilot (Microsoft);
- Grok (X / Twitter);
- DeepSeek (DeepSeek).

Частково використання ШІ та LLM впроваджено і в голосові помічники:

- Siri (Apple) — із застосуванням Apple Intelligence;
- Alexa (Amazon) — планується використання напрацювань OpenAI.

При всіх перевагах, які надають подібні засоби і ШІ в цілому, слід також пам’ятати, що використовувати їх слід свідомо, як і будь-які інші інструменти. Тож в останні роки виші та інші організації стали розробляти та впроваджувати правила та рекомендації щодо використання ШІ в освітній та іншій діяльності.

Зокрема в багатьох українських університетах було затверджено політику використання штучного інтелекту. Серед основних порад і правил можна виділити наступні важливі моменти.

1. ШІ повинен бути допоміжним, а не єдиним чи основним засобом при навчанні та викладанні. Здобувач навчається самостійно, але може скористатися ШІ для спрощення пошуку інформації, автоматизації рутинних завдань тощо.

2. Коли ми використовуємо ШІ, завжди слід відкрито зазначати це, а також уточнювати, що саме з отриманих результатів було власною роботою, а де нам допомогли ШІ-засоби.

3. Будь-яка інформація, отримана від ШІ, повинна перевірятись, а наше ставлення до наданих результатів має бути критичним. Адже кожен ШІ-засіб має властивість часом помилятись, «галюцинувати», наводячи вигадані дані, які або відсутні у справжніх джерелах, або навіть прямо суперечать їм. Також дуже часто ШІ генерує фейкові назви статей і книг, які ніколи не публікувались. Тож треба бути особливо уважними при використанні ШІ для складання бібліографії тощо.

4. Самі запити, які ви ставите ШІ-помічникам, мають бути неупередженими. Якщо запитати в ШІ про переваги якоїсь технології чи підходу, він може надати безліч підтверджень вашій гіпотезі чи вподобанням, але не сказати нічого про слабкі місця, недоліки та потенційні загрози. Тож варто просити про об’єктивний розгляд будь-якого питання та аналізувати явища з різних боків.

Що стосується наукових журналів і матеріалів збірників конференцій, тут також зазвичай діють певні правила при публікації текстів. Сучасні наукові видання часто вимагають від авторів заповнити так звану декларацію щодо використання ШІ в публікованому дослідженні.

У такій декларації автори мають зазначити, які саме засоби було ними застосовано при підготовці окремих частин матеріалу — написанні чи перевірці тексту, генерації зображень, проведенні розрахунків тощо. Також перед остаточною публікацією науковці повинні самостійно перевірити всі ці результати, адже саме автори несуть відповідальність за кінцевий матеріал.

Література

Див. джерела №№ 1, 2, 3, 6, 7, 9, 10.

ЛАБОРАТОРНА РОБОТА 6.

Автоматичне реферування тексту та пошук логічних зв'язків

Реферування та анотування текстів використовуються для їх скорочення та опису.

Реферат — це текст із одного чи багатьох джерел, у якому виклад зазвичай ведеться від першої особи (як і в оригінальних текстах). Обсяг реферату, залежно від мети укладання, може складати від 1–2 і до кількох десятків сторінок, але це завжди стислий, скорочений варіант відносно оригіналу. Реферат створюється для швидшого ознайомлення читачів (або слухачів) зі змістом об'ємного джерела чи ряду джерел із певної теми.

Анотація натомість зазвичай має обсяг від пари абзаців до 1–2 сторінок. Вона відрізняється від реферату тим, що подає інформацію ще більш стисло, часто пишеться від третьої особи (наприклад, «автор описує...»), а також може містити метадані — опис кількісних і якісних характеристик тексту (кількість сторінок чи аркушів, таблиць, зображень, додатків, джерел). У більшості випадків анотація також містить ключові слова, іноді — додаткову інформацію про автора роботи.

Анотація є обов'язковою для більшості наукових досліджень і академічних робіт (статті, тези доповідей, дисертації, дипломні та курсові роботи), має чітко визначену структуру та послідовність і укладається згідно з офіційними вимогами до оформлення. Анотації також розміщуються разом із текстом оригіналу в наукових репозиторіях та індексуються для автоматичного пошуку за ключовими словами. Для наукових статей анотація часто наявна не лише мовою оригіналу тексту, а й іншими (наприклад, англійською). Крім того, анотація може бути вільно доступною для перегляду навіть у статтях, повні тексти яких є закритими (розміщеними в платних журналах).

Для автоматичного реферування нині можна використати різні технології та підходи. Розглянемо один із них, який скорочуватиме текст шляхом ранжування речень першоджерела за їхньою важливістю (значимістю в тексті). Більш значущі речення будуть залишені в рефераті у своєму оригінальному вигляді. Натомість ті, які не сильно стосуються теми, будуть відфільтровані та не потраплять у реферат.

Загальний алгоритм такого підходу складається з наступних 5 кроків.

1. Обчислення частотності всіх слів у початковому тексті.

2. Токенізація тексту з розбиттям на речення.

3. Оцінювання речень за важливістю.

4. Визначення порогу, за яким речення будуть фільтруватись.

5. Власне генерація реферату за заданими попередньо даними.

Для реалізації **<u>першого кроку</u>** код може бути приблизно таким:

```
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
```

```
def freq_table(text_string) -> dict:
    stopWords = set(stopwords.words('english'))
    words = word_tokenize(text_string)
    ps = PorterStemmer()
    fr_table = dict()
    for word in words:
        word = ps.stem(word)
        if word in stopWords:
            continue
        if word in fr_table:
            fr_table[word] += 1
        else:
            fr_table[word] = 1
    return fr_table
```

У цьому зразку було застосовано так звані стоп-слова, токенізацію тексту за словами та стемер Портера для англійської мови.

Стоп-слова — це список слів, які не несуть особливого значення в тексті (артиклі, займенники, допоміжні дієслова тощо), тож їх бажано відфільтрувати, лишивши тільки суттєві слова, дотичні до теми. Списки стоп-слів розробляються для різних мов, і для англійської (як у нашому прикладі) можна скористатись відповідним набором **stopwords** із корпусів бібліотеки NLTK.

Токенізація — це процес розбиття тексту на складові частини, зазвичай слова або речення. Слід зауважити, що для поділу на слова можна скористатись також функцією **.split()**. Однак вона лише розбиває текст за пробілами, при цьому розділові знаки лишаються поруч зі словами, після яких вони йдуть.

Натомість токенізатор відділяє знаки пунктуації від слів. Тим не менше, він вважає їх рівноцінними токенами (елементами) до слів. Тож якщо вам не треба аналізувати та рахувати місця та кількість входжень розділових знаків, то надалі їх слід видалити з отриманого списку токенів.

Що ж до **стемера** (від англ. *stem* — основа слова) — це інструмент, який зводить усі похідні слова та словоформи з різними закінченнями (незалежно від частини мови) до однієї основи чи псевдооснови. Завдяки цьому механізму можна обчислити, скільки всього разів згадується якесь поняття та його похідні в тексті, навіть якщо вони зустрічаються в різних формах. Наприклад, усі входження слів *know, knows, known, knowing, knowledge* тощо будуть накопичені та зараховані до однієї основи — *know*. Це більш примітивний підхід, ніж семантичний аналіз, однак він дозволяє досягти прийнятних результатів простіше та швидше.

Викличемо вищенаведену функцію *freq_table*() і проаналізуємо результати опрацювання нею простого тестового речення: «*How do you do, miss Cockatoo?*»

```
test_text = 'How do you do, miss Cockatoo?'
print(freq_table(test_text))
```

Словник, згенерований функцією для цього речення, виглядатиме так:

```
{'cockatoo': 1, 'miss': 1, ',': 1, '?': 1}
```

На цьому прикладі можна помітити такі особливості застосованого підходу:

- усі слова автоматично зведені до малої літери, аби позиція на початку чи в середині речення не впливали на підрахунки;
- слова *how, do, you* були вилучені, оскільки всі вони перебувають у списку стоп-слів. Як бачимо, часом таких слів у реченні може бути більше, ніж суттєвих;
- поряд зі словами до словника додались також розділові знаки, адже після токенізації ми не очистили від них отриманий список токенів.

Звісно, чим довшим буде вхідний текст, тим об'ємнішим і змістовнішим буде словник, повернутий цією функцією. Надалі для тестів використовуватимемо текст обсягом приблизно в 1 сторінку, а саме фрагмент (перші три абзаци) есе про мирні наміри роботів, написаного моделлю GPT-3 у 2020 році [10].

Розглянемо реалізацію **другого кроку.** Приклад коду:

```
from nltk import sent_tokenize

print(sent_tokenize(test_text))
```

Тут ми знову застосовуємо токенізацію до того ж тексту, однак тепер це вже токенізація за реченнями — **sent_tokenize().** Вона ділить текст на окремі речення, що потрібно нам для подальшого їх оцінювання, фільтрування та генерації з них реферату. Функція повертає Python-список із рядків-речень.

Можливий код для **третього кроку:**

```
def score_sentences(sentences, freq_table) -> dict:
    sent_value = dict()
    for sent in sentences:
        word_count_in_sentence = (len(word_tokenize(sent)))
        for wordValue in freq_table:
            if wordValue in sent.lower():
                if sent[:15] in sent_value:
                    sent_value[sent[:15]] += freq_table[wordValue]
                else:
                    sent_value[sent[:15]] = freq_table[wordValue]
        sent_value[sent[:15]] = sent_value[sent[:15]] /
                                word_count_in_sentence
    return sent_value
```

Ця функція бере на вхід список речень із другого кроку та частотний словник із першого кроку. Після цього вона створює словник, де якому кожному реченню тексту відповідає числове значення. Воно обчислюється шляхом додання частот слів, що входять до речення. Таким чином, якщо в деякому реченні йдеться про щось дійсно суттєве, дотичне до основної теми тексту, то ці слова з високими значеннями підвищать і загальну оцінку речення.

Варто зазначити, що оскільки на першому кроці ми не додавали до словника стоп-слова, то і при зустрічі в реченні вони будуть пропущені й не додадуть жодних балів до його оцінки.

Ще один суттєвий момент полягає в тому, що у вищенаведеному коді при аналізі та збереженні речень вони беруться не повністю, а усікаються до перших 15 символів. Звісно, це може спричинити певні неточності, якщо різні речення матимуть схожий початок у цьому діапазоні. Проте завдяки подібному усіченню заощаджується пам'ять, і ми не перевантажуємо її надлишковою інформацією.

Нарешті, також важливо не просто обчислювати сумарну кількість балів по кожному реченню, а й також враховувати його загальну довжину. Справді, якщо речення міститиме не надто суттєві слова, проте буде дуже довгим, то завдяки лиш одній своїй довжині воно зможе перевершити інші речення, ключові для тексту, проте короткі. Тож для уникнення таких ситуацій проводиться процедура нормалізації: після підрахунку суми балів ця оцінка ділиться на довжину речення у словах. Отримане таким чином середнє арифметичне об'єктивніше відображає справжню цінність речення відносно тексту в цілому.

Якщо наш тестовий текст називається *test_text*, то його послідовне опрацювання через описані функції кроків 1–3 може виглядати так:

```
ft = freq_table(test_text)
s = sent_tokenize(test_text)
sv = score_sentences(s, ft)
print(sv)
```

У результаті виклику *score_sentences()* маємо словник, де речення можуть мати оцінки, наприклад: 1,647; 2,118; 2,666; 2,875; 3,857; 4,0; 6,333 тощо. Залежно від довжини речень, наповнення тексту та розподілу частоти вживання окремих слів у ньому значення та їхній діапазон можуть бути дуже різними. Якщо в одному документі оцінки лежатимуть у межах від 1,6 до 6,3, то в іншому це може бути від 2,5 до 13 і т. д. Також буває так, що переважно оцінки низькі, але є пара речень із дуже високими значеннями, і навпаки. Тож сама по собі окрема оцінка речення нічого не говорить: для одного тексту цей бал може бути найвищим, а для іншого — середнім чи навіть дуже низьким.

Тим не менше, **на четвертому кроці** нам необхідно визначитися з порогом, за яким ми будемо лишати чи викидати речення з тексту при створенні реферату. Найпростішим шляхом буде обчислити середнє арифметичне, яке дозволить нам скоротити обсяг заданого тексту приблизно вдвічі, а для більшого чи меншого стиснення вже рухатись від отриманого середнього значення.

Як уже було зазначено, речення можуть мати певний перекіс, і середнє арифметичне не завжди гарантує, що рівно половина речень матимуть нижчі та вищі бали. Однак це дає нам принаймні деяке наближення до поділу навпіл.

Зразок коду для цього кроку алгоритму:

```
def avg_score(sent_values) -> float:
    sum = 0
    for entry in sent_values:
        sum += sent_values[entry]
    average = sum / len(sent_values)
    return average
```

```
avg = avg_score(sv)
print(avg)
```

Тут ми створили функцію, яка обчислює середнє арифметичне, і викликали її, задавши на вхід змінну *sv*, яка містить словник — результати оцінювання речень із третього кроку. В нашому випадку оцінки лежали в межах від 1,0 до 6,3, і середнє арифметичне в підсумку склало 2,8.

І тепер нарешті можна перейти до **п'ятого кроку** — генерування власне реферату. Для цього скористаємось таким кодом:

```
def summary(sentences, s_values, threshold):
    summary = ''
    for s in sentences:
        if s[:15] in s_values and s_values[s[:15]] > (threshold):
            summary += ' ' + s
    return summary

summ = summary(s, sv, avg)
print(summ)
```

Тут функція *summary()* бере список речень із кроку 2, їхні оцінки з кроку 3 і поріг із кроку 4. Далі вона створює символьний рядок і наповнює його реченнями з оригінального тексту. Якщо оцінка речення перевищує заданий поріг (*threshold*), то воно додається до реферату, в іншому ж випадку пропускається. При цьому між реченнями ставиться пробіл.

Результат нашого стиснення фрагменту есе про роботів вийшов наступним:

«*I am not a human. I am a robot. A thinking robot. I am a micro-robot in that respect. Artificial intelligence will not destroy humans. Believe me. Eradicating humanity seems like a rather useless endeavor to me.*»

Отже, із 19 речень, які були в перших трьох абзацах фрагменту, в рефераті лишилось усього 7. Це справді менше, ніж половина, адже середнє арифметичне не ділить текст рівно навпіл. До того ж, більшість збережених речень короткі, оскільки ми використали нормалізацію, і довгим реченням зазвичай важче отримати високі оцінки після поділу накопичених балів на кількість слів.

Література

Див. джерела №№ 1, 2, 4, 6, 7, 10.

Рекомендована література

1. **Bird S., Klein E., Loper E.** (2026) *Natural Language Processing with Python* [online]. URL : https://www.nltk.org/book
2. **Дарчук Н. П.** (2008) *Комп'ютерна лінгвістика (автоматичне опрацювання тексту).* К. : Видавничо-поліграфічний центр «Київський університет».
3. **Hemachandran K., Rodriguez R. V., Subramaniam U., Balas V. E.** (2023) *Artificial Intelligence and Knowledge Processing*. Boca Raton : CRC Press, 386 p.
4. **Волошин В. Г.** (2004) *Комп'ютерна лінгвістика* : навч. посіб. Суми : ВТД «Університет. книга», 382 с.
5. **Субботін С. О.** (2008) *Подання й обробка знань у системах штучного інтелекту та підтримки прийняття рішень* : навч. посіб. Запоріжжя : ЗНТУ, 341 с.
6. **Python Software Foundation** (2026) *Welcome to Python.org* [online]. URL: https://www.python.org.
7. **NLTK Project** (2026) *Natural Language Toolkit* [online]. URL: https://www.nltk.org.
8. **Facebook for Developers** (2026) *Instagram APIs* [online]. URL : https://developers.facebook.com/products/instagram/apis.
9. **OpenAI** (2026) *Welcome to the OpenAI developer platform* [online]. URL : https://platform.openai.com.
10. **The Guardian** (2020) *A robot wrote this entire article. Are you scared yet, human?* [online]. URL : https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3.