МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Ректор, професор
_____ Олександр ШЕВЧЕНКО
(підпис)
«__» _____ **2025 р.**

КОМП'ЮТЕРНА ЛІНГВІСТИКА

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до виконання лабораторних робіт
для здобувачів освітнього ступеня «бакалавр»
спеціальності 122 «Комп'ютерні науки»
освітньо-професійної програми
«Інформаційні системи та штучний інтелект»
денної форми здобуття освіти

Всі цитати, цифровий та фактичний матеріал, бібліографічні відомості перевірені. Написання одиниць відповідає стандартам

Підписи укладачів:

_____ Микола КОСТІКОВ

« 17 » квітня 2025 р

СХВАЛЕНО
на засіданні кафедри
інформаційних технологій,
штучного інтелекту і
кібербезпеки
Протокол № 11
від 18.04.2025 р.

Реєстраційний номер електронних методичних рекомендацій у НМВ

50.161-2025

КИЇВ НУХТ 2025

Комп’ютерна лінгвістика [Електрон. ресурс]: методичні рекомендації до виконання лабораторних робіт для здобувачів освітнього ступеня «бакалавр» спеціальності 122 «Комп’ютерні науки» освітньо-професійної програми «Інформаційні системи та штучний інтелект» денної форми здобуття освіти / уклад. : М. П. Костіков. К. : НУХТ, 2025. 50 с.

Рецензент: Андрій МОШЕНСЬКИЙ, к. т. н., доц. _____

Укладач: Микола КОСТІКОВ, канд. техн. наук, доц. _____

Відповідальний за випуск: Сергій ГРИБКОВ, д-р техн. наук, проф. _____

Подано в авторській редакції

ЗМІСТ

1. ЗАГАЛЬНІ ВІДОМОСТІ

Лабораторний практикум охоплює усі змістові модулі навчальної програми дисципліни «Комп'ютерна лінгвістика» та призначений для здобувачів закладів вищої освіти, що навчаються за спеціальністю 122 «Комп'ютерні науки» освітньо-професійної програми «Інформаційні системи та штучний інтелект» денної форми навчання.

Метою виконання лабораторних робіт є закріплення у здобувачів знань і вмінь із дисципліни «Комп'ютерна лінгвістика», набуття навичок використання сучасних технологій і засобів опрацювання природної мови для подальшого їх використання у своїй професійній діяльності.

Завданням лабораторного практикуму є допомога здобувачам в опануванні методів комп'ютерної лінгвістики, які розглядаються в рамках навчальної дисципліни. В цьому практикумі викладено загальні рекомендації щодо створення та використання програмних засобів для опрацювання природної мови, наведено короткі теоретичні відомості, завдання та індивідуальні варіанти для виконання лабораторних робіт, запитання для самоконтролю та рекомендовану літературу.

Лабораторні роботи з дисципліни «Комп'ютерна лінгвістика» розроблено відповідно до робочої програми навчальної дисципліни.

Методичні вказівки складаються з чотирьох лабораторних робіт, до яких наведено завдання та індивідуальні варіанти, а також у теоретичній частині послідовно описано різні типи завдань із комп'ютерної лінгвістики, розглянутих у рамках навчальної дисципліни.

Виконання кожної лабораторної роботи передбачає ознайомлення здобувачів із методичними вказівками та теоретичну підготовку з відповідних розділів дисципліни «Комп'ютерна лінгвістика».

Для виконання лабораторних робіт при створенні програмних засобів можна використовувати довільні сучасні мови програмування (зокрема Python, C# тощо), а також довільні середовища розроблення програмних засобів (IDE).

Для здачі лабораторної роботи здобувач має оформити звіт, який містить:

- титульний аркуш;
- формулювання завдання;
- індивідуальний варіант;
- опис ходу виконання роботи;
- висновки.

Бали, отримані за виконання лабораторних робіт, підсумовуються та враховуються при виставленні підсумкової оцінки з навчальної дисципліни.

2. ПРАВИЛА ТЕХНІКИ БЕЗПЕКИ ПРИ ВИКОНАННІ ЛАБОРАТОРНИХ РОБІТ

Загальні положення

1. До роботи в комп'ютерному класі допускаються особи, ознайомлені з даною інструкцією з техніки безпеки та правил поведінки.

2. Робота здобувачів у комп'ютерному класі дозволяється лише у присутності викладача (інженера, лаборанта).

3. Під час занять сторонні особи можуть знаходитися в класі лише з дозволу викладача.

4. Під час перерв між парами проводиться обов'язкове провітрювання комп'ютерного кабінету з обов'язковим виходом здобувачів з нього.

Перед початком роботи необхідно:

1. Переконатися у відсутності видимих пошкоджень на робочому місці.

2. Включити комп'ютери та налагодити роботу.

При роботі в комп'ютерному класі забороняється:

1. Знаходитися в класі у верхньому одязі.

2. Класти одяг і сумки на столи.

3. Знаходитися в класі з напоями та їжею.

4. Розташовуватися збоку або ззаду від включеного монітора.

5. Приєднувати або від'єднувати кабелі, чіпати роз'єми, дроти і розетки.

6. Пересувати комп'ютери і монітори.

7. Відкривати системний блок.

8. Вмикати і вимикати комп'ютери самостійно.

9. Намагатися самостійно усувати несправності в роботі апаратури.

10. Перекривати вентиляційні отвори на системному блоці та моніторі.

11. Ударяти по клавіатурі, натискувати безцільно на клавіші.

12. Приносити і запускати комп'ютерні ігри.

Перебуваючи в комп'ютерному класі, здобувачі зобов'язані:

1. Дотримуватись тиші і порядку.

2. Виконувати вимоги викладача та інженера/лаборанта.

3. Дотримуватись режиму роботи.

4. Після роботи завершити всі активні програми і коректно вимкнути комп'ютер.

5. Залишити робоче місце чистим.

Необхідно дотримуватись правил:

1. Відстань від екрану до очей — 70–80 см.

2. Вертикально пряма спина.

3. Плечі опущені і розслаблені.

4. Ноги на підлозі і не схрещені.

5. Лікті, зап'ястя і кисті рук на одному рівні.

Вимоги безпеки в аварійних ситуаціях:

1. При появі програмних помилок або збоях устаткування здобувач повинен негайно звернутися до викладача (інженера/лаборанта).

2. При появі запаху гару, незвичайного звуку негайно припинити роботу і повідомити викладача (інженера/лаборанта).

3. ПЕРЕЛІК ТА ОПИС КОМПЕТЕНТНОСТЕЙ, ЩО ФОРМУЮТЬСЯ У ЗДОБУВАНА ПІД ЧАС ЛАБОРАТОРНИХ ЗАНЯТЬ

Мета дисципліни — дати здобувачам освіти уявлення про сучасні підходи до опрацювання природної мови в інформаційних системах, а також застосування методів штучного інтелекту для цих завдань. Програма містить розділи, присвячені семантичним мережам і автоматичному опрацюванню текстів. Завданнями навчальної дисципліни є набуття теоретичних знань і практичних навичок щодо методів комп'ютерної лінгвістики та опрацювання природної мови, аналізу текстів тощо для розв'язання прикладних завдань (транслітерація та транскрипція, автоматична перевірка правопису, автокорекція, аналіз слів і тексту, класифікація тексту за тематикою, генерація тексту, створення чат-ботів і т.д.).

Технології, що вивчаються в рамках дисципліни: мова програмування Python; бібліотеки NLTK, Chatterbot та інші; технологія Semantic Web і словник WordNet; середовища розроблення ПЗ JetBrains PyCharm і аналоги; мовні моделі GPT і аналоги.

Міждисциплінарні зв'язки: пререквізитами є дисципліни «Об'єктно-орієнтоване програмування», «Кросплатформне програмування, «Інтелектуальні системи» і «Моделювання систем штучного інтелекту». Постреквізитами є дисципліни «Проєктування інформаційних систем» і «Проєктування експертних систем». У подальшому знання та навички, набуті на дисципліні «Комп'ютерна лінгвістика», так само можуть використовуватися для розв'язання різноманітних завдань опрацювання природної мови, а також опрацювання, аналізу і генерації текстових даних під час виконання випускних кваліфікаційних робіт.

Згідно з вимогами освітньо-професійної програми «Інформаційні системи та штучний інтелект» здобувачі повинні набути **здатності** отримувати компетентності:

інтегральна:

- Здатність розв'язувати складні спеціалізовані задачі та практичні проблеми у галузі комп'ютерних наук або у процесі навчання, що передбачає застосування теорій та методів інформаційних технологій і характеризується комплексністю та невизначеністю умов.

загальні:

- ЗК 2. Здатність застосовувати знання у практичних ситуаціях;
- ЗК 3. Знання та розуміння предметної області та розуміння професійної діяльності;
- ЗК 6. Здатність вчитися і оволодівати сучасними знаннями;
- ЗК 7. Здатність до пошуку, оброблення та аналізу інформації з різних джерел;
- ЗК 10. Здатність бути критичним і самокритичним;
- ЗК 12. Здатність приймати обґрунтовані рішення.

фахові:

- ФК 3. Здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проектування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв’язності та нерозв’язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем;
- ФК 8. Здатність проектувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, крос-платформного з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління;
- ФК 17. Здатність застосовувати методи та алгоритми штучного інтелекту та машинного навчання різного рівня складності для розв’язання прикладних задач при розробленні інформаційних систем, у тому числі для завдань комп’ютерної лінгвістики та опрацювання текстів природною мовою.

Здобувачі повинні досягти таких **програмних результатів навчання:**

- ПРН 1. Застосовувати знання основних форм і законів абстрактно-логічного мислення, основ методології наукового пізнання, форм і методів вилучення, аналізу, обробки та синтезу інформації, в предметній області комп'ютерних наук, володіти іноземною мовою професійного спрямування;
- ПРН 3. Використовувати знання закономірностей випадкових явищ, їх властивостей та операцій над ними, моделей випадкових процесів та сучасних програмних середовищ для розв’язування задач статистичної обробки даних і побудови прогнозних моделей;
- ПРН 4. Використовувати методи обчислювального інтелекту, машинного навчання, нейромережевої та нечіткої обробки даних, генетичного та еволюційного програмування для розв’язання задач розпізнавання, прогнозування, класифікації, ідентифікації об’єктів керування тощо;
- ПРН 9. Розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук;
- ПРН 17. Вміння застосовувати методи та алгоритми штучного інтелекту та машинного навчання різного рівня складності для розв’язання прикладних задач при розробленні інформаційних систем. Вміння будувати та використовувати моделі обробки природної мови, застосовувати алгоритми машинного перекладу, розробляти мовні технології та додатки.

4. РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Виконання лабораторної роботи передбачає ознайомлення здобувача з методичними вказівками до відповідної лабораторної роботи, його теоретичну та практичну підготовку з відповідних розділів дисципліни.

Звіт до лабораторної роботи має бути оформлено відповідно до вимог: титульна сторінка роботи має бути оформлена за шаблоном, наведеним у додатку А, містити завдання та дані індивідуального варіанту, повний код програм(и) та знімки екранів результатів виконання.

Звіти лабораторних робіт виконуються в текстовому редакторі та надсилаються в електронній формі на платформу дистанційного навчання університету.

Робота, виконана з порушеннями наведених вимог, не зараховується і повертається здобувачу для доопрацювання. Робота, що виконана (повністю або частково) за неправильним варіантом, не зараховується.

ЛАБОРАТОРНА РОБОТА 1.
Базові завдання автоматичного опрацювання тексту

Мета: навчитися опрацьовувати текстові дані з допомогою вбудованих методів сучасних мов програмування, а також будувати алгоритми та створювати програмні засоби для транслітерації та транскрипції текстів.

Завдання

Значення по індивідуальних варіантах наведено нижче в **табл. 1.**

Індивідуальний варіант визначається як порядковий номер здобувача в загальному списку групи.

Мова програмування для виконання завдань — довільна.

Варіанти завдань про голосні та приголосні звуки можна реалізувати для української, англійської чи будь-якої іншої (довільної) природної мови.

1. Створити новий консольний проєкт мовою ***Python*** (або іншою). При запуску вивести на екран власне прізвище, ім'я, групу, номер ЛР. Після цього попросити користувача ввести з клавіатури рядок тексту, прийняти цей рядок і зберегти у змінну. Рядок повинен містити літери латинської абетки, української кирилиці, а також пробіли та спеціальні символи.
2. Використовуючи вбудовані методи мови програмування для опрацювання тексту, обробити введений користувачем рядок за індивідуальним варіантом (див. табл. нижче) та вивести результат на екран (у консоль). Для кожного з трьох підзавдань (а, б, в) брати на вхід той же самий початковий рядок у первісному вигляді (як його ввів користувач).
3. За зразками з лекції створити власний метод для транслітерації (або транскрипції) для пари мов за індивідуальним варіантом. Правила транслітерації (транскрипції) для заданих мов знайти самостійно в інтернеті чи інших доступних довідкових джерелах. Метод повинен приймати як параметр рядок тексту вхідною мовою, переводити його у вихідну мову та повертати перетворений рядок як результат.
4. Викликати цей метод, обробити з його допомогою введений користувачем текст і вивести результат на екран (у консоль). Текст повинен містити мінімум 25 слів, при цьому в ньому принаймні по 1 разу мають зустрічатись усі літери абетки вхідної мови (можна використати панграми).
5. Зазначити у звіті з ЛР використану мову програмування, назву та версію встановленого середовища розроблення (IDE), а також назву, версію і мову своєї операційної системи. Помістити у звіт із ЛР повний код програми, таблицю з використаними правилами транслітерації або транскрипції, а також скриншоти результатів роботи програм(и) для різних введених даних.

Табл. 1. Індивідуальні варіанти до лабораторної роботи № 1

№	*Завдання 2а*	*Завдання 2б*	*Завдання 2в*	*Мови*
1	Видалити 2 символи, починаючи з 8-го	Вивести 2-й, 6-й і 9-й символи поспіль	Замінити в рядку всі символи-літери на символ «%»	італійська → українська
2	Замінити всі літери «e» на «i»	Вивести введений рядок в одинарних лапках	Видалити з рядка всі розділові знаки	латинська → українська
3	Вставити рядок «never» між 1 і 2 символом	Перевірити, чи рядок починається з «a» (написати: так / ні)	Підрахувати кількість великих літер у рядку	німецька → українська
4	Вивести перші 4 символи рядка	Об’єднати рядок «cat» і введений рядок	Замінити всі великі літери в рядку на символ «!»	українська → італійська
5	Зробити всі літери в рядку великими	Розбити на слова та вивести кожне слово з нового рядка	Видалити з рядка всі літери, що позначають приголосні звуки	іспанська → українська
6	Знайти індекс першого входження літери «p»	Вивести 1-й, 5-й і 7-й символи через пробіл	Підрахувати в рядку кількість літер, які позначають голосні звуки	українська → португальська
7	Видалити 6 символів, починаючи з 3-го	Вивести введений рядок у подвійних лапках	Замінити в рядку всі символи, що не є цифрами, на символ «=»	українська → французька
8	Замінити всі літери «z» на «s»	Вставити відступ (tab) рівно посередині рядка	Видалити з рядка всі малі літери	румунська → українська
9	Вставити рядок «gonna» між 3 і 4 символом	Об’єднати рядок «dog» і введений рядок	Підрахувати в рядку кількість символів, які є літерами	польська → українська
10	Вивести перші 6 символів рядка	Розбити на слова та вивести кожне слово через відступ (табуляцію)	Замінити всі розділові знаки на символ «@»	словацька → українська
11	Зробити всі літери в рядку малими	Вивести 4-й, 9-й і 11-й символи через дефіс	Видалити з рядка всі символи, які не є літерами	українська → німецька
12	Знайти індекс першого входження літери «r»	Вивести введений рядок у квадратних дужках	Підрахувати в рядку кількість символів, які не є цифрами	хорватська → українська
13	Видалити 4 символи, починаючи з 7-го	Вставити символ «+» після першого слова рядка	Замінити всі літери, що позначають приголосні звуки, на символ «#»	шведська → українська
14	Замінити всі літери «v» на «w»	Об’єднати рядок «woman» і введений рядок	Видалити з рядка всі цифри	французька → українська
15	Вставити рядок «give» між 5 і 6 символом	Розбити на слова та вивести кожне слово через похилу риску («/»)	Підрахувати кількість слів у рядку	українська → іспанська

16	Вивести останні 3 символи рядка	Вивести 3-й, 5-й і 8-й символи через кому	Замінити всі малі літери в рядку на символ «^»	сербська → українська
17	Замінити всі малі літери на великі, і навпаки	Вивести введений рядок у квадратних дужках	Видалити з рядка всі спецсимволи (~!@#$%^&*)	німецька → українська
18	Знайти індекс першого входження літери «і»	Перевірити, чи рядок закінчується на «о» (написати: так / ні)	Підрахувати кількість розділових знаків у рядку	французька → українська
19	Видалити всі символи, починаючи з 4-го	Об’єднати введений рядок і рядок «man»	Замінити в рядку всі символи, які не є літерами, на символ «&»	чеська → українська
20	Замінити всі літери «g» на «h»	Розбити на слова та вивести кожне слово через символ «&»	Видалити з рядка всі літери, що позначають голосні	українська → шведська
21	Вставити рядок «you» між 9 і 10 символом	Вивести 5-й, 7-й і 10-й символи через крапку	Підрахувати в рядку кількість символів, які є цифрами	румунська → українська
22	Вивести останні 7 символів рядка	Вивести введений рядок у трикутних дужках	Замінити в рядку всі дужки (круглі, квадратні, фігурні) на символ «\»	іспанська → українська
23	Вивести кількість символів у рядку	Поділити введений рядок навпіл і вивести кожну половину з нового рядка	Видалити з рядка всі великі літери	українська → німецька
24	Знайти індекс останнього входження літери «d»	Об’єднати введений рядок і рядок «mouse»	Підрахувати в рядку кількість символів, які не є літерами	італійська → українська
25	Видалити всі символи, починаючи з 7-го	Розбити на слова та вивести кожне слово через знак «%»	Замінити всі літери, що позначають голосні звуки, на символ «*»	польська → українська
26	Замінити всі літери «f» на «t»	Вивести 5-й, 7-й і 10-й символи через знак «_»	Видалити з рядка всі символи, які є літерами	хорватська → українська
27	Вставити рядок «up» між 13 і 14 символом	Вивести введений рядок у фігурних дужках	Підрахувати кількість малих літер у рядку	угорська → українська
28	Вивести символи рядка з 5-го по 8-й	Вставити символ «*» перед останнім словом рядка	Замінити в рядку всі цифри на символ «$»	українська → польська
29	Видалити пробіли на початку та в кінці введеного рядка	Об’єднати введений рядок і рядок «horse»	Видалити з рядка всі символи, які не є цифрами	українська → нідерландська
30	Знайти індекс останнього входження літери «е»	Розбити на слова та вивести кожне слово через символ «#»	Підрахувати в рядку кількість літер, які позначають приголосні звуки	українська → сербська

Теоретичні відомості

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'єгер 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.

Висновки

У ході виконання лабораторної роботи визначено, яким чином можна опрацьовувати текстові дані з допомогою вбудованих методів сучасних мов програмування, а також будувати алгоритми та створювати програмні засоби для транслітерації та транскрипції текстів. З'ясовано, що при опрацюванні текстів природною мовою транслітерація та транскрипція дають різні результати.

Контрольні запитання

1. Типи даних для текстової інформації (тут і надалі — на прикладі довільних мов програмування).
2. Способи виведення тексту в консоль із програми.
3. Способи введення користувацького тексту в консоль.
4. Особливості латинської абетки в різних сучасних мовах світу.
5. Особливості кириличної абетки в різних сучасних мовах світу.
6. Кодування тексту. Сучасні види кодувань.
7. Стандарт UTF-8 і його переваги.
8. Таблиці символів (що це таке, принцип побудови, для чого застосовується).
9. Налаштування кодування для кирилиці.
10. Особливості підтримки кирилиці в різних сучасних шрифтах.
11. Панграми та їх застосування.
12. Методи для визначення довжини рядка та індексація символів у рядк.
13. Методи для порівняння рядків між собою.
14. Суть і способи конкатенації рядків.
15. Розбиття рядків на частини.
16. Транслітерація. Приклади.
17. Транскрипція. Приклади.
18. Відмінності між транслітерацією і транскрипцією.
19. Історія стандартів транслітерації української мови латинською абеткою.
20. Особливості сучасного стандарту транслітерації (КМУ, 2010 рік).
21. Застосування права на ім’я при транслітерації.
22. Особливості програмної реалізації транслітерації.
23. Відмінності у транслітерації української мови англійською, німецькою тощо.

Література

Див. джерела №№ 1, 2, 3, 6.

ЛАБОРАТОРНА РОБОТА 2.
Semantic Web і визначення подібності слів

Мета: навчитися працювати з семантичними мережами, словником WordNet і визначати подібність слів за їхніми значеннями через методи обчислення семантичної близькості, а також подібність за написанням через відстань редагування.

Завдання

Значення по індивідуальних варіантах наведено нижче в **табл. 2.** Варіант визначається як порядковий номер здобувача в загальному списку групи.

У завданнях №№ 2–4 слід брати до уваги лише значення іменників.

У завданні № 4 можна взяти лише перше значення кожного іменника.

Для реалізації завдання № 6 можна скористатись кодом із лекції.

6. Створити новий консольний проєкт мовою ***Python*** (або іншою) при запуску вивести власне прізвище, ім'я, групу, номер ЛР. Імпортувати до проєкту бібліотеку ***NLTK*** (або аналог) і корпус ***WordNet,*** який містить семантичний словник англійської мови.

7. Вивести в консоль визначення (тлумачення) для всіх семантичних значень **іменника 1** і **іменника 2** за індивідуальним варіантом.

8. Вивести в консоль усі гіпоніми та гіпероніми для цих же слів.

9. Обчислити семантичну подібність **іменника 1** і **іменника 2** за допомогою методів:
- *Path Distance Similarity*
- *Wu-Palmer Similarity*
- *Leacock Chodorow Similarity*

10. Знайти відстань редагування (Левенштейна) між **іменником 1** і **іменником 2.** При цьому можна обчислювати відстань або реалізувавши код алгоритму вручну, або скориставшись наявними вбудованими функціями бібліотек.

11. Попросити користувача ввести довільне слово англійською мовою. Знайти до цього слова ***N*** (за індивідуальним варіантом) найближчих слів із наявного словника в долученому до завдання текстовому файлі ***1–1000.txt.***

12. ***Завдання на максимальний бал*:** взяти як джерело даних ***текстовий файл за індивідуальним варіантом,*** після чого:
- знайти всі слова, які зустрічаються в цьому файлі;
- відсортувати їх за за спаданням частотності;
- створити ***новий текстовий файл*** і зберегти в ньому відсортовані слова за зразком ***1–1000.txt:*** кожне наступне слово з нового рядка;
- реалізувати для користувача той же функціонал, що і в завданні **№ 6,** але замість ***1–1000.txt*** використати як словник ***новостворений файл.***

13. Помістити у звіт із ЛР повний код програм(и) та скриншоти результатів роботи.

Табл. 2. Індивідуальні варіанти до лабораторної роботи № 2

№	*Іменник 1*	*Іменник 2*	*N*	*Файл*
1	quality	quantity	4	alcott-women.txt
2	employer	employee	5	austen-emma.txt
3	hell	bell	6	austen-persuasion.txt
4	Sunday	Monday	7	austen-sense.txt
5	expression	impression	8	bronte-eyre.txt
6	mountain	fountain	9	bronte-heights.txt
7	money	honey	4	bryant-stories.txt
8	stone	bone	5	carroll-alice.txt
9	pain	gain	6	carroll-glass.txt
10	might	right	7	chesterton-ball.txt
11	lizard	wizard	8	chesterton-brown.txt
12	cat	rat	9	chesterton-thursday.txt
13	chance	dance	4	edgeworth-parents.txt
14	beach	peach	5	melville-moby_dick.txt
15	sun	fun	6	milton-paradise.txt
16	name	game	7	alcott-women.txt
17	winner	loser	8	austen-emma.txt
18	grace	face	9	austen-persuasion.txt
19	word	world	4	austen-sense.txt
20	love	leave	5	bronte-eyre.txt
21	tear	fear	6	bronte-heights.txt
22	career	Korea	7	bryant-stories.txt
23	lust	trust	8	carroll-alice.txt
24	pleasure	treasure	9	carroll-glass.txt
25	violence	silence	4	chesterton-ball.txt
26	ace	base	5	chesterton-brown.txt
27	life	line	6	chesterton-thursday.txt
28	pen	pan	7	edgeworth-parents.txt
29	pineapple	apple	8	melville-moby_dick.txt
30	forgiveness	forgetfulness	9	milton-paradise.txt

Теоретичні відомості

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

NLTK (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 веб-версія 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 веб-сайти, 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.

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;
- 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.

Висновки

У ході виконання лабораторної роботи визначено, яким чином можна працювати зі словником WordNet і визначати подібність слів за їхніми значеннями через методи обчислення семантичної близькості, а також подібність за написанням через відстань редагування. З'ясовано, що різні методи можуть давати різні результати для тієї самої пари слів.

Контрольні запитання

1. Бібліотека NLTK та її особливості.
2. Завантаження та використання корпусів текстів при роботі з NLTK.
3. Semantic Web.
4. Словник WordNet і його особливості.
5. Види лексичних відношень між словами у WordNet. Приклади.
6. Синсети WordNet та основні дії над ними.
7. Гіпоніми та гіпероніми. Приклади.
8. Що таке спільний гіперонім для 2 слів та яке його практичне застосування?
9. Методи обчислення семантичної подібності у WordNet та їхні особливості.
10. Відстань редагування (Левенштейна) і Дамерау — Левенштейна. Суть поняття.
11. Сфери практичного застосування відстані редагування.
12. Переваги та недоліки підходу до обчислення подібності слів через відстань редагування.

13. Альтернативи до методів Левенштейна і Дамерау — Левенштейна та їхні особливості.
14. Застосування відстані редагування для завдання перевірки правопису.

Література

Див. джерела №№ 1, 2, 3, 4, 6, 7.

ЛАБОРАТОРНА РОБОТА 3.
Автоматичне опрацювання текстів

Мета: навчитися проводити автоматичну класифікацію текстів із використанням методів машинного навчання (зокрема наївного баєсівського методу), а також визначати точність отриманих результатів.

Завдання

Значення по індивідуальних варіантах наведено нижче в **табл. 3.** Варіант визначається як порядковий номер здобувача в загальному списку групи.

Необхідні для ЛР вказівки та зразки коду наведено в лекціях.

1. Створити новий консольний проєкт мовою ***Python*** (або іншою), при запуску вивести власне прізвище, ім'я, групу, номер ЛР.
2. Встановити та імпортувати в проєкт бібліотеку ***NLTK*** (або аналог) і корпус ***movie_reviews*,** який містить позитивні та негативні відгуки на фільми.
3. Створити список відгуків і перемішати його (наприклад, із допомогою бібліотеки ***random***).
4. Створити список, у який додати всі слова з усіх відгуків, відсортувати його за частотою вживання і вивести в консоль **20** найбільш уживаних у цьому корпусі.
5. Знайти кількість вживань слова за індивідуальним варіантом (див. нижче).
6. Взяти ***N*** (за інд. варіантом) найбільш уживаних слів у корпусі та створити функцію, яка перевіряє їх наявність у поданому текстовому файлі та повертає словник у вигляді: `{'word1': True, 'word2': False, ...}`
7. Перевірити створену функцію, подавши на вхід файл із папки pos за інд. варіантом. Вивести на екран ті слова з ***N*** найуживаніших, які є в цьому файлі.
8. Реалізувати класифікатор за наївним баєсівським методом (*Naive Bayes*). Навчити модель, узявши для тренувального набору **1800** випадкових відгуків із корпусу ***movie_reviews*.**
9. Перевірити створену модель на решті відгуків із корпусу і вивести на екран точність класифікації.
10. Вивести на екран **20** слів, які згідно з проведеною класифікацією найчіткіше вказують на приналежність відгуку до позитивних чи негативних.
11. Помістити у звіт із ЛР повний код програм(и) та скриншоти результатів роботи.

Табл. 3. Індивідуальні варіанти до лабораторної роботи № 3

№	***Слово***	***N***	***Файл***
1	young	2000	cv001_18431.txt
2	wonderful	2100	cv002_15918.txt
3	miracle	2200	cv003_11664.txt
4	beautiful	2300	cv004_11636.txt

5	magical	2400	cv005_29443.txt
6	happy	2500	cv006_15448.txt
7	joyful	2600	cv007_4968.txt
8	playful	2700	cv008_29435.txt
9	sensible	2800	cv009_29592.txt
10	logical	2900	cv010_29198.txt
11	responsible	3100	cv011_12166.txt
12	practical	3200	cv012_29576.txt
13	dependable	3300	cv013_10159.txt
14	clinical	3400	cv014_13924.txt
15	intellectual	3500	cv015_29439.txt
16	cynical	3600	cv016_4659.txt
17	deep	3700	cv017_22464.txt
18	simple	3800	cv018_20137.txt
19	absurd	3900	cv019_14482.txt
20	radical	4000	cv020_8825.txt
21	liberal	4100	cv021_15838.txt
22	fanatical	4200	cv022_12864.txt
23	criminal	4300	cv023_12672.txt
24	acceptable	4400	cv024_6778.txt
25	respectable	4500	cv025_3108.txt
26	digital	4600	cv026_29325.txt
27	unbelievable	4700	cv027_25219.txt
28	bloody	4800	cv028_26746.txt
29	marvelous	4900	cv029_18643.txt
30	super	5000	cv030_21593.txt

Теоретичні відомості

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 6 найбільш інформативних слів:

```
classifier.show_most_informative_features(6)
```

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
   balancing = True   pos : neg  =   7.0 : 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.

Висновки

У ході виконання роботи визначено, як можна проводити автоматичну класифікацію текстів із використанням методів машинного навчання (зокрема наївного Баєсового) та визначати точність отриманих результатів. З’ясовано, що точність залежить від обраного методу, тренувального і тестового наборів даних.

Контрольні запитання

1. Функція entailments() при роботі з WordNet.
2. Пошук логічних зв’язків у тексті (recognizing text entailments) через NLTK.
3. Реферування тексту.
4. Анотування тексту.
5. Різниця між анотуванням і реферуванням тексту.
6. Суть завдання класифікації тексту. Приклади застосування.
7. Сентимент-аналіз (аналіз тональності) тексту. Приклади.
8. Підхід (кроки) до розв’язання задачі класифікації відгуків на фільми.
9. Особливості реалізації класифікації тексту в NLTK.
10. Наївний Баєсів алгоритм. Суть підходу.
11. Методи-альтернативи наївного баєсівського алгоритму.
12. Точність класифікації тексту (що відображає, в яких межах лежить тощо).
13. Серіалізація та десеріалізація в Python (pickling та unpickling).
14. Суть поняття «машинне навчання». Тренувальні та тестові набори даних.
15. Пошук кількості входжень слова у текстовому файлі. Приклад реалізації.
16. Пошук N найуживаніших слів при аналізі текстових файлів.
17. Задача множинної класифікації текстів та підходи до її розв’язання.

Література

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

ЛАБОРАТОРНА РОБОТА 4.
Автоматична генерація текстів

Мета: навчитися проводити автоматичну генерацію текстів і створювати чат-ботів.

Завдання

Значення по індивідуальних варіантах наведено нижче в **табл. 4.** Варіант визначається як порядковий номер здобувача в загальному списку групи.

Необхідні для ЛР вказівки та зразки коду наведено в лекціях.

1. Створити новий консольний проєкт мовою ***Python*** (або іншою), при запуску вивести власне прізвище, ім'я, групу, номер ЛР. Встановити та імпортувати до проєкту бібліотеки ***NLTK*** (або аналог) і ***ChatterBot*** (версію **1.0.0** або іншу).
2. Створити функцію, яка на вхід як аргумент приймає текст (змінну рядкового типу), визначає залежності між словами за ланцюгом Маркова (для кожного наявного слова — всі слова, які йдуть після нього) та повертає їх як словник (ключ — слово, значення — список наступних слів).
3. Створити функцію генерації тексту, яка на вхід як аргумент приймає створений у **п. 2** словник і число слів для генерації, після чого генерує текст заданої довжини з урахуванням залежностей зі словника та виводить його в консоль.
4. Використати розроблені функції, задавши для створення словника текстовий файл за індивідуальним варіантом (див. табл. нижче), а для генерації тексту — число слів ***N*** (там само).
5. Реалізувати чат-бота через бібліотеку ***ChatterBot*** (або аналог) за зразком із лекції: задати список фраз ***small_talk,*** провести навчання бота на цих фразах і перевірити його роботу в діалозі (мінімум — **10** фраз користувача).
6. Завдання на максимальний бал: Провести навчання чат-бота на корпусі мовою за індивідуальним варіантом (див. табл. нижче) та перевірити його роботу в діалозі (мінімум — **10** фраз користувача).
7. Створити власного телеграм-бота через бот **@BotFather.** Задати назву, ім'я користувача (***username***), опис профіля (***abouttext***), опис роботи (***description***), встановити аватарку (***userpic***) та зберегти отриманий токен для керування ботом.
8. Задати та запрограмувати дві команди для виконання ботом:
   - **/start,** яка виводить вітання та інформацію про автора (ПІБ тощо);
   - **/topic,** яка виводить коротку інформацію на довільну тему.
9. Додати для виконання ботом третю команду, яка просить користувача ввести список чисел (мінімум 8), виконує над цим списком дію за індивідуальним варіантом і повертає результат опрацювання повідомленням у чат.
10. Помістити у звіт із ЛР повний код програм(и) та скриншоти результатів роботи.

Табл. 4. Індивідуальні варіанти до лабораторної роботи № 4

№	*Файл*	*N*	*Мова*	*Дія*
1	alcott-women.txt	10	англійська	помножити всі числа на 2
2	austen-emma.txt	11	іспанська	поділити всі числа на 3
3	austen-persuasion.txt	12	італійська	додати до всіх чисел 4
4	austen-sense.txt	13	німецька	відняти від усіх чисел 5
5	bronte-eyre.txt	14	французька	піднести числа до квадрата
6	bronte-heights.txt	15	шведська	піднести числа до куба
7	bryant-stories.txt	16	англійська	лишити тільки парні числа
8	carroll-alice.txt	17	іспанська	лишити тільки непарні числа
9	carroll-glass.txt	18	італійська	видалити більші за сер.арифм.
10	chesterton-ball.txt	19	німецька	видалити менші за сер.арифм.
11	chesterton-brown.txt	10	французька	обчислити суму елементів
12	chesterton-thursday.txt	11	шведська	обчислити добуток елементів
13	edgeworth-parents.txt	12	англійська	знайти суму парних чисел
14	melville-moby_dick.txt	13	іспанська	знайти суму непарних чисел
15	milton-paradise.txt	14	італійська	знайти добуток парних чисел
16	alcott-women.txt	15	німецька	знайти добуток непарних чисел
17	austen-emma.txt	16	французька	зробити додатні числа від'ємними
18	austen-persuasion.txt	17	шведська	зробити від'ємні числа додатними
19	austen-sense.txt	18	англійська	замінити парні числа на 0
20	bronte-eyre.txt	19	іспанська	замінити непарні числа на 0
21	bronte-heights.txt	10	італійська	зробити всі числа додатними
22	bryant-stories.txt	11	німецька	зробити всі числа від'ємними
23	carroll-alice.txt	12	французька	піднести до квадрата парні числа
24	carroll-glass.txt	13	шведська	піднести до куба непарні числа
25	chesterton-ball.txt	14	англійська	знайти суму додатних чисел
26	chesterton-brown.txt	15	іспанська	знайти суму від'ємних чисел
27	chesterton-thursday.txt	16	італійська	знайти добуток додатних чисел
28	edgeworth-parents.txt	17	німецька	знайти добуток від'ємних чисел
29	melville-moby_dick.txt	18	французька	знайти корені додатних чисел
30	milton-paradise.txt	19	шведська	піднести до куба від'ємні числа

Теоретичні відомості

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. 5. 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. 6. 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.

Висновки

У ході виконання лабораторної роботи визначено, яким чином можна проводити автоматичну генерацію текстів і створювати чат-ботів.

Контрольні запитання

1. Ланцюги Маркова. Історія та суть поняття.
2. Граф взаємозв'язків у ланцюгах Маркова.
3. Моделі Маркова 1-го та інших порядків.
4. Особливості моделі Маркова.
5. Варіанти (приклади) застосування моделі Маркова в лінгвістиці.
6. Виправлення помилок у тексті завдяки моделі Маркова.
7. Процес генерації тексту на основі моделі Маркова. Матриця переходів.
8. Особливості реалізації моделі Маркова мовою Python.
9. Генерація тексту на основі текстового корпусу.
10. Чат-боти та їх застосування в сучасному світі.
11. Особливості бібліотеки ChatterBot.
12. Процес навчання (тренування) чат-бота. Джерела даних для навчання.
13. Застосування генерації тексту для інших завдань, крім чат-ботів.
14. Суть і принцип роботи моделі GPT.
15. Особливості моделі GPT.
16. Шляхи застосування GPT.
17. Якість роботи та недоліки GPT.
18. Пошук інформації за допомогою GPT.
19. Генерація коду за допомогою GPT.
20. Аналоги моделі GPT.
21. ChatGPT.

Література

Див. джерела №№ 1, 2, 3, 6, 7, 8.

Рекомендована література

1. **Bird S., Klein E., Loper E.** Natural Language Processing with Python [online]. 2025. URL : https://www.nltk.org/book
2. **Дарчук Н. П.** Комп’ютерна лінгвістика (автоматичне опрацювання тексту). К. : Видавничо-поліграфічний центр «Київський університет», 2008.
3. **Волошин В. Г.** Комп’ютерна лінгвістика : навч. посіб. Суми : ВТД «Університет. книга», 2004. 382 с.
4. **Hemachandran K., Rodriguez R. V., Subramaniam U., Balas V. E.** Artificial Intelligence and Knowledge Processing. Boca Raton : CRC Press, 2023. 386 p.
5. **Дьогтяр К. В., Костіков М. П.** Проєктування програмного засобу для автоматичного визначення мови ворожнечі // Матер. 89 міжнар. наук. конф. молодих учених, аспірантів і студентів «Наукові здобутки молоді — вирішенню проблем харчування людства у XXI столітті», 3–7 квітня 2023 р. – К.: НУХТ, 2023. – Ч. 2. – С. 326.
6. **Python Software Foundation.** Welcome to Python.org [online]. URL: https://www.python.org
7. **NLTK Project.** Natural Language Toolkit [online]. URL: https://www.nltk.org
8. **OpenAI.** Welcome to the OpenAI developer platform [online]. URL : https://platform.openai.com
9. **TextAttributor 1.0:** Комп'ютерний аналіз українськомовного тексту [online]. URL : http://ta.mova.info

ДОДАТОК А. ШАБЛОН ТИТУЛЬНОЇ СТОРІНКИ ЛАБОРАТОРНОЇ РОБОТИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

кафедра інформаційних технологій,
штучного інтелекту і кібербезпеки

ЗВІТ

із лабораторної роботи № __

з дисципліни «Комп'ютерна лінгвістика»

на тему: «_____»

Варіант ___

Виконав/-ла:

Здобувач(ка) групи КН-_____

_____

Перевірив:

к.т.н., доц. Костіков М. П.

Київ — 2025

ДОДАТОК Б. КОНТРОЛЬ ТА ОЦІНЮВАННЯ РЕЗУЛЬТАТІВ НАВЧАННЯ

Розподіл балів за окремими елементами змістових модулів та методи поточного контролю успішності навчання здобувачів денної форми здобуття освіти

Таблиця Б.1

№ елем. змістового модуля	Елементи змістового модуля	Кількість балів		Поточний контроль навчальної роботи здобувачів
		мінімальна	максимальна	методи контролю
1	2	3	4	5
Модуль 1				
1.	**Лекційний курс.** Теми 1–10	6	10	Написання модульної контрольної роботи
	Лабораторна робота 1.	9	15	Виконання і захист лабораторної роботи
	Лабораторна робота 2.	9	15	Виконання і захист лабораторної роботи
	Лабораторна робота 3.	9	15	Виконання і захист лабораторної роботи
	Лабораторна робота 4.	9	15	Виконання і захист лабораторної роботи
	Разом за модулем	**42,0**	**70,0**	
	Диф. залік	**18,0**	**30,0**	
	Усього за семестр	**60**	**100**	

Критерії оцінювання програмних результатів навчання здобувачів денної форми здобуття освіти за окремими елементами змістових модулів

Таблиця Б.2

Елемент модуля та критерії його оцінювання	Кількість балів
Лабораторна робота 1, 2, 3, 4	**15**
Робота відпрацьована та вчасно захищена, надані повні обґрунтовані відповіді	14–15
Робота відпрацьована та вчасно захищена, при відповіді допущені неточності	11–13
Робота відпрацьована, відповіді неповні, допущені помилки	9–10
Робота відпрацьована, відповіді незадовільні, допущені грубі помилки	5–8
Робота не відпрацьована або дані незадовільні відповіді	0–4

Модульна контрольна робота	**10**
У роботі надано повні обґрунтовані відповіді	9–10
Відповіді неповні, допущено помилки	7–8
Відповіді неповні, допущено суттєві помилки	6
Дано незадовільні відповіді	0–5

Критерії оцінювання програмних результатів навчання здобувачів денної форми здобуття освіти

(форма підсумкового контролю — диф. залік)

27…30 балів ставиться здобувачу, який демонструє повні і глибокі знання навчального матеріалу з питань дисципліни, вільно володіє науковими термінами та проявляє високу комунікативну культуру;

23…26 балів ставиться здобувачу, який засвоїв основний навчальний матеріал, володіє необхідними знаннями з дисципліни, але при цьому допускає окремі несуттєві помилки та неточності, демонструє достатній рівень комунікативної культури;

18…22 бали ставиться здобувачу, який виявляє дещо обмежені знання навчального матеріалу, не виявляє самостійності суджень, не володіє достатніми знаннями з дисципліни, демонструє недоліки комунікативної культури;

0…17 балів ставиться здобувачу, який не володіє необхідними знаннями, науковими термінами в області дисципліни, демонструє низький рівень комунікативної культури.