КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп’ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття ступеня магістра

за спеціальністю 121 Інженерія програмного забезпечення
на тему:

РОЗРОБКА ПРОГРАМИ ДЛЯ ВИПРАВЛЕННЯ ГРАМАТИЧНИХ ПОМИЛОК В ТЕКСТІ

Виконав студент 2-го курсу магістратури
Яковлев Владислав Олександрович ____________
(підпис)

Науковий керівник:
Доцент, кандидат фізико-математичних наук
Ярослав ЛІНДЕР ____________
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент ____________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«____» ________________ 202_ р.,
протокол № ___
Завідувач кафедри, професор, доктор фізико-математичних наук
О. І. Провотар ____________
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 46 сторінок, 4 ілюстрації, 3 таблиці, 11 використаних джерел.

РОЗПІЗНАВАННЯ ПОМИЛОК, ВИПРАВЛЕННЯ ПОМИЛОК, СТАТИСТИЧНА МОДЕЛЬ МОВИ, МАШИННЕ НАВЧАННЯ

Об'єктом роботи є дослідження методів машинного навчання для виправлення помилок у тексті.

Метою даної роботи є створення програмного забезпечення для виправлення помилок у тексті.

Новизною даної роботи є створення фреймворку, який пропонує слова на основі вводу початку цих слів навіть якщо він містить помилку.

Інструменти розроблення: середовище розробки Xcode 12, мова програмування Swift 5.

Результати роботи: була розроблена програма, яка створює модель мови за допомогою тренування на великих текстових збірках. Також створена програма для випралення помилок в словах англійської мови та фреймворк для автоматичного закінчення слів на основі вводу з помилками.

ЗМІСТ

ВСТУП 4
РОЗДІЛ 1 МЕТОДИ ЗНАХОДЖЕННЯ ПОМИЛОК 7
1.1 Метод словникового пошуку 10
1.2 Метод аналізу n-грам 15
РОЗДІЛ 2 МЕТОДИ ВИПРАВЛЕННЯ ПОМИЛОК 23
2.1 Метод мінімальної відстані 25
2.2 Метод подібності ключів 29
2.3 Метод правил 31
2.4 Метод n-грам 31
2.5 Метод оцінки ймовірності 32
2.6 Метод нейронних мереж 33
РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 34
3.1 Опис програмного забезпечення 36
3.2 Реалізація програмного забезпечення 40
ВИСНОВКИ 45
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 46

ВСТУП

Оцінка сучасного стану об’єкта дослідження

Завдання текстової корекції цікавила дослідників в процесі обробки природної мови та машинного навчання протягом багатьох років. Протягом останніх декількох десятиліть, коли люди все більше покладаються на комп'ютери для написання та редагування, потреба в автоматичних системах, які можуть ідентифікувати та виправляти помилки, які зазвичай не обробляються стандартними інструментами корекції тексту, зросли ще більше.

Автоматичне виявлення та виправлення помилок тексту, що має широке значення та застосування, є важливим напрямом дослідження обробки природної мови. Дослідження втоматичного виявлення та виправлення текстових помилок за кордоном почалися ще в 1960-х роках. Зараз, з розвитком комп'ютерних технологій, у вигляді нескінченного потоку з'явилися нові технології, такі як розпізнавання голосу, розпізнавання тексту на зображеннях та розпізнавання жестів, які потребують перевірки, а дослідження щодо автоматичного виявлення та виправлення текстових помилок також швидко розвиваються та впроваджуються всюди, де використовується системи розпізнавання, пошук інформації за ключовими словами, перевірка текстів та при наборі текстів.

Актуальність роботи та підстави для її виконання

Проблема знаходження та виправлення помилок є дуже актуальною, оскільки зараз широко використовуються системи розпінавання та корекції помилок у різних комп’ютерних системах, де, використовуючи системи розпізнавання, маємо на виході текстову інформацію.

Найпопулярнішими темами у світі розпізнавання зараз є системи розпізнавння голосу, тексту та жестів.

Ці системи не дуже досконалі і мають іноді велику похибку розпізнавання. Тому було б добре розробити систему коректування вихідного результату у вигляді текстової інформації. Це в рази підвищило б точність систем розпізнавання.

Мета й завдання роботи

Метою кваліфікаційної роботи є створення програмного забезпечення для розпізнавання та коректування помилок у тексті. Для досягнення цієї мети поставлено такі завдання.

- Дослідити існуючі методи розпізнавання помилок.
- Дослідити існуючі методи виправлення помилок.
- Розробити алгоритм роботи програми.
- Розробити інтерфейс та дизайн програмного продукту.
- Розробити загальний фреймворк.

Об'єкт, методи й засоби розроблення

Об'єктом розроблення програмного забезпечення є процес розв'язування задачі знаходження та коректування помилок у тексті. Розробці програмного засобу передувало створення алгоритму роботи програми. Основу для цього склало дослідження основних методів розпізнавання помилок.

Під час створення програмного забезпечення також був розроблений допоміжний програмний модуль, який на вхід отримує збірник текстів деякої конкретної мови, а на вихід передає створену модель цієї мови за допомогою процесу навчання. Ця модель, як ресурс, порібна для роботи основного програмного модуля.

В якості інструменту створення програмного засобу було обрано Xcode 12 IDE – інтегроване середовище розробки, та мова Swift 5.

Можливі сфери застосування

Розроблений програмний продукт може бути застосований як допоміжний модуль до будь-яких систем розпізнавання, в яких результатом є текстова інформація. Це допоможе зменшити кількість помилкових результатів.

Також може допомогти при перевірці текстових документів та при наборі текстів.

Може разом зі системами розпізнавання рукописних текстів перевіряти помилки у роботах студентів і школярів для автоматизації процесу перевірки знань.

РОЗДІЛ 1 МЕТОДИ ЗНАХОДЖЕННЯ ПОМИЛОК

Проблема розробки алгоритмів та методів автоматичного виправлення слів у тексті стала проблемою багаторічного дослідження. Роботи почалися ще в 1960-х рр. над методами автоматичної корекції орфографії та автоматичного розпізнавання тексту, і це продовжувалось до сьогодення. Існують вагомі підстави для продовження досліджень у цій галузі.

Розвиваються людино-комп’ютерні та комп’ютерно-комунікаційні технології, відкрив двері для безлічі нових систем які потребують кращого розпізнавання слів і можливості виправлення помилок. Нові інтерфейси дозволять користувачам забезпечити рукописний ввід на комп'ютерах; пристрої розпізнавання тексту дозволять здійснити сканування друкованого матеріалу в повсякденних умовах; пристрої синтезу голосу (текста в мову) дозволять почути текстовий матеріал; а розпізнавання голосу (мова до тексту) дозволить голосові введення в комп'ютерних системах. Але жодне з цих додатків не стане практичним доти, доки не будуть внесені значні покращення в області розпізнавання та виправлення слів. Деякі інші програми, які будуть корисні від таких покращень, включають більш складні програмні засоби для редагування тексту та кодів, машинний переклад, вивчення мов, комп'ютерне навчання та взаємодія з базою даних, а також різні голосові введення та допоміжні засоби для інвалідів, такі як пристрої факсимільного зв'язку та послуги фонетичної транскрипції.

На початку дослідники, що працювали в рамках парадигми автоматичної корекції орфографії та автоматичного розпізнавання тексту, створювали дещо самостійно, використовуючи різні методи. З часом різні методи стали перетинатися між собою, так що сьогодні існують численні гібридні підходи та багато досить успішних систем. Але ті, хто вважає, що питання правопису це вирішена проблема, можуть не усвідомлювати всю природу данної проблеми та границі існуючих методів.

Необхідно провести відмінність між завданнями виявлення помилок та виправленням помилок. Ефективні методи були розроблені для виявлення рядків, які не відображаються у певному списку слів, словниках або лексиконі. Але виправлення неправильно написаного рядка є набагато складнішою проблемою [1].

Багато існуючих коректорів правопису використовують обмеження для конкретних завдань. Наприклад, коректори правопису інтерактивного командного рядка використовують невеликий розмір лексики для командної мови для швидкого реагування. Деякі майбутні програми, такі як синтез текстового мовлення, вимагатимуть від системи повністю автоматичного розпізнавання слів у реальному часі та виправлення помилок для словників багатьох тисяч слів та імен. Контраст між першим прикладом та останнім підкреслює відмінність між інтерактивними перевірками правопису та автоматичною корекцією. Останнє завдання є набагато більш вимогливим, і незрозуміло, наскільки існуючі методи корекції орфографії можуть перейти до повної автоматичної корекції слів.

Більшість існуючих методів корекції орфографії зосереджують увагу на ізольованих словах, не беручи до уваги будь-яку інформацію, яка може бути знята з контексту, в якому це слово знаходиться. Такі методи корекції ізольованих слів не можуть виявити значну частину помилок, включаючи типографічні, фонетичні та граматичні помилки. Розробка контекстної корекції помилок стала головною проблемою для автоматичного розпізнавання слів і виправлення помилок в тексті [2].

Два основні методи, які досліджується для виявлення помилок це n-грамний аналіз і пошук у словнику. N-грами - це послідовності з n-букв або слів, де n, як правило, один, два або три. Однорядкові n-грами називаються уніграмами або монограмами; дворядкові n-грами називають digrams або bigrams; трьохрядкові n-грами як триграми. Загалом, методи виявлення помилок за допомогою n-gram працюють, вивчаючи кожну n-граму в рядку вводу та дивлячись на попередньо скомпільовану таблицю n-грамової статистики, щоб

з'ясувати або її існування, або її частоту. Рядки, які, як вважають, містять неіснуючі або дуже рідкісні n-грами, визначаються як ті, що містять орфографічні помилки. Методи n-gram звичайно вимагають або словника, або великий корпус тексту, щоб попередньо компілювати таблицю n-грам. Методи пошуку у словнику працюють, просто перевіряючи, чи наявний в словнику рядок введення, тобто список вхідних слів. Якщо ні, то рядок позначається як неправильний. Є тонкі проблеми, пов'язані зі складанням корисного словника для програми корекції орфографії [3].

1.1 Метод словникового пошуку

Метод словникового пошуку - це простий метод. Проте час пошуку стає проблемою, коли розмір словника перевищує кілька сотень слів. Для обробки документів та отримання інформації кількість записів в словнику може складати від 25 000 до більш ніж 250 000 слів. Ця проблема була вирішена завдяки ефективному алгоритму пошуку за словником, алгоритмам узгодження з шаблонами, за допомогою схем розділу словників та методів морфологічної обробки.

Найпоширенішим методом отримання швидкого доступу до словника є використання хеш-таблиць.

Хеш-таблиця — це спеціальна структура даних, що імплементує інтерфейс асоціативного масиву, а точніше, вона дозволяє зберігати пари об’єктів (ключ та його значення) і здійснювати три операції: операцію додавання нової пари об’єктів, операцію пошуку і операцію видалення за ключем.

Існує два основних типи хеш-таблиць: з відкритою адресацією та з ланцюжками. Хеш-таблиця відображає в собі деякий масив S, елементами якого є пари об’єктів (хеш-таблиця з відкритою адресацією) або списки пар (хеш-таблиця з ланцюжками).

Виконання операцій в хеш-таблиці починається з обчислення хеш-функції від ключа, яка повертає отримане хеш-значення $i = hash(key)$, яке представляє роль індексу в масиві S. Після цього операція (додавання, видалення, пошук) передаються об'єктові, який зберігається у відповідній комірці масиву $S[i]$. Випадок, при якому для різних ключів отримується одне й те саме хеш-значення, називається колізією. Такі події непоодинокі — наприклад, при додаванні в хеш-таблицю розміром 365 комірок усього лише 23-х елементів ймовірність колізії вже перевищує 50 відсотків (якщо кожний елемент може з

однаковою ймовірністю потрапити в будь-яку комірку). Через це механізм розв'язання колізій — важлива складова будь-якої хеш-таблиці.

Є деякі особливі випадки коли взагалі вдається уникнути колізій. Наприклад, якщо всі ключі елементів відомі наперед (або дуже рідко змінюються), тоді для цих ключів можна побудувати деяку досконалу хеш-функцію, яка розподілить їх за комірками хеш-таблиці без колізій. Хеш-таблицям, які використовують подібні хеш-функції, не потрібні механізми розв'язання колізій, і вони називаються хеш-таблицями з прямою адресацією .

Важлива особливість хеш-таблиць полягає в тому, що, всі три операції (вставлення, пошук і видалення елементів) зазвичай виконується за час *O(1)*. Але при цьому не має гарантії, що час виконання окремої операції малий, з деякою ймовірністю час може бути близьким із пошуком у списку. При збільшенні коефіцієнта заповненості таблиці ця ймовірність, і, відповідно, середній час виконання операцій, росте. Тому при досягненні деякого значення коефіцієнта заповнення необхідно проводити перебудову індексів хеш-таблиці: збільшити розміри масиву S і з початку додати в порожню хеш-таблицю всі пари [4].

Основна перевага хеш-таблиць полягає в тому, що випадковий доступ до значення ключа за хеш-кодом виключає велику кількість порівнянь, необхідних для послідовних чи навіть дерево-подібних пошуків. Основним недоліком є необхідність розробки розумної хеш-функції, яка дозволяє уникнути колізій, не вимагаючи величезного розміру хеш-таблиці.

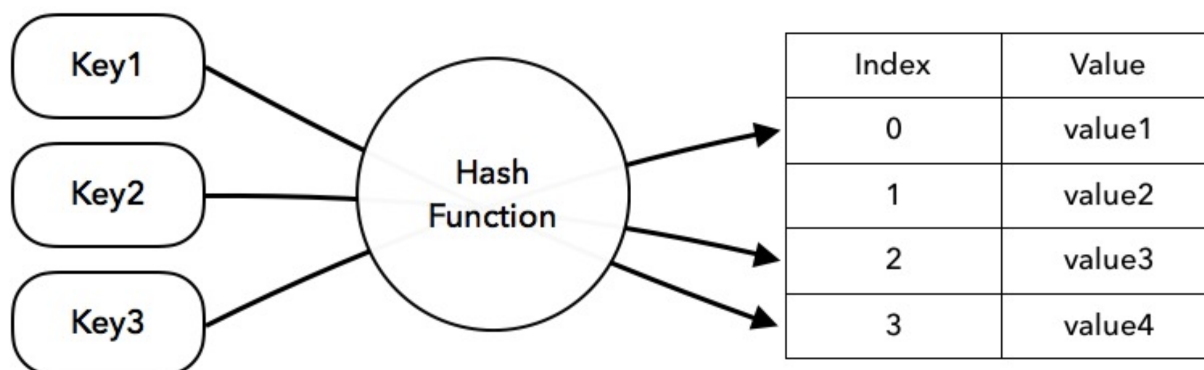


Рисунок 1 - Хеш-таблиця

Отже, щоб перевірити слово на правильність, обчислюється його хеш-адреса і отримується слово, що зберігається по цій адресі, у попередньо сформованій хеш-таблиці. Іноді для пошуку потрібно пройтися по декількох словах, які належать одній хеш-адресі, якщо сталися колізії під час побудови хеш-таблиці. Якщо слово, збережене на хеш-адресі, відрізняється від шуканого слова або взагалі не існує даної хеш-адреси, то слово позначається як орфографічно неправильне.

Інші методи стандартної пошуку, такі як, префіксні дерева пошуку, бінарні дерева пошуку з заданими частотами, були використані для скорочення часу пошуку в словнику.

Префіксне дерево (англ. prefix tree) — це спеціальна структура даних, дерево, в якому зберігаються рядки (ланцюжок), кожен з яких визначається як шлях від кореня до листа. Ланцюжки з однаковими префіксами формують спільний шлях від кореня довжиною цього префікса.

Префіксне дерево дає можливість зберігати асоціативний масив, ключами якого є рядки. На відміну від бінарних дерев, в листах дерева не зберігаються ключі. Значення ключа можна отримати, проходжуючись по всім батьківським вузлам, кожний з яких зберігає один або кілька символів алфавіту. Корінь дерева асоціюється з порожнім рядком. Таким чином, нащадки вузла мають ідентичний префікс, звідки і відбулася назва даного абстрактного типу даних. Значення, пов'язані з ключем, зазвичай не пов'язані з кожним вузлом, а тільки з листами і, можливо, деякими внутрішніми вузлами.

Префіксне дерево має такі переваги над хеш-таблицею:

- Пошук даних швидше у найгіршому випадку, час *O(m)* (де *m* - довжина рядка пошуку), порівняно з недосконалою хеш-таблицею. Недосконала хеш-таблиця може мати колізії з ключами. Найгірша швидкість пошуку

в недосконалій хеш-таблиці - це час *O(N)*, але набагато більш типовим є *O(1)*, з часом *O(m)*, витраченим на розрахунок хешу.

- Не існує колізій різних ключів у дереві.
- *Buckets* в дереві, які є аналогами хэш-табличним *buckets*, що зберігають колізії ключів, потрібні, лише якщо один ключ пов'язаний з кількома значеннями.
- Немає необхідності надавати хеш-функцію або змінювати хеш-функцію, по мірі додавання нових ключів.
- Дерево може забезпечити алфавітне упорядкування записів за ключем.

Префіксне дерево також має такі недоліки:

- У деяких випадках дерева можуть бути повільнішими, ніж хеш-таблиці для пошуку даних, особливо якщо ці дані доступні безпосередньо на жорсткому диску або на якомусь іншому вторинному пристрої зберігання даних, де час доступу є високим порівняно з основною пам’яттю.
- Деякі дерева можуть вимагати більше місця, ніж хеш-таблиці, тому що пам'ять може бути виділена для кожного символу в рядку пошуку, а не як у хеш-таблицях, де для всього рядка виділяється частина пам’яті.

Загальним застосуванням префіксних дерев є створення систем пропонування та автозакінчення тексту, наприклад, що використовуються в мобільних телефонах при наборі тексту. В таких випадках використовується спроможність дерев швидко шукати, вставляти та видаляти записи.

Отже, щоб перевірити слово на правильність, перевіряється чи містить префіксне дерево таке слово. Для цього вхідне слово розбивається на окремі символи. Тепер потрібно знайти шлях від кореня дерева до листка, який містить всі символи по порядку. Для кожного ноду (листка) з поточним символом перевіряємо чи містить він дочірній нод з наступним символом. Якщо повністю пройшли шлях до останього символа, то слово існує в словнику і вважається правильним, в іншому випадку слово позначається як орфографічно неправильне [5].

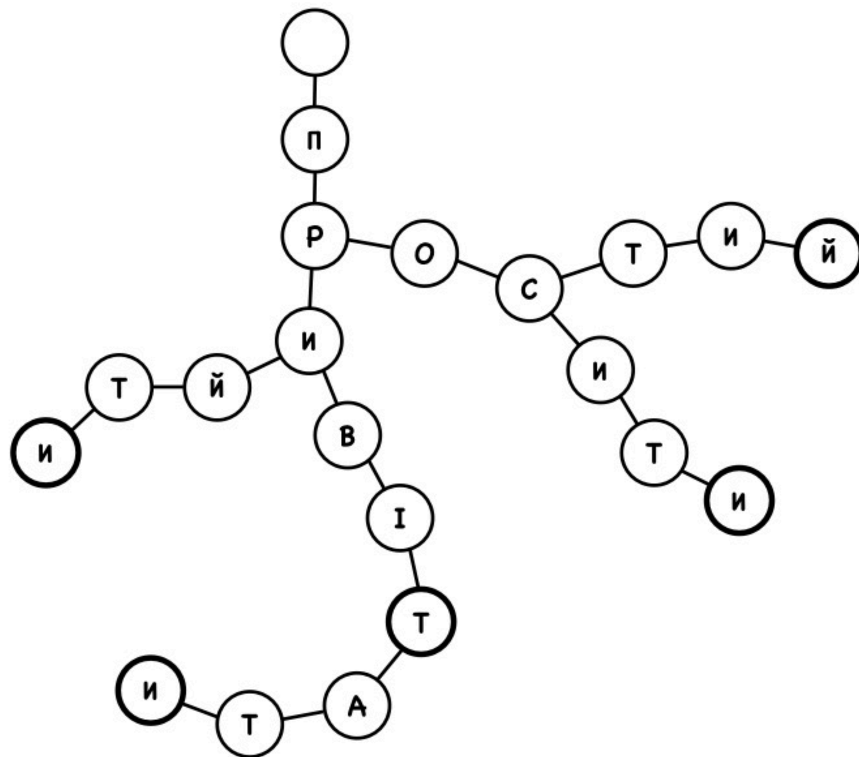


Рисунок 2 - Префіксне дерево, яке містить слова: привіт, привітати, прийти, просити, простий

1.2 Метод аналізу n-грам

Системи розпізнавання тексту зазвичай зосереджені на одному з трьох типів представлення тексту: друкований текст, рукописний текст, або машинодрукований текст. Всі три типи можуть бути оброблені методом оптичного розпізнавання (OCR). Помилки, зроблені пристроями OCR, як правило, є такими, які плутають подібні символи, наприклад 0 і D, S і 5, або t і f. Аналіз n-грам виявився корисним для виявлення таких помилок, оскільки вони, як правило, призводять до неіснуючих або малоймовірних n грам. Поняття n-грам зв'язано з поняттям статистичної моделі мови.

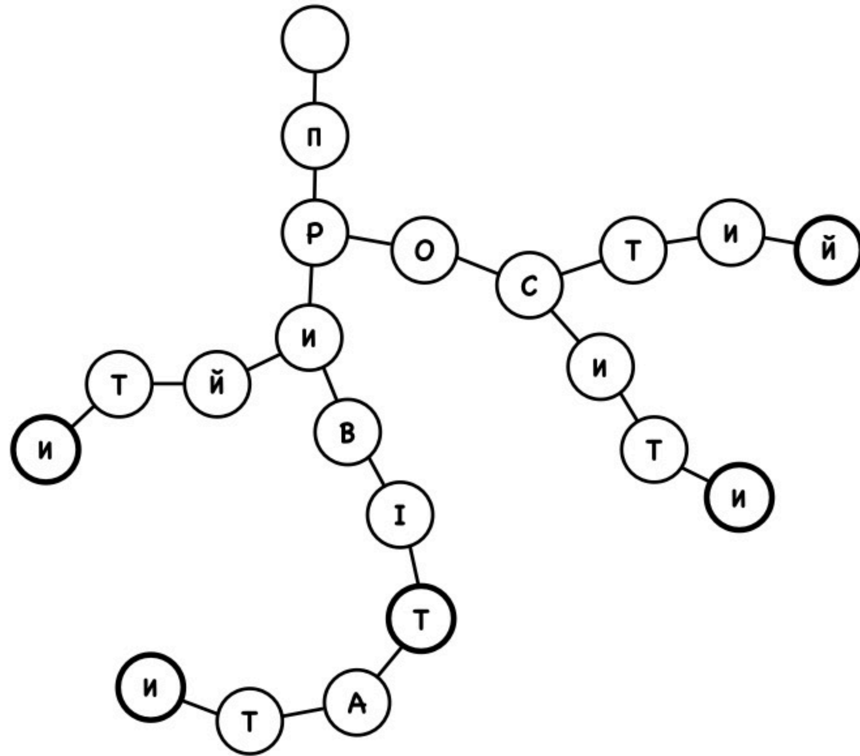
Статистична модель мови (Statistical language model)

Статистична модель мови - це модель, яка описує ймовірнісний розподіл послідовностей слів, який намагається передбачити, як часто дана послідовність зустрічається в мові. Для послідовності довжини m ймовірність позначається як $P(w_1, \ldots, w_m)$.

Спосіб оцінити відносну ймовірність використання різних фраз корисний у багатьох програмах для обробки природної мови, особливо тих, які створюють текст як висновок. Мовне моделювання використовується при розпізнаванні мовлення, машинному перекладі, тегуванні частин мови, розпізнаванні рукописного тексту, пошуку інформації та інших програм [6].

Швидкість даних є однією з основних проблем при побудові мовних моделей. Одним із припущень є те, що ймовірність слова залежить тільки від попередніх n слів. Це називається n-грамною моделлю або моделлю unigram, якщо n = 1.

В unigram моделі ймовірність кожного слова залежить тільки від власної ймовірності цього слова в тексті. Наступним є ілюстрація уніграмної моделі текстового документа (табл. 1).



(1)

Ймов … ся як:

$$P(query) = \prod P(word) \quad (2)$$

Для різних документів ми можемо побудувати власні моделі уніграм з різною вірогідністю слів у ньому. І ми використовуємо ймовірності з різних документів, щоб генерувати різну вірогідність для запиту. Потім ми можемо використовувати документи для запиту відповідно до генеруючих імовірностей. Наступним є приклад двох уніграм моделей для двох документів (табл. 2).

слово	ймовірність в док1	ймовірність в док2
та	0.2	0.3
привіт	0.1	0.08
планета	0.03	0.04

Таблиця 2 - Приклад уніграмних моделей для різних документів

N-грама — це послідовність з n елементів. З точки зору семантики, це може бути послідовність складів, звуків, букв або слів. Загалом частіше зустрічається n-грами як послідовності слів. Послідовність з двох елементів часто називають біграмою, послідовність з трьох елементів називаєть триграмою. Послідовністі з чотирьох і вище елементів позначаються як n-грами, n замінюється на кількість послідовних елементів.

n-грамна модель обчислює ймовірність останнього слова n-грами, якщо відомі всі попередні. Цей підхід для створення моделі мови передбачає, що поява кожного слова залежить тільки від попередніх слів.

n-грами часто успішно використовуються для категоризації тексту та мови. Крім того, їх можна використовувати для створення функцій, які дозволяють отримувати знання з текстових даних. Використовуючи n-грами, можна ефективно знайти кандидатів, щоб замінити слова з помилками правопису [7].

У n-грамній моделі ймовірність $P(w_1, \ldots, w_m)$ для речення $w_1, \ldots, w_m$ розраховується як:

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \quad (3)$$

Передбачається, що ймовірність спостереження *i-го* слова w_i в контексті попередніх *i-1* слів може бути апроксимована ймовірністю його спостереження в укороченому контексті попередніх *n-1* слів. Тоді формула для розрахунку ймовірністі прийме наступний вигляд:

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\mathrm{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\mathrm{count}(w_{i-(n-1)}, \ldots, w_{i-1})} \quad (4)$$

Умовна ймовірність може бути розрахована з кількості n-грам:

Наприклад, для біграмної моделі ймовірність фрази «щастя є задоволення без каяття» розраховується так:

P = P (щастя) * P (є | щастя) * P (задоволення | щастя є) * P (без | щастя є задоволення) * P (каяття | щастя є задоволення без)

Для того, щоб розрахувати ймовірність P (щастя), потрібно лише порахувати кількість разів коли це слово зустрілося в тексті і поділити це значення на загальне число слів. Але розрахувати ймовірність P (каяття | щастя є задоволення без) вже складніше. Зазначимо, що ймовірність слова в тексті залежить тільки від попереднього слова. Тоді наша формула для розрахунку фрази прийме наступний вигляд:

P = P (щастя) * P (є | щастя) * P (задоволення | є) * P (без | задоволення) * P (каяття | без)

Для розрахунку умовної ймовірністі P (є | щастя) треба порахувати кількість пар 'щастя є' і поділити результат на кількість в тексті слова 'щастя'.

У результаті, якщо ми порахуємо всі пари слів в деякому тексті, ми зможемо обчислити ймовірність довільній фрази. Цей набір розрахованих ймовірностей і буде біграмною моделлю.

Також, є завдання балансування ваги між нечастими n-грамами (наприклад, якщо власне ім'я зустрінеться один раз в навчальних даних) та частими n-грамами. Крім того, елементи які не з'являлись в навчальних даних отримають ймовірність 0.0 без згладжування. На практиці, необхідно згладити розподіл ймовірності додавши ненульові ймовірності для слів чи n-грамів які ще не зустрічались.

n-грами знаходять застосування в інформатиці, обчислювальній лінгвістиці та прикладній математиці.

Вони використовуються для:

- знайти можливих кандидатів на правильне написання неправильно написаного слова.
- збільшити компресію в алгоритмах стиснення, де невелика множина даних вимагає n-грами більшої довжини.
- оцінити ймовірність заданої послідовності слів, яка з'являється в тексті мови, для використання такими системами розпізнавання образів як: розпізнавання мови, OCR (оптичне розпізнавання символів), інтелектуальне розпізнавання символів (ICR), машинний переклад та аналогічні системи.
- покращує пошукову продуктивність при аналізі генетичної послідовності
- ідентифікувати мову на якій написаний текст.
- прогнозувати букви або слова при написанні тексту вилучення даних для кластеризації серії супутникових знімків Землі з космосу, щоб потім вирішити, які конкретні частини Землі на зображенні.
- для індексування даних в пошукових системах.

Побудова n-грамної моделі

Побудова або тренування n-грамної моделі відбувається за допомогою спеціальної збірки текстів (text corpus), які відносяться до різних тем (публіцистична література, художня література, розмовна мова і тд.).

Текстовий корпус - це великий і структурований набір текстів (в даний час вони звичайно зберігаються та обробляються електронним способом). Вони

використовуються для проведення статистичного аналізу та тестування гіпотез, перевірки випадків або перевірки лінгвістичних правил у певній частині мови.

Корпус може містити тексти на одній мові (одномовний корпус) або текстові дані на декількох мовах (багатомовний корпус).

Багатомовні корпуса, спеціально відформатовані для порівняння, називаються порівняними паралельними корпусами. Є два основних типи паралельних корпусів, які містять тексти на двох мовах. У перекладацькому корпусі тексти на одній мові є перекладами текстів іншою мовою. У подібному корпусі тексти однакового типу і охоплюють один і той же вміст, але вони не є перекладами один одного. Алгоритми машинного перекладу для перекладу між двома мовами часто тренуються за допомогою паралельних фрагментів, що складаються з корпусу першої мови та корпусу другої мови, який є перекладом корпусу першої мови [8].

Щоб зробити корпуси більш корисними для проведення лінгвістичних досліджень, вони часто піддаються процесу, відомому як анотація. Прикладом коментування корпусу є позначення тегами "частина мови", в яких міститься інформація про частину мови кожного слова (дієслова, іменник, прикметник і т. д.). Іншим прикладом є вказівка лемової (базової) форми кожного слова.

Деякі корпуси мають додаткові структуровані рівні аналізу. Зокрема, цілий ряд менших корпусів може бути повністю розібраний. Такі корпуси, як правило, називаються Treebanks (в них наявні анонтації синтаксичного або семантичного структурування речення). Можливі інші рівні лінгвістичного структурованого аналізу, включаючи анотації для морфології, семантики та прагматики.

Корпуси є основною базою знань у галузі лінгвістики. Аналіз та обробка різних типів корпусів також є предметом значної роботи в галузі обчислювальної лінгвістики, розпізнавання мови та машинного перекладу, де їх

часто використовують для створення прихованих моделей Маркова для позначення частини мови та інших цілей. Корпуси та частотні списки, отримані з них, корисні для навчання мови.

Отже, алгоритм створення n-грамної моделі такий:

1. Береться текстовий корпус деякої мови.
2. Корпус проходить процес розбиття на суміжні n-грами.
3. Для кожної n-грами обчислюємо її ймовірність.
4. Створюємо модель, наприклад, як хеш-таблицю, де ключ - це n-грама, а значення ключа - ймовірность

Виявлення помилок за допомогою n-грам

Отже, n-грами використовуються для знаходження помилок у тексті. Цей метод виявлення неправильно написаних слів у тексті гарно підходить для знаходження помилок не тільки в окремих словах, але й словосполученнях . Замість того, щоб порівнювати ціле слово в тексті зі словником, просто порівнюються n-грами зі словником, тому що процес порівнювання кожного окремого слова зі словником займає багато часу. Перевірка проводиться за допомогою n-мірної матриці, де зберігаються реальні ймовірності n-gram. Якщо виявляється неіснуючий або рідкісний n-грам, це слово позначається як помилка. Кожен рядок, який бере участь у процесі порівняння, поділяється на набори суміжних n грам. Алгоритми n-грам мають велику перевагу, оскільки вони не потребують знання мови, з якою вони працюють.

Тобто, для вхідного тексту алгоритм виявлення помилок буде такий:

1. Розбиваємо текст на набори суміжних n-грам.

2. Для кожної n-грами перевіряємо чи міститься вона у нашій мовній моделі. Якщо не міститься або її ймовірність дуже низька, то помічаємо дану послідовність слів як помилкову.

Мовну модель, як модель n-грам, можна представити двома способами:

1. n-мірна матриця, де зберігаються реальні ймовірності n-gram.
2. хеш-таблиця, де ключ - це n-грама, а значення ключа - ймовірность.

Загалом, хоча n-грамнй аналіз може бути корисним для виявлення машинних помилок, таких як ті, що породжуються розпізнавальними системами оптичного характеру, він виявився менш точним для виявлення помилок, спричинених людиною. Отже, більшість поточних методів корекції орфографії спираються на пошук у словнику для виявлення помилок. Оскільки швидкість доступу може бути проблемою, коли розмір словника є великим (наприклад, > 20 000 записів), використовують хеш-таблиці, дерева та інші структури даних.

РОЗДІЛ 2 МЕТОДИ ВИПРАВЛЕННЯ ПОМИЛОК

Для деяких програм просто виявлення помилок у тексті може виявитись достатнім, однак для більшості систем просте виявлення недостатньо. Наприклад, оскільки мета систем розпізнавання тексту - точне відтворення вхідного тексту, помилки виводу повинні бути виявлені та виправлені.

Подібним чином користувачі сподіваються, що при перевірці орфографії будуть запропоновані виправлення для неправильних слів. Також, більшість систем розпізнавання, такі як синтез текста в мову, вимагають виявлення та виправлення помилок без втручання користувача.

Проблема виправлення помилок в словах містить три підпроблеми:

1. Виявлення помилки
2. Генерація кандидатів на виправлення
3. Надання найбільш релевантних кандидатів

Більшість методів розглядають кожну підпроблему як окремий процес і виконують їх послідовно.

Процес виявлення помилок зазвичай полягає у перевірці, чи є вхідний рядок дійсним словом у словнику або його n-грами є дійсними. Процес генерації кандидатів зазвичай використовує словник або базу даних n-грам, щоб знайти один або кілька можливих кандидатів. Процес отримання релевантних кандидатів зазвичай використовує деяку міру лексичної подібності між помилковим словом та кандидатами або сортує кандидатів за їх ймовірностями, отриманими з n-грам.

Деякі методи пропускають третій процес, залишаючи вибір кандидатів користувачеві. Інші методи, включаючи багато ймовірнісних та нейромережевих методів, об'єднують всі три підпрограми в одну систему обчислення.

Статистична модель n-грам спочатку відігравала центральну роль у способах розпізнавання тексту, тоді як словникові методи підходять більше для корекції правопису. Але дослідники систем розпізнавання тексту швидко з'ясували, що використовувати тільки n-грамний аналіз є недостатнім для завдання корекції, тому вони почали розробляти підходи, які об'єднують використання словників з n-грамною моделлю.

Крім того, було винайдено багато інших розумних методів, що базуються на алгоритмах мінімальної відстані, подібності ключів, оцінці ймовірностей та нейронних мережах.

Отже, методи виправлення помилок:

1. Метод мінімальної відстані.
2. Метод подібності ключів.
3. Метод правил.
4. Метод n-грам.
5. Метод оцінки ймовірності.
6. Метод нейронних мереж.

2.1 Метод мінімальної відстані

Метод мінімальної відстані - це спосіб кількісної оцінки того, наскільки два різні рядки (наприклад, слова) відрізняються один від одного. При цьому підраховують мінімальну кількість операцій, необхідних для перетворення одного рядка в інший. Цей метод використовується в обробці природної мови, де система автоматичної корекції орфографії генерує кандидатів на виправлення, вибираючи слова зі словника, які мають найменшу відстань до неправильного слова.

Різні означення мінімальної відстані використовують різні набори операцій над рядом. Метод Левенштейна полягає в видаленні, вставці або заміні символу в рядку. Оскільки цей метод є найпоширенішим, то говорячи про мінімальну відстань, мають на увазі відстань Левенштейна.

Відстань Левенштейна (також функція Левенштейна, алгоритм Левенштейна або відстань редагування) у теорії інформації і комп'ютерній лінгвістиці міра відмінності двох послідовностей символів (рядків). Обчислюється як мінімальна кількість операцій вставки, видалення і заміни, необхідних для перетворення одної послідовності в іншу.

Метод розроблений у 1965 році радянським математиком Володимиром Йосиповичем Левенштейном і названий його іменем.

Приклад:

Щоб перетворити слово *небо* на слово *треба* необхідно зробити дві заміни та одну вставку, відповідно дистанція Левенштейна становить 3:

1. небо → неба (замінюємо о на а)
2. неба → реба (замінюємо н на р)
3. реба → треба (вставляємо т)

На практиці відстань Левенштейна використовується для визначення того, наскільки дві послідовності символів подібні, наприклад для корекції орфографії або для пошуку дублікатів.

Для обчислення відстані Левенштейна найчастіше застосовують простий алгоритм, в якому використовується матриця розміром *(m + 1) * (n + 1)*, де m і n - довжини порівнюваних послідовностей символів. Окрім цього вважається, що вартість операцій вилучення, заміни однакова. Для побудови матриці використовують таке рекурсивне рівняння:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \quad (5)$$

$\operatorname{lev}_{a,b}(i,j)$ - це відстань між першими i символами послідовності a та першими j символами послідовності b.

Для відстані Левенштейна існують такі верхня і нижня межі:

- Дистанція Левенштейна не є меншою за різницю довжини рядків, що порівнюються
- Вона не є більшою довжини найдовшого рядка
- Вона дорівнює 0 тоді і тільки тоді, коли рядки однакові (однакові символи на однакових позиціях)

У псевдокоді алгоритм виглядає так:

```
int levenshteinDistance(char string1[1..lenStr1], char string2[1..lenStr2])
    // d таблиця кількість рядків = lenStr1+1 та кількість стовпців = lenStr2+1
   declare int d[0..lenStr1, 0..lenStr2]
    // i та j використовуються для індексування позиції у string1 та у string2
   declare int i, j, cost

   for i from 0 to lenStr1
       d[i, 0] := i
   for j from 0 to lenStr2
       d[0, j] := j

   for i from 1 to lenStr1
       for j from 1 to lenStr2
           if string1[i] = string2[j] then cost := 0       //однакові
                                else cost := 1       //заміна
           d[i, j] := minimum(
                                d[i-1, j  ] + 1,     // вилучення
                                d[i  , j-1] + 1,     // вставка
                                d[i-1, j-1] + cost   // заміна або
                           )                           однакові

   return d[lenStr1, lenStr2] //значення відстані Левенштайна в останній клітинці матриці
```

Цей алгоритм можна реалізувати, заповнивши таблицю. Наприклад, для визначення відстані між словами *корабель* і *бал* таблиця виглядатиме так:

		К	О	Р	А	Б	Е	Л	Ь
	0	1	2	3	4	5	6	7	8
Б	1	1	2	3	4	4	5	6	7
А	2	2	2	3	3	4	5	6	7
Л	3	3	3	3	4	4	5	5	6

Таблиця 3 - Обчислення відстані Левенштейна

Тобто відстань:

між пустим словом і словом КОРАБЕЛЬ = 8 (довжина слова КОРАБЕЛЬ)

між Б і КОРАБЕЛЬ відстань = 7 (літера Б в обох словах і може бути використана)

між БА і КОРАБЕЛЬ відстань = 7 (лише одну з літер Б або А можна використати)

між БАЛ і КОРАБЕЛЬ відстань = 6 (можна використати дві літери (Б або А) + Л)

Отже, для того, щоб знайти кандидатів на виправлення слова з помилкою, потрібно розрахувати відстань Левенштейна для неправильного слова та кожного слова в словнику. Потім взяти тих кандидатів, які мають найменшу відстань. Також можна відсортувати їх по частоті використання в мові за допомогою n-грамної моделі ймовірностей.

2.2 Метод подібності ключів

Поняття методу подібності ключів полягає в тому, щоб кожен рядок перетворити на ключ таким чином, щоб схожі рядки мали однакові або подібні ключі. Таким чином, коли ключ обчислюється за помилковим рядком, він буде вказувати на всі слова у словнику, які схожі на помилкове (кандидати).

Метод ключа подібності має перевагу в швидкості, тому що не обов'язково безпосередньо порівнювати неправильно написані рядки з кожним словом у словнику.

Існує два алгоритми:

1. Soundex алгоритм.
2. SPEEDCOP алгоритм.

Soundex - фонетичний алгоритм для індексування слів на основі їх фонетичного звуку. Слова, які однаково вимовляються, але мають різне значення, кодуються аналогічно, щоб їх можна було порівняти незалежно від тривіальних відмінностей у їх написанні [9].

Алгоритм встановлює однакове представлення омофонів, що спрощує їх пошук, не дивлячись на неточності в написанні. Алгоритм загалом кодує приголосні звуки, голосні опускаються, крім першої букви. Soundex — найвідоміший зі всіх фонетичних алгоритмів та часто використовується як синонім до «фонетичного алгоритму». Удосконалення Soundex є основою для багатьох сучасних фонетичних алгоритмів.

Саундекс розробили Роберт Руссел і Маргарет Оделл і запатентували у 1918 і 1922 роках. Так званий американський саундекс, використовувався в 1930-х роках для ретроспективного аналізу переписів населення США від 1890 до 1920 року.

Нижче описано американський Саундекс:

Саудекс-код складається з букви й трьох числових розрядів: першу літеру імені й цифри кодування наступних приголосних. Подібні приголосні мають одні й ті ж цифри, так, наприклад, губні приголосні B, F, P, V кодуються номером 1. Голосні можуть вплинути на кодування, але не кодуються, окрім першої літери.

Правильне значення може бути знайдено так чином:

1. Перша літера імені вводиться безпосередньо.
2. Кожна приголосна має свій код:

   - b, f, p, v → 1
   - c, g, j, k, q, s, x, z → 2
   - d, t → 3
   - l → 4
   - m, n → 5
   - r → 6
   - h, w не кодуються.

3. Дві сусідні літери з однаковим числом кодуються як одне. Літери з тим же числом, розділених h або w також кодуються як одне число.
4. Продовжуєте поки не має одної букви і трьох цифер.

Використання цього алгоритму з «Robert» і «Rupert» повернене рядок «R163», а «Rubin» дає «R150». «Ashcraft» і «Ashcroft» дає на виході «A261».

SPEEDCOP система - це спосіб автоматичного виправлення орфографічних помилок - переважно набір помилок в дуже великій базі даних. Для кожного слова в словнику було обчислено ключ. Він складається з першої літери, а потім приголосних букв слова в порядку їх виникнення у слові, а потім і гласних букв, також у порядку їх виникнення, причому кожна буква записана лише один раз.

2.3 Метод правил

Методи, засновані на правилах, - це алгоритми, які намагаються представляти знання про найпоширеніші помилки орфографії у вигляді правил перетворення помилкових слів на дійсні слова. Процес створення кандидатів полягає у застосуванні всіх правил до неправильно написаного рядка та збереженні кожного отриманого слова, якщо воно існує в словнику.

2.4 Метод n-грам

Символьні n-грами, в тому числі триграми, біграми і унігри, були використані різними способами при розпізнаванні тексту та методах корекції орфографії. Вони були використані в коректорах OCR, щоб зафіксувати лексичний синтаксис словника та запропонувати правильні виправлення. Вони використовувались в корекціях правопису як ключі доступу до словника для визначення корекції кандидата та як лексичні функції для обчислення міри подібності. Вони також використовувались для репрезентування слів та орфографічних помилок як векторів лексичних ознак, до яких можна застосувати традиційні та нові векторні відстані для визначення правильних кандидатів. Деякі методи корекції на основі n-грам виконують процеси виявлення помилок, пошуку кандидатів та класифікації подібності як три окремі етапи.

2.5 Метод оцінки ймовірності

Методи, засновані на n-грамах, привели природним чином до ймовірнісних методів у парадигмах розпізнавання тексту та корекції орфографії. Було використано два типи ймовірностей: ймовірності переходу та ймовірності плутанини. Перехідні імовірності представляють ймовірності того, що дана літера (або послідовність букв) буде супроводжуватися іншою заданою буквою (або послідовністю букв). Перехідні імовірності залежать від мови. Їх іноді називають ймовірностями Маркова. Їх можна розрахувати шляхом збору n-грамних частот на великому корпусі тексту.

Ймовірності плутанини - це оцінки того, як часто помилково вводять дану літеру чи замінюють її на іншу літеру. Ймовірності плутанини залежать від джерела. Оскільки різні пристрої OCR використовують різні методи та функції для розпізнавання символів, кожен пристрій матиме унікальний розподіл ймовірностей плутанини. Той же пристрій оптичного розпізнавання може генерувати різні розбіжності ймовірностей для різних типів шрифтів і їх розмірів точок.

Модель ймовірності плутанини для конкретного пристрою може бути розрахована шляхом подання пристрою вибіркової статистики помилок з тексту.

Цей процес іноді називають фазою "тренування", коли результати використовуються при розробці пост-процесора, що коректує помилки.

2.6 Метод нейронних мереж

Нейронні мережі - це кандидати на орфографічні коректори, бо вони можуть навчатися на фактичних орфографічних помилках, вони можуть адаптуватися до конкретних моделях помилок які роблять люди, таким чином максимізуючи їх точність корекції.

Метод зворотнього поширення помилки (англ. backpropagation) — метод навчання багатошарового перцептрону. Це ітеративний градієнтний алгоритм, який використовується з метою мінімізації помилки роботи багатошарового перцептрону та отримання бажаного виходу. Основна ідея цього методу полягає в поширенні сигналів помилки від виходів мережі до її входів, в напрямку, зворотному прямому поширенню сигналів у звичайному режимі роботи [10].

РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Система перевірки орфографії — це допоміжна програма, яка шукає в тексті документа слова, написані неправильно. Знайдені помилки позначаються спеціальним чином — зазвичай для цього використовується червоне підкреслення. Також користувачеві окрім зазначення місць можливих помилок, надається можливість вибрати один із правильних варіантів написання.

Системи перевірки правопису можуть бути незалежні (як правильно, у такому разі передбачено можливість інтеграції з іншими програмами) або входити як окремий модуль до складу іншої програми, зокрема текстового процесора, поштового клієнта, електронного словника, пошукової системи тощо.

Зазвичай система перевірки орфографії виконує такі дії:

1. Зчитує текст і виокремлює слова, з яких він складається.
2. Порівнює кожне слово тексту зі списком правильно написаних слів (тобто словником). Такий список, окрім власне слів, може містити додаткову інформацію, як-от місця, де може бути знак переносу, лексичні й граматичні атрибути тощо.

Як додаток до цих компонентів, інтерфейс програм дає користувачам змогу згоджуватися або відмовлятись від запропонованих замін і змінювати спосіб роботи програми.

Перші системи із функціями перевірки орфографії з'явилися в 1957 р. — зокрема системи перевірки орфографії для точкових зображень курсивного письма та спеціальні програми, які замість помилкових записів шукали записи в базах даних.

Перші системи перевірки орфографії для персональних комп'ютерів з'явилися в 1980 р. на комп'ютерах CP/M та TRS-80. Невдовзі після цього, в 1981 р., було створено орфографічні пакети для IBM PC. Чимало розробників, зокрема Марія Маріані (*Maria Mariani*), Random House, Soft-Art, Microlytics, Proximity, Circle Noetics та Reference Software, поширювали комплекти OEM-програм та продукти для кінцевих користувачів на ринку програмного забезпечення, який швидко зростав, — здебільшого для персональних комп'ютерів (PC), але також і для Apple Macintosh, VAX та UNIX. На персональних комп'ютерах ці системи перевірки орфографії працювали автономно; більшість із них за наявності достатньої пам'яті могли працювати як резидентні програми (TSR) в комплектах програм для обробки текстів на PC.

Утім, ці програми недовго залишалися на ринку окремими програмами: у середині 1980-х рр. розробники популярних текстових редакторів, як-от WordStar і WordPerfect, вбудували системи перевірки орфографії, здебільшого за ліцензіями описаних вище компаній, у розроблювані ними пакети програм. Невдовзі ці системи почали підтримувати не тільки англійську, а й інші європейські, а згодом навіть і азійські мови [11].

3.1 Опис програмного забезпечення

Було створено програму, яка на вхід отримує слово, перевіряє його на правильність і у разі некоректності, на виході передає правильну версію слова.

Програма виправляє наразі слова англійської мови.

Робота програми поділяється на два процеси:

1. Виявлення помилки.
2. Знаходження правильного кандидита на заміну.

Для перевірки на правильність будо використано метод n-грам. Так як програма працює з окремими словами, будемо використати модель уніграм. Зберігати цю модель модель у вигляді хеш-таблиці, де ключ - це слово, а значення ключа - ймовірність слова.

Для створення моделі уніграм розроблено окремий програмний модуль, який на вхід отримує тренувальний корпус, розбиває його на суміжні уніграми, рахує їх частоти та ймовірності, а на виході передає модель мови у вигляді файлу в форматі JSON.

JSON (англ. JavaScript Object Notation) — це текстовий формат даних для обміну між комп'ютерами. JSON сформований таким чином, що він може бути прочитаний людиною. Формат дозволяє описувати об'єкти та інші структури даних.

JSON будується на двох структурах:

- Набір пар назва/значення. У різних мовах це реалізовано як об'єкт, запис, структура, словник, хеш-таблиця, список з ключем або асоціативним масивом.

- Впорядкований список значень. У багатьох мовах це реалізовано як масив, вектор, список, або послідовність.

Приклад моделі:

```
{
	«the» : 72300,
	«of» : 39713,
	«and» : 36933,
	«to» : 28257,
	«a» : 20169,
	«in» : 19966,
	«he» : 18331,
	«that» : 11976,
	«was» : 11366,
	«his» : 10429,
	«is» : 10052
}
```

Для тренування моделі використовувався текстовий корпус розміром в 1000000 слів.

Отже, для перевірки слова на правильність, воно як ключ шукається в моделі, репрезентованій хеш-таблицею. Якщо такий ключ відсутній, то слово вважається неправильним і переходить до процесу виправлення.

Для виправлення помилки використовується метод відстані Левенштейна. Але використовується не сам алгоритм пошуку відстаней для всіх слів моделі, бо це дуже повільний процес. Використовується ідея цього алгоритму і шукаються всі слова на відстані 1 від помидкового. Цей процес набагато швидший і ефективніший.

Для кожного знайденого слова проводимо перевірку наявності його в моделі уніграм. Після фільтрації залишаються реальні кандидати на виправлення слова. З данної вибірки слів беремо одне слово з найбільшою частотою використання в мові (тобто слово з найбільшим значенням ключа в хеш-таблиці). Це і буде результат.

У разі, якщо після фільтрації кількість кандидатів дорівнює 0, то це означає, що потрібно шукати слова на відстані 2 і так далі. І кожен раз проводити фільтрацію.

Також було створено окремий фреймворк для автоматичного закінчення слів на основі вводу з помилками.

Користувач починає набирати слово і програма видає список найвирогідніших слів, що починаються з введеного префікса. Якщо при написані початку слова користувач допустив помилку, програма автоматично виправляє її і всерівно видає релевантний список.

Для розробки даного фреймворку використовувалися методи n-грам, ідея методу відстані Левенштейна та префіксне дерево.

Префіксне дерево використовується для зберігання словника мови (тобто всіх правильних слів). Також воно потрібне для зберігання одночасно всіх уніграм данної мови (частота кожної уніграми зберігається в нодах дерева, які є кінцевими для кожного слова). За допомогою дерева будемо також шукати всі слова, які починаються на деякий префікс. Отже, префіксне дерево в даній програмі бере на себе сразу декілька обов’язків.

Алгоритм роботи фреймворку наступний:

1. Спочатку подається на вхід початок слова, який користувач ввів. Будемо називати його префіксом слова.
2. Перевіряємо наявність префікса в дереві. Якщо префікс присутній, то просто вилучаємо з дерева всі слова, які починаються на даний префікс і виводимо їх список. Якщо префікс відсутній, то значить він помилковий.
3. Для неправильного префікса знаходимо всі префікси на відстані один.
4. Починаємо перевіряти кожен знайдений префікс на наявність його у дереві.
5. Для першого знайденого префікса знаходимо всі відповідні слова і виводимо їх список. Якщо всі префікси відсутні в дереві повторюємо крок 3, але на відстані два, і так далі, поки не буде знайдено присутній префікс.

Кожний раз коли виводиться список рекомендованих слів, він формується за допомогою частот використання отриманих слів у мові, які зберігаються у моделі уніграм. Отже, показується, наприклад, десять перших найбільш популярних слів.

3.2 Реалізація програмного забезпечення

Для написання модуля тренування моделі мови та основної програми використовувалася мова Swift 5. Розробка проводилася у інтегрованому середовищі розробки — Xcode 12.

Код модуля тренування моделі:

```swift
private var knownWords = [String : Int]()

private func train(_ word: String) {
    if let count = knownWords[word] {
        knownWords[word] = count + 1
    } else {
        knownWords[word] = 1
    }
}

init?(trainingText path: String) {
    do {
        let text = try String(contentsOf: URL(fileURLWithPath: path),
            encoding: .utf8)
        let words = parseWords(text: text)
        for word in words { train(word) }
    } catch {
        return nil
    }
}

private func parseWords(text: String!) -> [String] {
    do {
        let regex = try NSRegularExpression(pattern: "[\\w-[\\d_]]+", options: [])
        let nsString = text.lowercased() as NSString
        let results = regex.matches(in: text.lowercased(), options: [], range:
            NSMakeRange(0, nsString.length))
        return results.map { nsString.substring(with: $0.range)}
    } catch {
        return []
    }
}
```

Методи знаходження слів на відстані 1 та 2 від неправильного слова:

```
private func wordsOnEditDistanceOne(from word: String) -> Set<String> {
    if word.isEmpty { return [] }

    let splits = word.indices.map {
        (word[..<$0], word[$0...])
    }
    let deletes: [String] = splits.map {
        String($0.0 + $0.1.dropFirst())
    }

    let transposes: [String] = splits.map { left, right in
        if let first = right.first {
            let drop1 = right.dropFirst()
            if let second = drop1.first {
                let drop2 = drop1.dropFirst()
                return "\(left)\(second)\(first)\(drop2)"
            }
        }
        return ""
    }.filter { !$0.isEmpty }

    let alphabet = "abcdefghijklmnopqrstuvwxyz"

    let replaces: [String] = splits.flatMap { left, right in
        alphabet.map { "\(left)\($0)\(right.dropFirst())" }
    }
    let inserts: [String] = splits.flatMap { left, right in
        alphabet.map { "\(left)\($0)\(right)" }
    }

    return Set(deletes + transposes + replaces + inserts)
}
private func wordsOnEditDistanceTwo(from word: String) -> Set<String>? {
    var edits = Set<String>()
    for word in wordsOnEditDistanceOne(from: word) {
        if let candidats = getCandidats(wordsOnEditDistanceOne(from: word)) {
            edits = edits.union(candidats)
        }
    }
    return edits.isEmpty ? nil : edits
}
```

Головний метод та метод фільтрації кандидатів:

```
private func getCandidats<S: Sequence>(_ words: S) -> Set<String>?
    where S.Element == String {
    let candidats = Set(words.filter {
        self.knownWords.index(forKey: $0) != nil
    })
    return candidats.isEmpty ? nil : candidats
}

func correct(word: String) -> String {
    let candidates = getCandidats([word]) ??
        getCandidats(wordsOnEditDistanceOne(from: word)) ??
        wordsOnEditDistanceTwo(from: word)

    if candidates == nil { return "" }

    return (candidates ?? []).reduce(word) {
        (knownWords[$0] ?? 1) < (knownWords[$1] ?? 1) ? $1 : $0
    }
}
```

Вікно робочої програми:

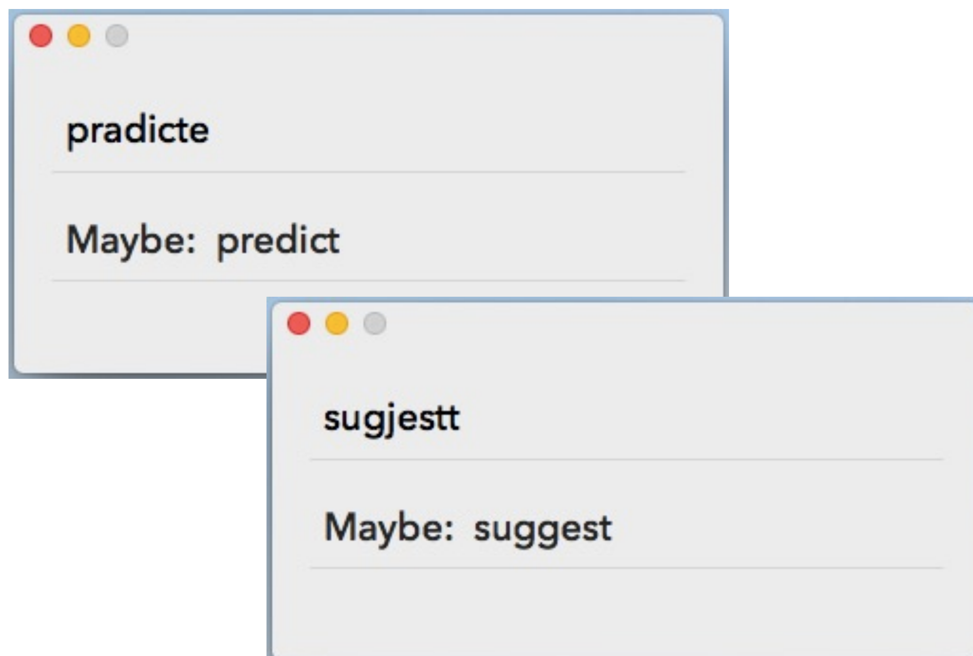
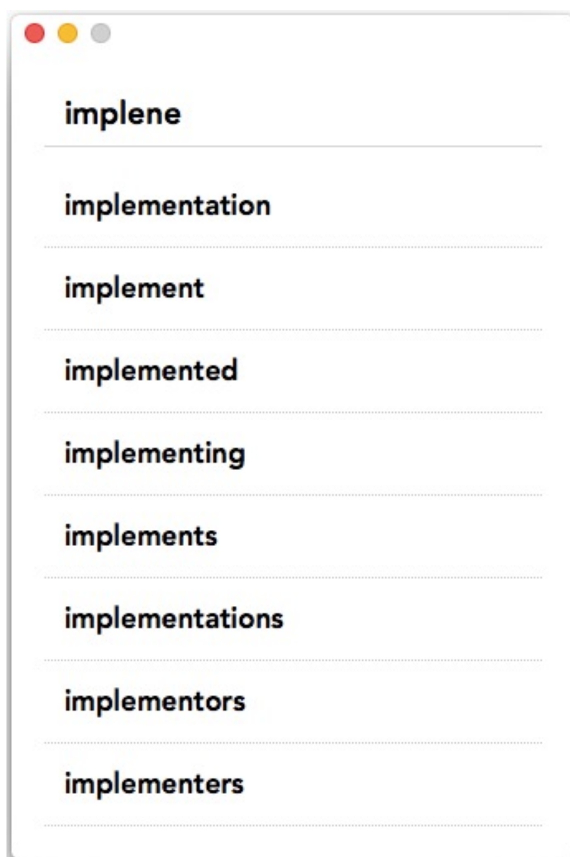


Рисунок 3 - Вікна робочої програми

Реалізація алгоритму роботи фреймворку:

```
public func getWords(with prefix: String) -> [String] {
    let trimPrefix = prefix.trimmingCharacters(in: .whitespaces)

    if trimPrefix.isEmpty { return [] }

    func croppedArray<T>(from array: [T]) -> [T] {
        let croppedArray: [T]

        if array.count > 10 {
            croppedArray = Array(array[0...9])
        } else {
            croppedArray = array
        }

        return croppedArray
    }

    if let words = trie.findWords(with: trimPrefix) {
        return croppedArray(from: words)
    } else {
        let editedPrefixes = wordsOnEditDistanceOne(from: trimPrefix)

        for editedPrefix in editedPrefixes {
            if let words = trie.findWords(with: editedPrefix, includePrefix: true) {
                return croppedArray(from: words)
            }
        }

        var edited2Prefixes = Set<String>()

        for word in editedPrefixes {
            let words = wordsOnEditDistanceOne(from: word)
            edited2Prefixes = edited2Prefixes.union(words)
        }

        for editedPrefix in edited2Prefixes {
            if let words = trie.findWords(with: editedPrefix, includePrefix: true) {
                return croppedArray(from: words)
            }
        }
    }

    return []
}
```

Вікно програми з встановленим фреймворком:



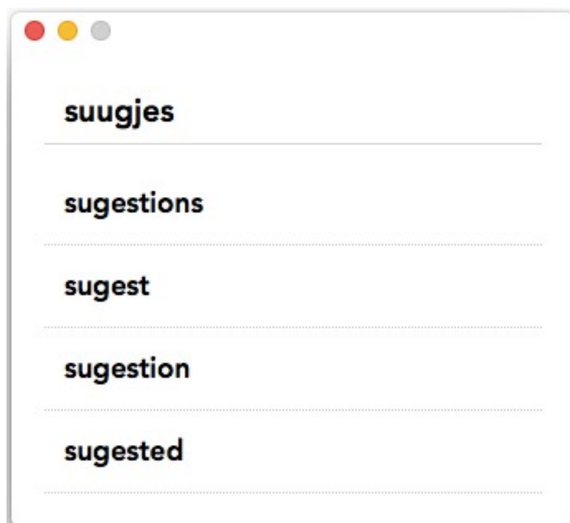


Рисунок 4 - Вікна програми з встановленим фреймворком

ВИСНОВКИ

В ході роботи були розглянуті методи розпізнавання помилок: метод словникового пошуку та метод аналізу n-грам. Та методи виправлення помилок: метод мінімальної відстані, подібності ключів, правил, n-грам, оцінки ймовірності та метод нейронних мереж.

Серед розглянутих методів основними методами, які можна викорисовувати для розробки ефективних систем виправлення помилок, є метод словникового пошуку, метод аналізу n-грам, який пов’язаний з машинним навчанням та метод найменшої відстані.

Була розроблена програма, яка створює модель мови за допомогою тренування на великих текстових збірках. Також, створена програма для випралення помилок в словах англійської мови та фреймворк для автоматичного закінчення слів на основі вводу з помилками.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sidorov, Grigori (2013). "Syntactic Dependency-Based n-grams in Rule Based Automatic English as Second Language Grammar Correction". International Journal of Computational Linguistics and Applications. 4 (2): 169–188.
2. Christopher D. Manning, Hinrich Schütze, Foundations of Statistical Natural Language Processing, MIT Press: 1999.
3. Frederick J. Damerau, Markov Models and Linguistic Theory. Mouton. The Hague, 1971.
4. Figueroa, Alejandro; Atkinson, John (2012). "Contextual Language Models For Ranking Answers To Natural Language Definition Questions". Computational Intelligence. 28 (4): 528–548.
5. Sidorov, Grigori; Velazquez, Francisco; Stamatatos, Efstathios; Gelbukh, Alexander; Chanona-Hernández, Liliana (2012). "Syntactic Dependency-based n-grams as Classification Features". LNAI 7630: 1–11.
6. Sidorov, Grigori; Velasquez, Francisco; Stamatatos, Efstathios; Gelbukh, Alexander; Chanona-Hernández, Liliana. "Syntactic n-Grams as Machine Learning Features for Natural Language Processing". Expert Systems with Applications. 41 (3): 853–860.
7. Ted Dunning (1994). "Statistical Identification of Language". New Mexico State University. Technical Report MCCS 94–273.
8. Yuanhua Lv and ChengXiang Zhai, Positional Language Models for Information Retrieval, in Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval (SIGIR), 2009.
9. "The Soundex Indexing System". National Archives and Records Administration. 2007-05-30. Retrieved 2010-12-24.
10. Nielsen, Michael A. (2015). "Chapter 6". Neural Networks and Deep Learning.
11. Mark Johnson. How the statistical revolution changes (computational) linguistics. Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and Computational Linguistics.