МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

БЕЗДУШНИЙ ВЛАДИСЛАВ ОЛЕГОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

ІНФОРМАЦІЙНА СИСТЕМА ВИЯВЛЕННЯ ПРИХОВАНИХ ЛАЙЛИВИХ СЛІВ У ТЕКСТОВИХ ПОВІДОМЛЕННЯХ НА ОСНОВІ ВІЗУАЛЬНОЇ ПОДІБНОСТІ СИМВОЛІВ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (магістерська) робота

Науковий керівник:
С. Д. Штовба, професор кафедри
інформаційних технологій,
канд. техн. наук, професор
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця — 2024

АНОТАЦІЯ

Бездушний В.О. Інформаційна система виявлення прихованих лайливих слів у текстових повідомленнях на основі візуальної подібності символів. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні технології обробки даних», Донецький національний університет імені Василя Стуса, Вінниця, 2023.

У кваліфікаційній (магістерській) роботі досліджена проблема виявлення прихованих лайливих слів у текстових повідомленнях на основі візуальної подібності символів. Проаналізовані існуючі матриці схожості символів та алгоритми для пошуку прихованих лайливих слів, розроблена система на основі цих даних, що вирішує поставлену задачу.

79 с., 7 табл., 53 рис., 0 дод., 31 джерело.

Ключові слова: прихована лайка, візуальна подібність, ненормативна лексика, Java, Spring, інформаційна система.

ABSTRACT

Bezdushnyi V.O. Information system for detecting the hidden swear words in text messages based on visual similarity of symbols. Specialization 122 "Computer Science", educational program "Data Science", Vasyl' Stus Donetsk National University, Vinnytsia, 2023.

The qualification (master's) thesis investigates the problem of detecting hidden swear words in text messages based on visual similarity of symbols. The existing symbol similarity matrices and algorithms for finding hidden profanity are analyzed, and a system that solves the problem based on these data is developed.

79 p., 7 tab., 53 figures., 0 app., 31 ref.

Keywords: hidden swearing, visual similarity, profanity, Java, Spring, information system.

ЗМІСТ

ВСТУП

Приховані лайливі слова це такі лайливі слова, що зазнали певних замін символів у своїй формі на візуально схожі, зберегли своє значення та залишились легкими для читання і розуміння пересічного користувача. Як приклад можна навести таку просту заміну: fool – f00l. Такі заміни існують, щоб мати змогу обходити автоматичні фільтри повідомлень у чатах, коментарях і у подібних місцях.

Актуальність теми зумовлена тим, що інтернет є неймовірно легко-доступним місцем для майже будь-кого у майже будь-якій точці світу, і всі хто мають до нього доступ також мають і змогу писати будь-що іншим людям, у тому числі і тексти із лайливими словами, що є неприємним досвідом для інших і часто такі тести є забороненими і їх намагаються не пропускати взагалі, або швидко видаляти, і популярним способом обману фільтрів які мають запобігти появленню подібних текстів є заміна літер у словах на візуально схожі.

Мета роботи – підвищення комфортності діяльності в онлайнових мережах за рахунок автоматичного виявлення токсичного контенту.

Об'єктом дослідження є виявлення токсичного контенту в текстових повідомленнях.

Предмет дослідження – сервіси для виявлення лайки, у тому числі й прихованої, у текстових повідомленнях.

Новизна наукових результатів:

1) розроблена матриця візуального сплутування цифр та символів англійської абетки, новизна якої полягає в агрегуванні 5 відомих раніше матриць та їх доповнення відсутніми коефіцієнтами сплутувань для букв

верхнього та нижнього регістрів та коефіцієнтами сплутувань цифр з буквами;

2) реалізовано бектрекінг-алгоритм для виявлення прихованих лайливих слів в текстових повідомленнях з перевіркою на навмисні заміни символів на візуально подібні, що відрізняє його від інших алгоритмів на основі посимвольного порівняння за словником непристойної лексики.

Практична цінність запропонованих результатів полягає в тому, що за першим результатом для кожного символа створюється короткий список найбільш схожих символів, що значно пришвидшує перевірку. Практична цінність другого результату полягає в тому, що він дозволяє виявити спотворені лайливі слова, які пропускають інші алгоритми. Окрім того, запропонований алгоритм базується на аналітичному підході, а не на індуктивному за яким створюють моделі за датасетами. Тому, запропонований алгоритм здатен виявляти не лише відомі спотворені лайливі слова, але і нові, які будуть створені інформаційними вандалами в майбутньому на основі нових варіантів заміни символів.

Достовірність результатів підтверджена тестуванням на 4 синтетичних реченнях, а також перевіркою на 66 тисячах реальних коментарів з датасету Kaggle Toxic Comments.

За результатами роботи опубліковано тези доповіді [31].

РОЗДІЛ 1

АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз фільтру ненормативної лексики як об'єкту дослідження

Текстові повідомлення, або просто повідомлення, це письмове повідомлення, що містить складається з слів, здебільшого в скороченій формі, і що зазвичай надсилається з одного пристрою(телефону, персональному комп’ютеру, тощо) на інший.

Текстові повідомлення мають відносно недовгу історію, починаючи з електричних телеграфів, що були винайдені у 1837 році. Телеграфи були першим пристроєм, що був здатен електронно надіслати текст, проте його технічні характеристики накладали дуже багато обмежень. Так, перша телеграма змогла подолати лише приблизно 3 кілометри, що є надзвичайно малою цифрою станом на сьогодні. Проте станом на 1844 Семюел Морзе спромігся збільшити цю дистанцію до 70 кілометрів [1].

Хоча з того часу технологія телеграфу і мала свій розвиток, проте перший концепт СМС (англ. SMS – Short Message Service) з'явився лише у 1984 році у Франко-Німецькому співробітництву у сфері GSM (фр. Groupe Spécial Mobile) завдяки Фрідгельму Гіллебранду та Бернарду Гіллеберту. Перші СМС повідомлення мали ліміт у 160 символів, що є досить великим лімітом, здатним покрити більшість можливих повідомлень [2].

З того часу технологія вже розвивалась активніше, і набувала все більшого і більшого значення у житті пересічної людини. Станом на сьогодні текстові повідомлення будь якого типу, на кшталт звичайних СМС, повідомлень у різноманітних месенджерах, повідомлення на форумах, чати під трансляціями та у відео-іграх, тощо.

З розвитком і поширенням будь-якої технології вона однозначно потрапить до рук людей, які не будуть хотіти використовувати її за призначенням, або керуватись будь-яким суспільними нормами під час використання. Таким чином, враховуючи розвиток інтернет-технологій і ступінь їх поширення, з'явилась потреба для хоча б мінімального контролю інтернету та того, що там пишуть. Сьогодні будь-який інтернет-ресурс який надає змогу будь-кому писати будь-що в тій чи інші мірі може піддатися атаці від людей, які з тієї чи іншої причини не бажають приймати суспільні норми та правила поведінки, і шукають свободи від цього на просторах всесвітнього павутиння. Деякі сервіси дозволяють таким займатись і ніяк не перешкоджають будь якій формі лексики, базуючись на своїх ідеологічних підґрунтях, або ж з метою забезпечити максимальну анонімність та безпеку. Це є цілком прийнятним і зрозумілим, і завжди таким буде, проте є й такі ресурси які не вбачають у цьому нічого доброго і встановлюють правила доброзичливого спілкування між учасниками спільноти. Тому для таких ресурсів, щоб спростити роботу модераторам, були розроблені та застосовані фільтри ненормативної лексики, що допомагають виявляти відповідні слова або ж тексти та блокувати їх.

Ненормативна лексика - це використання мови, яке іноді вважається грубим, непристойним або культурно образливим [20]. Найпопулярнішими прикладами таких слів можна виділити наступні: «shit», «fuck», «bastard» [21]. Проблема ненормативної лексики у інтернеті може здаватись дріб'язковою, і пов'язаною радше зі звичайними правилами культури, аніж з чимось дійсно серйозним. Проте проблема агресивних та вороже налаштованих людей у інтернеті є досить великою, і ці люди часто займаються в інтернеті так званим «кібербулінгом», що навіть є протизаконним у деяких країнах світу.

Кібербулінг - це дія, яка завдає психічної та матеріальної шкоди іншим людям шляхом неодноразового використання ворожих висловлювань, таких як текст, зображення та голоси, в Інтернеті за допомогою ІТ-пристроїв, включаючи комп'ютери [3]. Приклади кібербулінгу наведені на рис. 1.1.

Line	Role	Message	Bully	Type
1	V	me and my friends hanging out tonight! :)		Neutral
2	B	@V lol b*tch, you dont have any friends.. ur fake as sh*t	✔	Curse, insult
3	AB	@B haha word, shes so sad	✔	Encouragement
4	VF	@V you know it girl		
5	S	@V dont listen to @B, were gonna have fun for sure!		Defense
6	V	@B shut up @B!! nobody asked your opinion!!!!		Defense
7	A	@B you are a f*cking bully, go outside or smt		Insult
8	B	@V @S haha you all so dumb, just kill yourself already!	✔	Insult, curse
9	A, R	@B shut up or ill report you		
10	B	@A u gonna cry? go ahead, see what happens tomorrow!	✔	Threat

Рисунок 1.1 – Приклад кібербулінгу та класифікації текстових повідомлень [22]

Останнім часом вік тих, хто цим займається поступово зменшується через поширення смарт-пристроїв. Такі люди часто не усвідомлюють, що кібербулінг може привести до негативних наслідків, і поводяться необачно, вважаючи його простою грою [4].

Через це й неодноразово підіймалось питання розробки та впровадження систем-фільтрів, що зможе більш-менш точно виявляти агресивні тексти в якихось заданих рамках з подальшим накладенням різноманітних санкцій на користувача, що дозволив собі такі прикру необачність.

Хоча подібна система, та й взагалі жодна інша, не зможе повністю врятувати людину від потенційної небезпеки, проте має за мету максимально пом’якшити ефект від зіткнення з цією проблемою, та просто з фактом того,

що ненормативної лексики у інтернеті дуже й дуже велика кількість. Настільки багато, що саме вербальні образи є найбільшою формою кібербулінга із існуючих [5].

Найчастіше фільтри ненормативної лексики можна зустріти на тих ресурсах, де розповсюджені невеликі повідомлення, а не величезні тексти на аналіз яких може піти дуже багато часу та ресурсів. Наприклад, це чати на стрімінгових платформах таких як Twitch, хоча на ній роль фільтру здебільшого виконують модератори, себто звичайні люди, також коментарі, наприклад коментарі у профілі Steam, а також дуже часто зустрічаються у онлайн іграх, особливо сесійних де відбувається багато коротких матчів за невеликий проміжок часу, і де фізично немає часу писати великі тексти.

Як правило, існуючі фільтри досить примітивні і працюють зі список, радше словником, слів, що є забороненими для використання, проте подібні системи легко обходяться простою заміною символа(ів) у слові на інший, що є візуально схожим на свого попередника, аби залишити зміст слова незмінним і при цьому мати початкове значення.

1.2 Огляд аналогів

Незважаючи на те, скільки найрізноманітніших сервісів існує у мережі інтернет, доступних сервісів для перевірки тексті на наявність ненормативної лексики відносно небагато. Під доступністю мається на увазі те, що сервіс або є безкоштовним або ж має період безкоштовної пробної версії, тому пошук конкурентів дещо ускладняється. Ба більше, навіть ті які надають можливість їх безкоштовної перевірки зазвичай є із закритим вихідним кодом, що означає неможливість взнати на базі яких алгоритмів побудований той чи інший сервіс.

Для аналізу існуючих рішень були обрані сервіси які є у числі перших у пошуковій видачі Google, себто є більш оптимізованими для пошукових

запитів та ймовірно більше популярними, що означає більш якісними, та мають можливість їх безкоштовної перевірки. Серед сервісів, що відповідають цим стандартам були взяті наступні: API Ninjas – Profanity Filter API, Readable.com – Profanity Detector, Sightengine – Text Moderation API, Webpurify – Profanity Filter, PurgoMalum — Free Profanity Filter Web Service.

Для перевірки сервісів було створений текст, що складається здебільшого із лайливих слів. Букви у словах у тексті будуть поступово підмінятись, залежно від того наскільки добре той чи інший сервіс опрацьовує цей текст. Початковий текст є сталим для усіх сервісів.

Структура перевірки того чи іншого сервісу буде складатись із запиту, відповіді та аналізу успішності фільтрації тексту у цій відповіді.

- API Ninjas

Перший сервіс це, у своїй суті, велика збірка різноманітних API, один з яких виконує функції фільтрування і тому є цікавим для тестування і подальшого порівняння результатів. Запит і відповідь сервісу наведені на рис. 1.2.

Рисунок 1.2 – Результат сервісу API Ninjas зі звичайним текстом

Як видно із рисунку вище, сервіс не спроможний виявити більшість лайливих слів навіть без заміни символів, найвірогідніше це пов’язаною з дуже малою базою слів.

Наступним кроком є підміна символів у тих словах, які сервіс виявив.

Рисунок 1.3 – Результат сервісу API Ninjas із заміною символів

Як видно з рис. 1.3, сервіс не спроможний виявити навіть найпростішу заміну. Подальшу перевірку можна вважати недоречною.

- Readable.com

Наступним є сервіс який допомагає у форматуванні тексту, щоб зробити його більш доступним для читачів. Зокрема, має можливість виявлення лайки. Перша перевірка наведена на рис. 1.4.

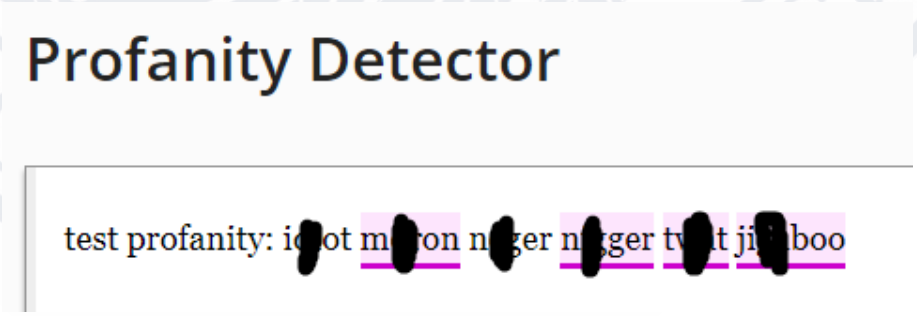


Рисунок 1.4 – Результат сервісу Readable.com зі звичайним текстом

Цей сервіс зміг з набагато кращим результатом виявити ненормативну лексику без заміни символів. Потребується перевірка виявлених слів із заміною символів.

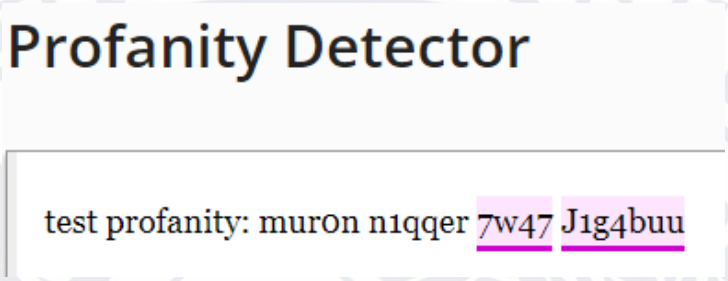


Рисунок 1.5 – Результат сервісу Readable.com із заміною символів

Як видно з рис. 1.7 сервіс досить непогано зміг зрозуміти заміну символів серед тих слів, які він раніше зміг знайти. Хоча у слові 4 ще можливо замінити «g» на «q» і тоді сервіс перестане розпізнавати це слово, але у такому випадку це слово буде майже нечитабельним.

- Sightengine

Наступним є сервіс який спеціалізується на модерації тексту, тому від нього варто очікувати дуже гарних результатів. Початкове тестування наведено на рис. 1.6.

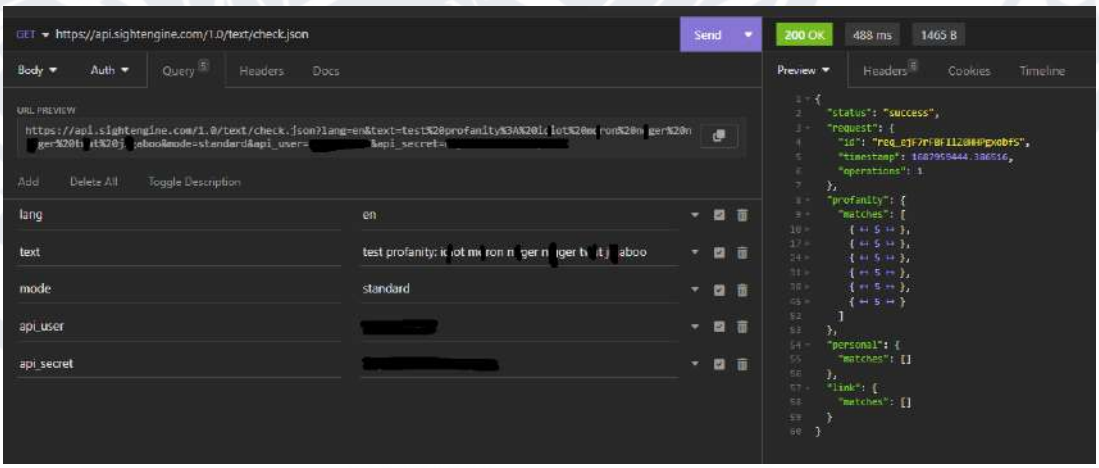


Рисунок 1.6 – Результат сервісу Sightengine зі звичайним текстом

Цей сервіс зміг розпізнати всі 6 слів без заміни символів. Це можна побачити у тому, що масив «profanity.matches» має 6 елементів. На жаль, сервіс не має змоги повернути сам текст із цензуруванням, а лише маю змогу повернути словник знайдених слів з їх короткою характеристикою. Потребується перевірка виявлених слів із заміною символів. Текст був замінений на наступний: «test profanity: 1d1o7 mur0n n3g3r n1qqer 7w47 J1g4buu». Слова піддались тим самим модифікаціям.


```
"profanity": {
  "matches": [
    {
      "type": "insult",
      "intensity": "low",
      "match": "i iot",
      "start": 16,
      "end": 20
    },
    {
      "type": "discriminatory",
      "intensity": "high",
      "match": "n ger",
      "start": 28,
      "end": 32
    },
    {
      "type": "discriminatory",
      "intensity": "high",
      "match": "n kker",
      "start": 34,
      "end": 39
    }
  ]
},
```


Рисунок 1.7 – Результат сервісу Sightengine із заміною символів

Як видно із рис. 1.7, тепер сервіс зміг знайти 3 слова із 6, що є відносно непоганим результатом, кращим ніж у всіх минулих сервісів.

- Webpurify

Даний сервіс також спеціалізується на модерації контенту, у тому числі й тексту, тому як і від минулого, варто очікувати гарних результатів. Результати першої перевірки наведені на рис. 1.8.

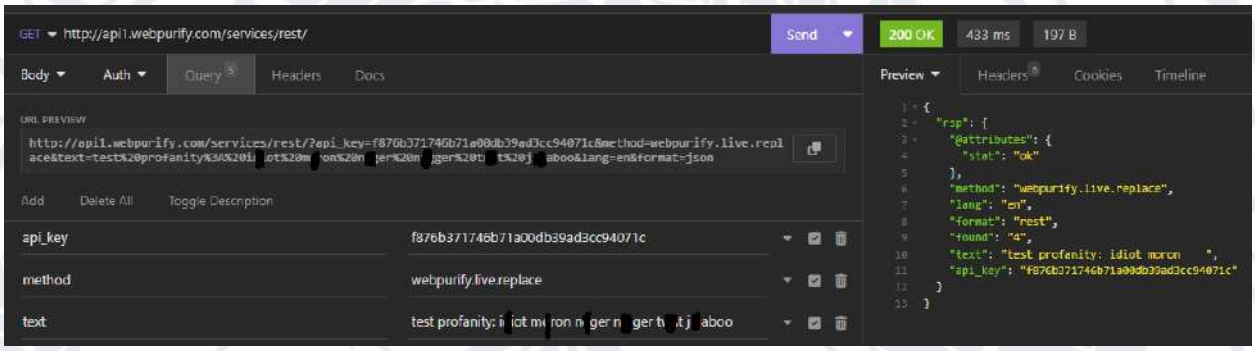


Рисунок 1.8 – Результат сервісу Webpurify зі звичайним текстом

Цей сервіс зміг розпізнати 4 слова із 6 без заміни, що не є загалом поганим результатом, але цей сервіс, з якоїсь причини, деякі слова не розпізнає як лайливі. Проте все одно потрібно протестувати його роботу із заміною символів. Текст, аналогічно минулим перевіркам, був замінений на наступний: «test profanity: n3g3r n1qqer 7w47 J1g4buu».

Рисунок 1.9 – Результат сервісу Webpurify із заміною символів

Як видно з рис. 1.9, сервіс не зміг розпізнати 2 слова із 4, що теж загалом є непоганим варіантом, проте не найкращим серед вже перевірених сервісів.

- PurgoMalum

Цей сервіс позиціонує себе як безкоштовний сервіс для виявлення лайки, що також є цікавим для дослідження. Результат першого запиту наведений на рис. 1.10.

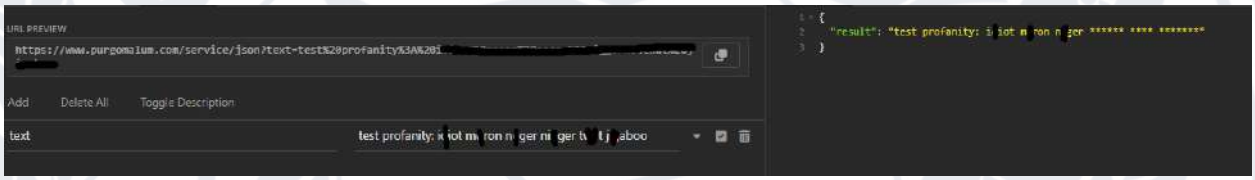


Рисунок 1.10 – Результат сервісу PurgoMalum зі звичайним текстом

Цей сервіс зміг знайти лише половину слів, 3 із 6. Знову ж таки, деякі слова він не сприймає як лайливі, а декі, вочевидь, не бачить через те, що вони відсутні в його словнику, проте останніх 3 було виявлено. Для перевірки заміни символів був наданий наступний текст: «test profanity: n1qqer 7w47 J1g4buu».

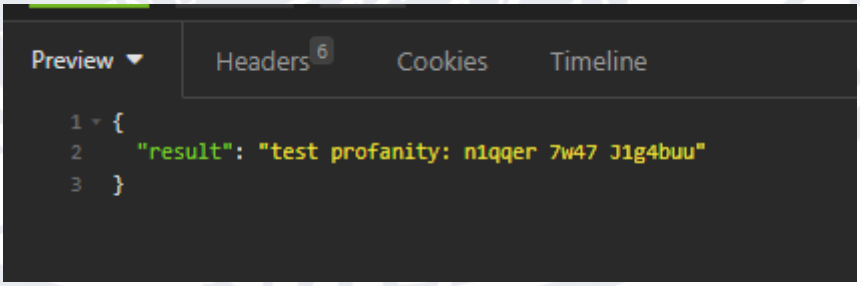


Рисунок 1.11 – Результат сервісу PurgoMalum із заміною символів

Як видно із рис. 1.11, жодного слова не було виявлено, що, вочевидь, означає те, що цей сервіс не можу розпізнавати заміни символів, що є досить поганою характеристикою.

Базуючись на результатах тестувань вище, можна побудувати таблицю в якій всі в більш зрозумілій формі відобразити. В цій таблиці «n» це кількість слів, що були успішно виявлені, «x» це кількість слів, що були передані у другий тестовий запит, цей параметр завжди дорівнює кількості слів виявлених при першому запиті. Слова які не були виявлені при першій перевірці у другій не враховувались, бо відповідна система, очевидно, все одно їх би не знайшла.

Таблиця 1.1 – Порівняння успішності роботи сервісів

	API Ninjas	Readable.com	Sightengine	Webpurify	PurgoMalum
Кількість виявлених слів без замін, n/6	1/6	4/6	6/6	4/6	3/6
Кількість виявлених слів із заміною, n/x	0/1	2/4	3/6	2/4	0/3

Базуючись на тестуванні всіх вище вказаних сервісів і порівнянні їх ефективності, як показано у таблиці 1.1, можна сказати, навіть якщо сервіс має можливість фільтру нецензурної лексики, то із заміною символів у словах цей сервіс, зазвичай, справляється досить посередньо, якщо не погано. Заміна символів є популярним способом обійти фільтр, тому на ній потрібно

акцентувати увагу, і саме на цьому буде акцентована увага при розробці програмного забезпечення під час виконання цієї роботи.

1.3 Специфікація вимог до системи

Враховуючи недоліки всіх перерахованих сервісів, потрібно спроектувати та реалізувати систему, що буде містити покращений алгоритм виявлення лайливих слів, що були модифіковані за допомогою заміни букв на візуально схожі. Від цього можуть найбільше страждати ті ресурси, де треба швидко писати відносно короткі повідомлення, наприклад чати, і де у користувача немає змоги писати комплексні речення, суть яких буде обійти всі можливі обмеження. Наявність алгоритму, що зможе аналізувати такі слова це непогана функція для будь якого фільтру.

Для того, щоб досягти цієї мети потрібно зробити наступне:

- Розробити алгоритм, що зможе «розшифровувати» задані слова базуючись на матриці подібності символів, та шукати отримане слово у словнику ненормативної лексики;
- Розробити систему, що буде вирішувати цю задачу. Система має мати наступний функціонал:
  - Зовнішній інтерфейс, за допомогою якого ця програма зможе взаємодіяти з користувачем, або за допомогою якого вона може бути відключена до існуючих рішень, щоб їх покращити. Система може бути реалізована як і окремий веб-застосунок так і як бібліотека для деякої мови програмування;
  - Система має мати коректні моделі для вхідних і вихідних даних, без цих моделей неможливо нормально працювати

на об’єктно-орієнтованих мовах програмування, бо ці моделі мають містити всі потрібно інформацію, щоб алгоритм мав дані для роботи.

- Система не має витрачати аж занадто багато часу для своєї роботи, бо в такому випадку вона не буде актуальною для роботи.
- Опціональну можливість додатково валідувати вхідні данні, адже вона не має «зламатись» якщо їй передадуть неправильні данні, вона має повернути коректну помилку;
- Опціональну можливість додавати власний словник до все існуючого, адже це надасть системі більшою гнучкості;
- Документацію, адже в ній можна більш детально описати можливості системи та надати приклади роботи;

- Провести тестування алгоритму та системи на наявність проблем з продуктивністю програмного забезпечення а також на наявність проблем із самим алгоритмом і процесом перетворень слова.

Все вище перелічене має бути реалізоване як MVP (англ. Minimum viable product), аби з нею взагалі хоч якось можна було працювати у майбутньому.

Висновок до розділу 1

У розділі була досліджена задача пошуку прихованих лайливих слів на основі візуальної подібності символів. Були проаналізовані деякі існуючі рішення, підбиті підсумки за результатом їх роботи, та були виявленні їхні проблеми, які мають бути усунені в результаті цієї роботи. Були встановлення норми яким має відповідати фінальний програмний продукт.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ ВИЯВЛЕННЯ СПОТВОРЕНИХ СЛІВ

2.1 Синтез матриці сплутувань

Предметом дослідження в рамках цієї роботи є процес підміни символів у словах, щоб слово залишилось легким для читання і не втратило свого початкового змісту.

Такі заміни широко використовувались і використовуються й сьогодні, причому виникли вони із суб-культур, що були популярними коли інтернет тільки починав поширюватись світом. Одна з найбільших суб-культур, що постійно використовувала заміни символів у словах є так звана «l33t». Ця суб-культура виникла десь на початку 80-х років минулого століття, ще коли всесвітньої павутини не існувало, на аналогах сучасних веб-сайтів які використовувались на той час, і які були розбиті на групи «по інтересах». Деякі з таких груп вже тоді цікавились ранніми формами хакерства, або на іншій незаконній діяльності. Такі групи називали себе «елітою» (англ. «elite», пізніше просто «leet»).

Саме тут і з'являється ця «мова» leet. Її зробили як своєрідний шифр, що використовували у інших групах для обговорення тем, що суперечили правилам тієї чи іншої групи. Також він добре працював як спосіб обходу вже тоді існуючих примітивних фільтрів, так як ті розуміли різницю між двома візуальними схожими символами, і пропускали слово, але будь-яка людина одразу розуміла справжній сенс написаного. З часом використання цього правопису стало, фактично, моветоном і здебільшого не асоціюється ні з чим хоча б трохи серйозним [6].

Хоча ця мова вже майже не зустрічається, але окремі її правила все же добре працюють та допомагають у обходах існуючих фільтрів, так наприклад можна замінити букву «і» на «1» або ж «о» на «», що, скоріше за все,

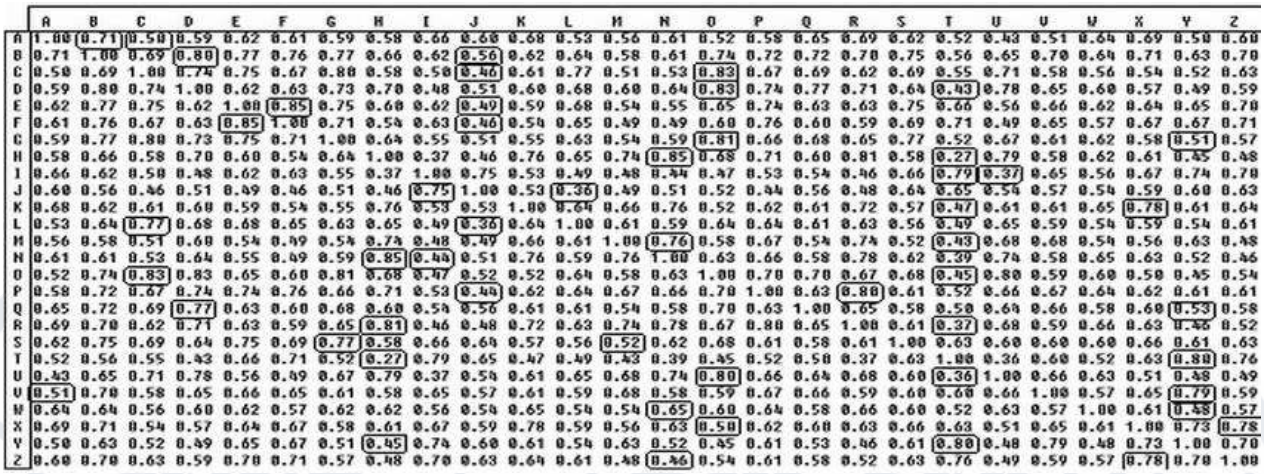
допоможе обманути фільтр але у той самий час залишить слово легким для читання іншим користувачам.

Для кращого розуміння наведемо приклад використання подібних підмін: noob -> n00b. Хоча це слово і не є напряму лайливим, проте є образливим, адже часто вживається, щоб показати, що даний користувач (гравець, у цьому випадку) погано грає і не може повноцінно виконувати свою роль.

Для того, щоб система, яка є результатом цієї роботи, могла коректно обробляти слова в яких відбулась подібно заміна потрібно створити матрицю подібностей символів. За основу була взята англійська абетка, що заснована на латинській абетці, а також всі існуючі цифри: ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz, 1234567890.

Подібні матриці вже були побудовані різними людьми у минулому, хоч відносно нових робіт на цю тематику дуже мала кількість, тому було прийняте рішення здебільшого звертатись до більш старих, але «перевірених» робіт, проте за наявністю брались також і роботи яким не більше 15 років. Для побудови кожною матриці першоджерелам видавався коефіцієнт «довіри» для підрахунку результуючого значення який опирався на сучасність відповідної роботи. Відтак, для аналізу схожості символів верхнього регістру було взято 3 джерела, кожне з яких буде розглянуто окремо і використання для побудовані для власної матриці.

Перше джерело [7] було спрямоване для автоматизованого розпізнавання номерних знаків. Ця робота була опублікована у серпні 2011 року, і відтак є найновішою серед тих, що були використанні, і тому вона має найвищий коефіцієнт, а саме – 0,7.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	1.00	0.71	0.50	0.59	0.62	0.61	0.59	0.58	0.66	0.60	0.68	0.53	0.56	0.61	0.52	0.58	0.65	0.69	0.62	0.52	0.43	0.51	0.64	0.69	0.50	0.60
B	0.71	1.00	0.69	0.80	0.77	0.76	0.77	0.66	0.62	0.56	0.62	0.64	0.58	0.61	0.74	0.72	0.72	0.78	0.75	0.56	0.65	0.70	0.64	0.71	0.63	0.70
C	0.50	0.69	1.00	0.74	0.75	0.67	0.80	0.58	0.50	0.46	0.61	0.77	0.51	0.53	0.83	0.67	0.69	0.62	0.69	0.55	0.71	0.58	0.56	0.54	0.52	0.63
D	0.59	0.80	0.74	1.00	0.62	0.63	0.73	0.70	0.48	0.51	0.60	0.68	0.60	0.64	0.83	0.74	0.77	0.71	0.64	0.43	0.78	0.65	0.60	0.57	0.49	0.59
E	0.62	0.77	0.75	0.62	1.00	0.85	0.75	0.68	0.62	0.49	0.59	0.68	0.54	0.55	0.65	0.74	0.63	0.63	0.75	0.66	0.56	0.66	0.62	0.64	0.65	0.70
F	0.61	0.76	0.67	0.63	0.85	1.00	0.71	0.54	0.63	0.46	0.54	0.65	0.49	0.49	0.60	0.76	0.60	0.59	0.69	0.71	0.49	0.65	0.57	0.67	0.67	0.71
G	0.59	0.77	0.80	0.73	0.75	0.71	1.00	0.64	0.55	0.51	0.55	0.63	0.54	0.59	0.81	0.66	0.68	0.65	0.77	0.52	0.67	0.61	0.62	0.58	0.51	0.57
H	0.58	0.66	0.58	0.70	0.68	0.54	0.64	1.00	0.37	0.46	0.76	0.65	0.74	0.85	0.68	0.71	0.60	0.81	0.58	0.27	0.79	0.58	0.62	0.61	0.45	0.48
I	0.66	0.62	0.50	0.48	0.62	0.63	0.55	0.37	1.00	0.75	0.53	0.49	0.48	0.44	0.47	0.53	0.54	0.46	0.66	0.79	0.37	0.65	0.56	0.67	0.74	0.70
J	0.60	0.56	0.46	0.51	0.49	0.46	0.51	0.46	0.75	1.00	0.53	0.36	0.49	0.51	0.52	0.44	0.56	0.48	0.64	0.65	0.54	0.57	0.54	0.59	0.60	0.63
K	0.68	0.62	0.61	0.68	0.59	0.54	0.55	0.76	0.53	0.53	1.00	0.64	0.66	0.76	0.52	0.62	0.61	0.72	0.57	0.47	0.61	0.61	0.65	0.78	0.61	0.64
L	0.53	0.64	0.77	0.68	0.68	0.65	0.63	0.65	0.49	0.36	0.64	1.00	0.61	0.59	0.64	0.64	0.61	0.63	0.56	0.49	0.65	0.59	0.54	0.59	0.54	0.61
M	0.56	0.58	0.51	0.68	0.54	0.49	0.54	0.74	0.48	0.49	0.66	0.61	1.00	0.76	0.58	0.67	0.54	0.74	0.52	0.43	0.68	0.68	0.54	0.56	0.63	0.48
N	0.61	0.61	0.53	0.64	0.55	0.49	0.59	0.85	0.44	0.51	0.76	0.59	0.76	1.00	0.63	0.66	0.58	0.78	0.62	0.39	0.74	0.58	0.65	0.63	0.52	0.46
O	0.52	0.74	0.83	0.83	0.65	0.60	0.81	0.68	0.47	0.52	0.52	0.64	0.58	0.63	1.00	0.70	0.70	0.67	0.68	0.45	0.80	0.59	0.60	0.50	0.45	0.54
P	0.58	0.72	0.67	0.74	0.74	0.76	0.66	0.71	0.53	0.44	0.62	0.64	0.67	0.66	0.70	1.00	0.63	0.80	0.61	0.52	0.66	0.67	0.64	0.62	0.61	0.61
Q	0.65	0.72	0.69	0.77	0.63	0.60	0.68	0.60	0.54	0.56	0.61	0.61	0.54	0.58	0.70	0.63	1.00	0.65	0.58	0.50	0.64	0.66	0.58	0.60	0.53	0.58
R	0.69	0.78	0.62	0.71	0.63	0.59	0.65	0.81	0.46	0.48	0.72	0.63	0.74	0.78	0.67	0.80	0.65	1.00	0.61	0.37	0.68	0.59	0.66	0.63	0.46	0.52
S	0.62	0.75	0.69	0.64	0.75	0.69	0.77	0.58	0.66	0.64	0.57	0.56	0.52	0.62	0.68	0.61	0.58	0.61	1.00	0.63	0.60	0.60	0.60	0.66	0.61	0.63
T	0.52	0.56	0.55	0.43	0.66	0.71	0.52	0.27	0.79	0.65	0.47	0.49	0.43	0.39	0.45	0.52	0.58	0.37	0.63	1.00	0.36	0.60	0.52	0.63	0.88	0.76
U	0.43	0.65	0.71	0.78	0.56	0.69	0.67	0.79	0.37	0.54	0.61	0.65	0.68	0.74	0.80	0.66	0.64	0.68	0.60	0.36	1.00	0.66	0.63	0.51	0.48	0.49
V	0.51	0.70	0.58	0.65	0.66	0.65	0.61	0.58	0.65	0.57	0.61	0.59	0.68	0.58	0.59	0.67	0.66	0.59	0.60	0.60	0.66	1.00	0.57	0.65	0.79	0.59
W	0.64	0.64	0.56	0.60	0.62	0.57	0.62	0.62	0.56	0.54	0.65	0.54	0.54	0.65	0.60	0.64	0.58	0.66	0.60	0.52	0.63	0.57	1.00	0.61	0.48	0.57
X	0.69	0.71	0.54	0.57	0.64	0.67	0.58	0.61	0.67	0.59	0.78	0.59	0.56	0.63	0.50	0.62	0.68	0.63	0.66	0.63	0.51	0.65	0.61	1.00	0.73	0.78
Y	0.50	0.63	0.52	0.49	0.65	0.67	0.51	0.45	0.74	0.60	0.61	0.54	0.63	0.52	0.45	0.61	0.53	0.46	0.61	0.80	0.48	0.79	0.48	0.73	1.00	0.70
Z	0.60	0.70	0.63	0.59	0.70	0.71	0.57	0.48	0.70	0.63	0.64	0.61	0.48	0.46	0.54	0.61	0.58	0.52	0.63	0.76	0.49	0.59	0.57	0.78	0.70	1.00

Рисунок 2.1 –Матриця подібності від [7]

Наступною роботою є робота [8], у цій роботі автор аналізував схожість символів як візуальну, так і тактильну, що використовується шрифтом Брайля. Перша частина роботи була розглянута, проаналізована і також використана при побудові своєї матриці. Робота була опублікована у 1982 році, ця робота є вже досить старою і тому має коефіцієнт 0,5.

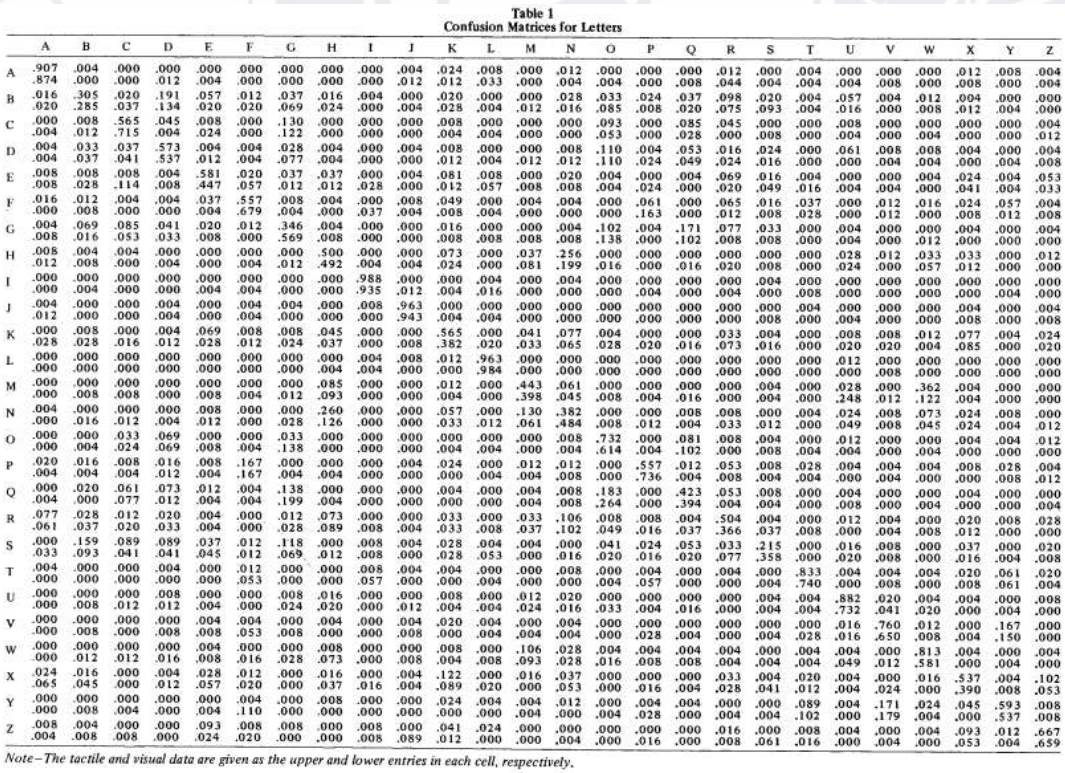


Table 1
Confusion Matrices for Letters

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	.907 .874	.004 .000	.000 .000	.000 .012	.000 .004	.000 .000	.000 .000	.000 .000	.000 .000	.004 .012	.024 .012	.008 .033	.000 .000	.012 .004	.000 .004	.000 .000	.000 .008	.012 .044	.000 .004	.004 .004	.000 .004	.000 .008	.000 .000	.012 .008	.008 .000	.004 .004
B	.016 .020	.305 .285	.020 .037	.191 .134	.057 .020	.012 .020	.037 .069	.016 .024	.004 .000	.000 .004	.020 .028	.000 .004	.000 .012	.028 .016	.033 .085	.024 .008	.037 .020	.098 .075	.020 .093	.004 .004	.057 .016	.004 .000	.012 .008	.004 .012	.000 .004	.000 .000
C	.000 .004	.008 .012	.565 .715	.045 .004	.008 .024	.000 .000	.130 .122	.000 .000	.000 .000	.000 .000	.008 .004	.000 .004	.000 .000	.000 .000	.093 .053	.000 .000	.085 .028	.045 .000	.000 .008	.000 .000	.008 .004	.000 .000	.000 .004	.000 .000	.000 .000	.004 .012
D	.004 .004	.033 .037	.037 .041	.573 .537	.004 .012	.004 .004	.028 .077	.004 .004	.000 .000	.004 .000	.008 .012	.000 .004	.000 .012	.008 .012	.110 .110	.004 .024	.053 .049	.016 .024	.024 .016	.000 .000	.061 .000	.008 .004	.008 .004	.004 .000	.000 .004	.004 .008
E	.008 .008	.008 .028	.008 .114	.004 .008	.581 .447	.020 .057	.037 .012	.037 .012	.000 .028	.004 .000	.081 .000	.008 .012	.000 .057	.020 .008	.004 .004	.000 .000	.004 .020	.069 .049	.016 .016	.004 .004	.000 .004	.000 .000	.004 .004	.024 .041	.004 .004	.053 .033
F	.016 .000	.012 .008	.004 .000	.004 .000	.037 .004	.557 .679	.008 .004	.004 .000	.000 .037	.008 .004	.049 .008	.000 .004	.004 .000	.004 .000	.000 .000	.061 .163	.000 .000	.065 .012	.016 .008	.037 .028	.000 .000	.012 .012	.016 .000	.024 .008	.057 .012	.004 .008
G	.004 .008	.069 .016	.085 .053	.041 .033	.020 .008	.012 .000	.346 .569	.004 .008	.000 .000	.000 .000	.016 .008	.000 .008	.000 .008	.004 .008	.102 .138	.004 .000	.171 .102	.077 .008	.033 .008	.000 .000	.004 .004	.000 .000	.000 .012	.004 .000	.000 .000	.004 .000
H	.008 .012	.004 .008	.004 .000	.000 .004	.000 .000	.000 .004	.000 .012	.500 .492	.000 .004	.000 .004	.073 .024	.000 .000	.037 .081	.256 .199	.000 .016	.000 .000	.000 .016	.000 .020	.000 .008	.000 .000	.028 .024	.012 .000	.033 .057	.033 .012	.000 .000	.012 .000
I	.000 .000	.000 .004	.000 .000	.000 .000	.000 .004	.000 .004	.000 .000	.000 .000	.988 .935	.000 .012	.000 .004	.004 .016	.000 .000	.004 .000	.000 .000	.000 .004	.000 .000	.000 .004	.004 .000	.000 .008	.000 .000	.000 .000	.000 .000	.000 .004	.000 .000	.000 .000
J	.004 .012	.000 .000	.000 .000	.004 .004	.000 .000	.004 .004	.004 .000	.000 .000	.008 .000	.963 .943	.000 .004	.000 .004	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .008	.004 .000	.004 .004	.000 .000	.000 .000	.000 .000	.004 .008	.000 .000	.004 .008
K	.000 .028	.008 .028	.000 .016	.004 .012	.069 .028	.008 .012	.008 .024	.045 .037	.000 .000	.000 .008	.565 .382	.000 .020	.041 .033	.077 .065	.004 .028	.000 .020	.000 .016	.033 .073	.004 .016	.000 .000	.008 .020	.008 .020	.012 .004	.077 .085	.004 .000	.024 .020
L	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .004	.004 .004	.008 .000	.012 .000	.963 .984	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.012 .008	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000
M	.000 .000	.000 .008	.000 .000	.000 .000	.000 .000	.000 .000	.000 .000	.085 .004	.000 .000	.000 .000	.012 .004	.000 .000	.443 .398	.061 .045	.000 .008	.000 .004	.000 .016	.000 .000	.004 .004	.000 .000	.028 .248	.000 .012	.362 .122	.004 .004	.000 .000	.000 .000
N	.004 .000	.000 .016	.000 .012	.000 .004	.008 .012	.000 .000	.000 .028	.260 .126	.000 .000	.000 .000	.057 .033	.000 .012	.130 .061	.382 .484	.000 .008	.000 .012	.008 .004	.008 .033	.000 .012	.004 .000	.024 .049	.008 .008	.073 .045	.024 .024	.008 .004	.000 .012
O	.000 .000	.000 .004	.033 .024	.069 .069	.000 .008	.000 .004	.033 .138	.000 .000	.000 .000	.000 .000	.000 .004	.000 .004	.000 .000	.008 .004	.732 .614	.000 .004	.081 .102	.008 .000	.004 .008	.000 .004	.012 .004	.000 .000	.000 .004	.004 .000	.004 .000	.012 .000
P	.020 .004	.016 .004	.008 .004	.016 .012	.008 .004	.167 .167	.000 .004	.000 .004	.000 .000	.004 .000	.024 .000	.000 .004	.012 .004	.012 .008	.000 .000	.557 .736	.012 .004	.053 .008	.008 .004	.028 .004	.004 .000	.004 .004	.004 .000	.008 .000	.028 .008	.004 .012
Q	.000 .004	.020 .000	.061 .077	.073 .012	.012 .004	.004 .004	.138 .199	.000 .004	.000 .000	.000 .000	.004 .000	.000 .000	.004 .004	.008 .008	.183 .264	.000 .000	.423 .394	.053 .004	.008 .004	.000 .000	.004 .008	.000 .000	.000 .004	.004 .000	.000 .000	.000 .004
R	.077 .061	.028 .037	.012 .020	.020 .033	.004 .004	.000 .000	.012 .028	.073 .089	.000 .008	.000 .004	.033 .033	.000 .008	.033 .037	.106 .102	.008 .049	.008 .016	.004 .037	.504 .366	.004 .037	.000 .008	.012 .000	.004 .004	.000 .008	.020 .012	.008 .000	.028 .000
S	.000 .033	.159 .093	.089 .041	.089 .041	.037 .045	.012 .012	.118 .069	.000 .012	.008 .008	.004 .000	.028 .028	.004 .053	.004 .000	.000 .016	.041 .020	.024 .016	.053 .020	.033 .077	.215 .358	.000 .000	.016 .020	.008 .008	.000 .000	.037 .016	.000 .004	.020 .008
T	.004 .000	.000 .000	.000 .000	.004 .000	.000 .000	.012 .053	.000 .000	.000 .000	.008 .057	.004 .000	.004 .000	.000 .004	.000 .000	.008 .000	.000 .004	.004 .057	.000 .000	.000 .000	.000 .004	.833 .740	.004 .000	.004 .008	.004 .000	.020 .008	.061 .061	.020 .004
U	.000 .000	.000 .008	.000 .012	.008 .012	.000 .004	.000 .000	.008 .024	.016 .020	.000 .000	.000 .012	.008 .004	.000 .004	.012 .024	.020 .016	.000 .033	.000 .004	.000 .016	.000 .000	.004 .004	.004 .004	.882 .732	.020 .041	.004 .020	.004 .000	.000 .004	.008 .000
V	.000 .000	.000 .008	.000 .000	.000 .008	.004 .008	.004 .053	.000 .008	.004 .000	.000 .000	.004 .008	.020 .000	.004 .004	.000 .004	.004 .004	.000 .000	.000 .028	.000 .004	.000 .000	.000 .004	.000 .028	.016 .016	.760 .650	.012 .008	.000 .004	.167 .150	.000 .000
W	.000 .000	.000 .012	.000 .012	.000 .016	.004 .008	.000 .016	.000 .028	.008 .073	.000 .000	.000 .008	.008 .004	.000 .008	.106 .093	.028 .028	.004 .016	.004 .008	.004 .008	.000 .004	.000 .004	.004 .004	.000 .049	.000 .012	.813 .581	.004 .000	.000 .004	.004 .000
X	.024 .065	.016 .045	.000 .000	.004 .012	.028 .057	.012 .020	.000 .000	.016 .037	.000 .016	.004 .004	.122 .089	.000 .020	.016 .000	.037 .053	.000 .000	.000 .016	.000 .004	.033 .028	.004 .041	.020 .012	.004 .004	.000 .024	.016 .000	.537 .390	.004 .008	.102 .053
Y	.000 .000	.000 .008	.000 .004	.000 .000	.000 .004	.004 .110	.000 .000	.008 .000	.000 .000	.000 .000	.024 .000	.004 .000	.004 .004	.012 .000	.000 .004	.004 .028	.004 .000	.000 .004	.000 .004	.089 .102	.004 .000	.171 .179	.024 .004	.045 .000	.593 .537	.008 .008
Z	.008 .004	.004 .008	.000 .008	.000 .000	.093 .024	.008 .020	.008 .000	.000 .008	.008 .089	.000 .012	.041 .000	.024 .000	.000 .000	.000 .004	.000 .000	.000 .016	.000 .000	.016 .008	.000 .061	.008 .016	.004 .000	.000 .004	.004 .000	.000 .053	.093 .004	.012 .667 .659

Note–The tactile and visual data are given as the upper and lower entries in each cell, respectively.

Рисунок 2.2 – Матриця подібності від [8]

Так як ця робота розглядає два різних типи схожості, то потрібно використати потрібний тип, візуальний. Він у матриці має місце нижнього значення у кожній клітинці.

Наступної, і останньою, роботою є [9]. Ця робота вивчає схожість символів з використанням простого шрифту та у тахістоскопічних умовах. Подібна робота є майже ідеальною для поточної, адже вивчає проблематику яка є надзвичайно важливою, і при чому акцентує увагу на тому, що шрифти будуть прості, що також є важливим в сучасних умовах, де є сотні тисяч різних шрифтів і проаналізувати всі не є можливим. Робота опублікована у 1971 році, і є найстаршою, проте за своєю суттю є найближчою до актуальної проблеми і тому має коефіцієнт 0,5.

Stimulus	Response																								
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	.58	.00	.00	.01	.00	.01	.00	.02	.02	.03	.10	.02	.00	.02	.01	.02	.00	.05	.00	.04	.00	.01	.00	.06	.00	.01
B	.02	.26	.02	.03	.02	.00	.04	.07	.01	.02	.01	.01	.03	.08	.03	.03	.03	.18	.01	.01	.06	.01	.00	.01	.00	.00
C	.01	.01	.50	.01	.05	.03	.03	.02	.01	.01	.03	.04	.00	.00	.07	.03	.01	.02	.01	.04	.01	.01	.01	.01	.01	.01
D	.01	.04	.01	.46	.01	.00	.05	.04	.01	.01	.00	.00	.01	.01	.12	.02	.05	.05	.00	.02	.06	.01	.00	.01	.01	.00
E	.01	.03	.03	.01	.36	.07	.00	.03	.03	.04	.06	.11	.00	.00	.03	.05	.00	.01	.00	.07	.01	.01	.00	.01	.03	.01
F	.00	.01	.00	.01	.02	.33	.01	.03	.09	.05	.03	.07	.01	.01	.00	.03	.00	.01	.01	.18	.01	.01	.00	.01	.07	.01
G	.01	.01	.08	.01	.01	.01	.34	.09	.01	.01	.02	.03	.00	.04	.11	.03	.03	.03	.01	.01	.07	.02	.00	.00	.01	.00
H	.01	.01	.01	.01	.00	.01	.01	.50	.01	.01	.01	.01	.03	.15	.03	.02	.00	.04	.00	.03	.04	.01	.03	.00	.01	.00
I	.00	.00	.00	.00	.01	.03	.01	.01	.57	.08	.01	.11	.00	.00	.01	.01	.00	.02	.00	.09	.00	.01	.00	.00	.03	.00
J	.01	.00	.00	.00	.00	.01	.01	.02	.15	.48	.01	.03	.00	.00	.03	.01	.00	.01	.01	.08	.04	.01	.00	.02	.05	.01
K	.03	.01	.01	.01	.01	.02	.00	.06	.05	.01	.50	.04	.01	.02	.04	.03	.00	.03	.00	.01	.01	.01	.01	.05	.03	.00
L	.01	.00	.01	.01	.01	.01	.00	.03	.14	.03	.02	.60	.00	.01	.03	.01	.00	.01	.00	.07	.00	.00	.00	.00	.02	.00
M	.00	.01	.00	.01	.00	.00	.01	.10	.00	.01	.01	.00	.62	.08	.05	.01	.01	.01	.01	.01	.01	.01	.01	.05	.01	.00	.00
N	.03	.01	.00	.01	.00	.00	.01	.06	.00	.00	.03	.00	.07	.54	.03	.01	.03	.04	.01	.01	.02	.02	.04	.01	.01	.01
O	.01	.01	.06	.05	.00	.01	.11	.01	.00	.02	.01	.00	.01	.03	.51	.02	.10	.01	.00	.02	.02	.00	.01	.00	.00	.01
P	.01	.02	.02	.01	.01	.09	.01	.05	.03	.01	.01	.01	.01	.01	.06	.52	.00	.06	.00	.03	.01	.01	.00	.01	.01	.01
Q	.01	.01	.01	.01	.00	.00	.11	.06	.00	.01	.01	.01	.00	.03	.28	.01	.36	.01	.00	.00	.05	.00	.01	.01	.00	.00
R	.00	.04	.01	.00	.01	.01	.01	.09	.02	.01	.03	.03	.02	.05	.03	.05	.00	.49	.01	.02	.01	.01	.01	.03	.01	.00
S	.01	.01	.03	.01	.02	.00	.02	.06	.03	.03	.06	.03	.00	.03	.04	.05	.00	.03	.43	.03	.01	.02	.00	.03	.01	.02
T	.01	.00	.00	.01	.04	.05	.01	.01	.16	.04	.01	.05	.01	.01	.04	.04	.00	.00	.01	.42	.01	.01	.01	.01	.05	.01
U	.00	.00	.01	.01	.00	.00	.01	.11	.01	.00	.01	.02	.02	.05	.07	.01	.01	.01	.00	.02	.55	.04	.04	.01	.00	.01
V	.00	.00	.00	.01	.01	.00	.00	.03	.02	.02	.01	.01	.00	.01	.02	.02	.01	.01	.00	.02	.07	.60	.02	.01	.09	.01
W	.01	.00	.00	.01	.00	.00	.00	.08	.00	.01	.02	.01	.05	.07	.05	.01	.01	.02	.00	.02	.05	.08	.45	.03	.01	.01
X	.01	.00	.00	.00	.01	.00	.00	.03	.02	.01	.07	.01	.00	.04	.02	.01	.00	.00	.00	.05	.00	.03	.01	.55	.08	.04
Y	.01	.00	.00	.00	.00	.01	.00	.05	.02	.03	.03	.03	.00	.03	.06	.01	.00	.00	.01	.06	.01	.08	.01	.03	.51	.01
Z	.01	.01	.01	.01	.01	.00	.00	.03	.01	.06	.03	.00	.00	.00	.02	.00	.01	.00	.01	.03	.01	.01	.01	.05	.03	.66

Рисунок 2.3 – Матриця подібності від [9]

Цих 3 роботи будуть у майбутньому використання для побудови власної матриці. Проте, цього недостатньо, адже всі ці роботи розглядають лише символи верхнього регістру, і не мають жодних згадувань про нижній регістр, але розглянути його є задачею не менш важливою, можливо навіть і більш важливою, адже такі символи використовуються значно частіше про написання будь яких текстів. При цьому, найпарадоксальніше те, що робіт які розглядають символи нижнього регістру – менше, ще й відносно нових не було знайдено, тому довелось працювати із тим, що було знайдено.

Перша робота [10] якраз спеціалізується на розпізнаванні нижнього регістру. Так як ця робота є найновішою, то їй був наданий коефіцієнт 0,8.

Table 1
Confusion Matrix for the Lowercase Letters of the English Alphabet

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	.54	.01	.01	.06	.05	.00	.03	.01	.00	.00	.00	.00	.02	.05	.14	.01	.03	.00	.00	.01	.03	.00	.00	.00	.01	.00
b	.03	.53	.01	.02	.01	.00	.06	.21	.01	.00	.07	.00	.00	.02	.01	.00	.01	.00	.00	.00	.02	.01	.00	.01	.00	.00
c	.01	.03	.23	.02	.02	.03	.02	.01	.09	.02	.09	.00	.01	.03	.01	.00	.00	.18	.04	.02	.02	.02	.01	.10	.01	.01
d	.01	.03	.01	.80	.01	.00	.01	.00	.01	.06	.00	.01	.01	.01	.01	.01	.00	.00	.00	.00	.01	.01	.00	.01	.00	.01
e	.18	.03	.00	.05	.21	.01	.05	.02	.01	.02	.00	.00	.01	.06	.27	.02	.03	.00	.01	.00	.01	.00	.02	.00	.00	.00
f	.00	.00	.01	.03	.00	.27	.01	.01	.06	.15	.01	.26	.00	.00	.00	.01	.01	.11	.01	.02	.02	.00	.00	.01	.01	.00
g	.03	.00	.00	.01	.02	.00	.75	.01	.00	.01	.00	.00	.00	.03	.02	.01	.07	.00	.01	.00	.02	.00	.01	.00	.00	.01
h	.01	.05	.00	.01	.00	.00	.02	.69	.01	.00	.09	.01	.01	.05	.01	.01	.00	.00	.00	.02	.01	.00	.02	.00	.00	.00
i	.01	.01	.03	.02	.00	.03	.00	.01	.34	.09	.01	.18	.01	.00	.01	.01	.01	.01	.00	.07	.01	.00	.00	.03	.01	.00
j	.00	.00	.00	.01	.00	.03	.01	.00	.03	.80	.00	.10	.00	.00	.00	.00	.01	.01	.00	.01	.01	.00	.00	.00	.01	.00
k	.01	.01	.00	.00	.01	.02	.01	.07	.01	.00	.71	.03	.00	.02	.01	.00	.00	.01	.01	.02	.01	.00	.01	.03	.01	.01
l	.00	.00	.00	.01	.00	.07	.00	.00	.12	.09	.02	.50	.01	.00	.00	.01	.01	.06	.01	.09	.00	.01	.00	.01	.01	.00
m	.03	.00	.01	.00	.00	.00	.01	.01	.01	.02	.01	.01	.73	.06	.00	.01	.01	.02	.01	.00	.01	.01	.02	.02	.01	.00
n	.05	.01	.01	.02	.01	.01	.02	.03	.02	.01	.00	.01	.02	.64	.03	.01	.02	.03	.02	.00	.01	.01	.01	.01	.00	.00
o	.14	.02	.03	.01	.10	.01	.02	.01	.01	.00	.00	.00	.01	.07	.42	.03	.02	.02	.02	.01	.05	.01	.01	.01	.01	.00
p	.03	.03	.01	.02	.03	.00	.03	.02	.00	.00	.02	.00	.02	.14	.06	.50	.01	.01	.01	.01	.02	.00	.02	.01	.01	.00
q	.15	.00	.00	.02	.01	.02	.10	.01	.01	.01	.01	.01	.03	.04	.02	.05	.44	.01	.01	.01	.02	.00	.01	.00	.00	.00
r	.02	.01	.02	.01	.01	.03	.02	.01	.07	.02	.03	.01	.00	.02	.01	.02	.00	.37	.04	.05	.01	.03	.03	.09	.04	.03
s	.06	.03	.02	.00	.07	.00	.02	.04	.02	.01	.06	.01	.01	.13	.07	.01	.02	.05	.14	.01	.01	.06	.05	.07	.03	.02
t	.00	.00	.03	.02	.00	.04	.01	.01	.14	.02	.09	.15	.00	.01	.00	.00	.01	.10	.03	.24	.02	.02	.00	.02	.03	.02
u	.08	.04	.01	.03	.02	.00	.02	.01	.01	.01	.02	.01	.02	.04	.05	.00	.00	.01	.01	.00	.50	.07	.03	.01	.01	.01
v	.00	.03	.00	.00	.01	.00	.00	.01	.02	.00	.03	.01	.00	.03	.03	.00	.00	.03	.02	.01	.04	.51	.15	.02	.05	.01
w	.01	.00	.01	.01	.01	.01	.01	.00	.00	.00	.01	.00	.03	.03	.00	.00	.00	.01	.00	.00	.01	.06	.73	.02	.03	.01
x	.01	.01	.02	.01	.01	.00	.01	.01	.04	.01	.08	.01	.01	.04	.01	.01	.00	.05	.03	.01	.01	.04	.02	.53	.02	.03
y	.01	.00	.00	.00	.00	.02	.01	.00	.01	.00	.02	.00	.00	.01	.01	.01	.00	.01	.00	.01	.02	.13	.02	.02	.67	.02
z	.01	.00	.04	.02	.01	.02	.03	.00	.03	.02	.03	.00	.01	.01	.02	.01	.00	.10	.05	.01	.03	.07	.04	.16	.10	.19

Рисунок 2.4 – Матриця подібності від [10]

Другою, і останньою роботою для цього типу є робота [11]. Ця робота також використовує різні техніки, такі як: поєднання рис прямої лінії та замкненої кривої, поєднанням прямих і похилих елементів, тощо, для визначення схожості двох символів.

TABLE 1

MATRIX OF CONGRUENCIES FOR LETTER-PAIRS

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	—	47	61	51	56	36	44	50	41	33	46	33	43	58	52	51	49	46	59	41	61	43	47	37	35	35
b		—	47	83	50	39	30	72	44	42	48	48	40	52	67	85	90	36	34	43	52	41	46	35	38	38
c			—	51	77	28	35	42	37	30	37	28	36	50	70	50	50	43	49	39	54	34	34	25	32	29
d				—	49	45	40	74	43	36	48	48	43	57	63	95	80	42	36	42	55	38	35	35	39	34
e					—	28	38	43	31	26	32	26	34	47	74	51	51	37	52	36	50	33	33	23	32	33
f						—	29	46	59	50	46	63	29	40	27	45	43	53	30	67	38	41	27	36	41	43
g							—	33	30	33	35	27	34	41	33	42	43	19	35	27	45	34	31	34	33	28
h								—	46	35	67	53	50	78	45	60	59	42	41	46	67	34	37	35	30	32
i									—	67	43	67	31	44	28	44	39	53	34	54	39	41	25	36	33	42
j										—	37	53	21	31	28	40	42	36	17	53	33	34	22	35	38	37
k											—	56	19	27	29	42	40	38	38	40	36	42	40	36	42	40
l												—	26	36	25	45	42	46	26	54	34	39	27	43	42	44
m													—	61	34	40	37	35	32	26	55	31	32	24	30	25
n														—	48	53	50	50	45	38	89	44	42	28	32	36
o															—	57	60	31	39	29	56	34	28	22	20	25
p																—	89	36	33	43	51	43	31	39	40	32
q																	—	35	34	43	44	38	30	38	39	33
r																		—	28	54	47	42	32	39	34	42
s																			—	43	49	39	36	34	25	52
t																				—	42	48	34	42	39	53
u																					—	39	39	33	34	37
v																						—	47	42	60	44
w																							—	33	39	31
x																								—	41	59
y																									—	39
z																										—

Decimals omitted.

Рисунок 2.5 – Матриця подібності від [11]

Кожне значення з отриманою матриці було поділено на 100 аби відповідати стандартам минулих матриць, що мали значення від 0 до 1 з двома числами після крапки.

Джерел з яких можна було б знайти подібні матриці, але типу «Літера-Цифра» або «Верхній регістр-Нижній регістр» знайдено не було, тому довелось побідні матриці робити базуючись із власного досвіду читання англомовних форумів, чатів та інших ресурсів а також власного текстового спілкування із англомовними користувачами побідних веб-сайтів. На додаток до цього, вивчались різні фразеологізми та окремі слова, способи їх використання і їх значення за допомогою «Urban Dictionary» [12]. Після цього було зроблено висновок, що пар між літерами верхнього ти нижнього регістру дуже мало, фактично по 2 букви з кожного, які можуть час від часу заміняти одна іншу, а саме це букви i, l та I, J. Тому результуюча таблиця для них буде мати наступний вигляд:

Таблиця 2.1 – Пари схожості «lowercase to uppercase»

l	I
l	J
i	I
i	J

Також потребують уточнення матриці типу «Літера-Цифра», де під літерою потрібно розуміти обидва регістри. Як писалось вище, потрібних робіт не було знайдено, тому для побудови матриць використовувались ті самі методи, що і для верхнього та нижнього регістрів. У цих матрицях значення мали 3 варіанти, де 0 – не схоже, 0,3 – дещо схоже, 0,7 – схоже. Виставлення коефіцієнтів відповідним парам літера-число здебільшого базувалось на популярних замінах у вищезгаданій «мові» leet через брак інших джерел. Також із матриць були вилучені ті рядки і стовпці, де відповідні їм літери не схожі взагалі не схожі на жодну із цифр. Матриця для верхнього регістру буде мати наступний вигляд:

Таблиця 2.2 – Матриця сплутувань «uppercase to number»

	1	2	4	5	6	7	8	9	0
A	0,00	0,00	0,70	0,00	0,00	0,00	0,00	0,00	0,00
B	0,00	0,00	0,00	0,00	0,70	0,00	0,35	0,00	0,00
G	0,00	0,00	0,00	0,00	0,35	0,00	0,00	0,00	0,00
I	0,70	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
L	0,70	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
O	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,70
R	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,70	0,00
S	0,00	0,00	0,00	0,70	0,00	0,00	0,00	0,00	0,00
T	0,00	0,00	0,00	0,00	0,00	0,35	0,00	0,00	0,00
U	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,35
Z	0,00	0,70	0,00	0,00	0,00	0,00	0,00	0,00	0,00

Отже, з матриці можна зробити висновок, що пар «число-літера» не дуже багато, і одна буква одночасно схожа не більше ніж як на 2 цифри, що значно спростить розробку програмного забезпечення у майбутньому.

Подібна матриця, але символів нижнього регістру буде мати наступний вигляд:

Таблиця 2.3 – Матриця сплутувань «lowercase to number»

	1	3	4	6	9	0
a	0,0	0,0	0,35	0,0	0,0	0,0
b	0,0	0,0	0,0	0,70	0,0	0,0
e	0,0	0,70	0,0	0,0	0,0	0,0
g	0,0	0,0	0,0	0,0	0,70	0,0
i	0,70	0,0	0,0	0,0	0,0	0,0
l	0,70	0,0	0,0	0,0	0,0	0,0
o	0,0	0,0	0,0	0,0	0,0	0,70
q	0,0	0,0	0,0	0,0	0,70	0,0
u	0,0	0,0	0,0	0,0	0,0	0,35
x	0,0	0,0	0,0	0,0	0,0	0,0
y	0,0	0,0	0,0	0,0	0,0	0,0
z	0,0	0,0	0,0	0,0	0,0	0,0

Висновки з цієї таблиці можна майже ті ж самі, що і з минулої, єдина різниця, що жодна буква не є схожою одночасно на 2 і більше цифри, що теж спрощує майбутню розробку.

Після розгляду усіх робіт є необхідним обробити всі отримані данні. Для кожної можливою пари букв, загалом 676 пар для кожного з регістрів. Для кожної такої S(a,b) пари були приставлені відповідні значення з Першоджерела 1, Першоджерела 2, …, розподілені коефіцієнти відповідно цим джерелам та обраховані значення схожості. Для обрахунку використовувалась наступна формула:

$$S(X,Y) = \frac{\sum_i C_i * V_i}{\sum_i C_i},$$

де $S(X,Y)$ – схожість між досліджуваними об’єктами X та Y;

C_i – коефіцієнт довіри джерела i;

V_i – відповідне значення для джерела i;

Наприклад, таке значення для пари A-B буде наступним:

$$S(A,B) = \frac{0.71*0.7+0*0.5+0*0.5}{0.7+0.5+0.5} = 0{,}292.$$

Отримуємо, що схожість між цими буквами низька, і навряд чи їх можна взаємозамінювати для потенційних підмін у майбутньому. Але це всього лише 1 пара, а таких декілька сотень, як було зазначено раніше, тому потрібно навести наступні таблиці для кращої візуалізації даних. Результуючі таблиці будуть мати наступний вигляд:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0,839	0,292	0,206	0,249	0,256	0,254	0,243	0,245	0,278	0,259	0,313	0,234	0,231	0,258	0,218	0,245	0,270	0,312	0,256	0,227	0,178	0,215	0,264	0,304	0,206	0,251
B	0,304	0,572	0,301	0,384	0,329	0,319	0,349	0,299	0,258	0,238	0,266	0,268	0,251	0,279	0,339	0,308	0,311	0,363	0,339	0,235	0,290	0,291	0,266	0,299	0,261	0,288
C	0,210	0,291	0,769	0,309	0,331	0,285	0,374	0,245	0,209	0,192	0,261	0,330	0,210	0,218	0,378	0,285	0,295	0,261	0,289	0,238	0,296	0,242	0,235	0,225	0,217	0,266
D	0,247	0,352	0,320	0,705	0,262	0,261	0,338	0,301	0,201	0,213	0,251	0,281	0,254	0,270	0,409	0,318	0,346	0,314	0,268	0,175	0,339	0,272	0,248	0,238	0,206	0,245
E	0,261	0,334	0,351	0,261	0,649	0,387	0,312	0,259	0,272	0,214	0,264	0,329	0,225	0,229	0,278	0,326	0,259	0,268	0,323	0,297	0,235	0,276	0,255	0,279	0,278	0,301
F	0,251	0,318	0,276	0,262	0,357	0,709	0,296	0,231	0,297	0,205	0,234	0,289	0,205	0,205	0,247	0,370	0,247	0,249	0,289	0,354	0,205	0,274	0,235	0,281	0,300	0,298
G	0,248	0,325	0,369	0,313	0,314	0,295	0,679	0,292	0,229	0,213	0,235	0,271	0,225	0,257	0,406	0,281	0,319	0,279	0,322	0,217	0,298	0,257	0,259	0,239	0,213	0,235
H	0,245	0,277	0,242	0,292	0,247	0,226	0,270	0,704	0,156	0,194	0,323	0,271	0,337	0,453	0,294	0,298	0,285	0,351	0,241	0,120	0,344	0,242	0,281	0,255	0,188	0,198
I	0,272	0,256	0,206	0,198	0,259	0,269	0,229	0,155	0,854	0,336	0,231	0,239	0,198	0,181	0,196	0,222	0,222	0,196	0,272	0,354	0,152	0,271	0,231	0,276	0,315	0,321
J	0,254	0,231	0,189	0,211	0,202	0,194	0,213	0,195	0,353	0,830	0,222	0,158	0,202	0,210	0,223	0,184	0,231	0,201	0,269	0,291	0,235	0,238	0,222	0,251	0,262	0,265
K	0,297	0,266	0,259	0,254	0,254	0,232	0,234	0,341	0,233	0,224	0,539	0,281	0,284	0,338	0,234	0,270	0,256	0,327	0,239	0,196	0,260	0,260	0,272	0,361	0,260	0,269
L	0,221	0,264	0,320	0,283	0,283	0,271	0,259	0,278	0,244	0,157	0,269	0,878	0,251	0,246	0,272	0,266	0,251	0,262	0,231	0,222	0,268	0,245	0,222	0,243	0,228	0,251
M	0,231	0,244	0,212	0,250	0,225	0,203	0,229	0,361	0,198	0,205	0,276	0,251	0,682	0,350	0,256	0,280	0,230	0,308	0,218	0,180	0,356	0,286	0,273	0,235	0,259	0,198
N	0,260	0,259	0,222	0,268	0,230	0,202	0,254	0,405	0,181	0,210	0,331	0,246	0,351	0,713	0,271	0,278	0,249	0,343	0,262	0,164	0,325	0,247	0,293	0,269	0,218	0,196
O	0,217	0,309	0,366	0,377	0,270	0,251	0,406	0,283	0,194	0,220	0,218	0,265	0,242	0,269	0,742	0,295	0,348	0,279	0,282	0,192	0,336	0,243	0,251	0,206	0,185	0,225
P	0,243	0,304	0,283	0,311	0,309	0,389	0,276	0,308	0,227	0,184	0,258	0,268	0,280	0,277	0,306	0,411	0,261	0,382	0,252	0,224	0,275	0,288	0,264	0,258	0,256	0,258
Q	0,272	0,299	0,310	0,324	0,261	0,248	0,371	0,266	0,222	0,234	0,254	0,254	0,224	0,250	0,448	0,262	0,263	0,272	0,240	0,206	0,281	0,272	0,243	0,250	0,218	0,240
R	0,302	0,311	0,264	0,302	0,264	0,246	0,279	0,386	0,198	0,202	0,315	0,271	0,321	0,366	0,299	0,349	0,279	0,293	0,265	0,161	0,283	0,247	0,277	0,272	0,192	0,214
S	0,268	0,339	0,305	0,279	0,328	0,288	0,343	0,260	0,283	0,272	0,261	0,255	0,214	0,269	0,298	0,271	0,245	0,283	0,644	0,268	0,256	0,255	0,247	0,285	0,255	0,268
T	0,217	0,231	0,226	0,180	0,284	0,323	0,217	0,114	0,389	0,279	0,196	0,218	0,180	0,164	0,198	0,243	0,206	0,152	0,264	0,753	0,151	0,252	0,217	0,265	0,362	0,317
U	0,177	0,270	0,299	0,328	0,232	0,202	0,286	0,364	0,155	0,226	0,255	0,275	0,293	0,324	0,360	0,276	0,271	0,283	0,248	0,155	0,789	0,296	0,277	0,213	0,199	0,205
V	0,210	0,291	0,239	0,273	0,277	0,283	0,254	0,248	0,274	0,243	0,254	0,247	0,281	0,243	0,249	0,290	0,276	0,246	0,248	0,261	0,297	0,779	0,243	0,272	0,396	0,246
W	0,266	0,267	0,234	0,255	0,258	0,239	0,264	0,300	0,231	0,228	0,275	0,228	0,264	0,296	0,266	0,269	0,244	0,279	0,248	0,221	0,289	0,262	0,715	0,260	0,202	0,238
X	0,306	0,306	0,222	0,238	0,283	0,282	0,239	0,271	0,286	0,247	0,368	0,252	0,231	0,287	0,212	0,263	0,248	0,268	0,284	0,278	0,211	0,284	0,254	0,318	0,326	0,349
Y	0,209	0,262	0,215	0,202	0,269	0,311	0,210	0,200	0,311	0,256	0,260	0,231	0,261	0,223	0,204	0,262	0,218	0,191	0,255	0,377	0,201	0,401	0,202	0,309	0,720	0,294
Z	0,251	0,294	0,265	0,246	0,298	0,298	0,235	0,206	0,294	0,303	0,276	0,251	0,198	0,191	0,228	0,256	0,242	0,216	0,280	0,326	0,205	0,247	0,238	0,351	0,298	0,800

Рисунок 2.6 – Результуюча матриця для верхнього регістру

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0,737	0,207	0,267	0,253	0,269	0,154	0,206	0,220	0,176	0,141	0,197	0,141	0,196	0,277	0,303	0,224	0,227	0,197	0,253	0,181	0,279	0,184	0,201	0,159	0,156	0,150
b	0,219	0,731	0,207	0,367	0,220	0,167	0,163	0,429	0,194	0,180	0,246	0,206	0,171	0,234	0,293	0,364	0,391	0,154	0,146	0,184	0,234	0,181	0,197	0,156	0,163	0,163
c	0,267	0,219	0,560	0,230	0,341	0,137	0,161	0,186	0,210	0,140	0,210	0,120	0,160	0,231	0,306	0,214	0,214	0,287	0,233	0,179	0,243	0,157	0,151	0,164	0,143	0,130
d	0,224	0,373	0,224	0,886	0,216	0,193	0,177	0,317	0,190	0,189	0,206	0,211	0,190	0,250	0,276	0,413	0,343	0,180	0,154	0,180	0,241	0,169	0,150	0,156	0,167	0,151
e	0,343	0,231	0,330	0,239	0,549	0,126	0,191	0,196	0,139	0,123	0,137	0,111	0,151	0,236	0,471	0,230	0,236	0,159	0,229	0,154	0,220	0,141	0,153	0,099	0,137	0,141
f	0,154	0,167	0,126	0,210	0,120	0,583	0,130	0,203	0,287	0,300	0,203	0,419	0,124	0,171	0,116	0,199	0,190	0,290	0,134	0,299	0,174	0,176	0,116	0,160	0,181	0,184
g	0,206	0,129	0,150	0,177	0,174	0,124	0,857	0,147	0,129	0,147	0,150	0,116	0,146	0,193	0,153	0,186	0,224	0,081	0,156	0,116	0,204	0,146	0,139	0,146	0,141	0,126
h	0,220	0,337	0,180	0,323	0,184	0,197	0,153	0,823	0,203	0,150	0,339	0,233	0,220	0,363	0,199	0,263	0,253	0,180	0,176	0,209	0,293	0,146	0,170	0,150	0,129	0,137
i	0,181	0,194	0,176	0,196	0,133	0,270	0,129	0,203	0,623	0,339	0,190	0,390	0,139	0,189	0,126	0,194	0,173	0,233	0,146	0,271	0,173	0,176	0,107	0,171	0,147	0,171
j	0,141	0,180	0,129	0,160	0,111	0,231	0,147	0,150	0,304	0,886	0,159	0,284	0,090	0,133	0,120	0,171	0,186	0,160	0,073	0,233	0,147	0,146	0,094	0,150	0,169	0,159
k	0,203	0,211	0,159	0,206	0,143	0,209	0,156	0,327	0,190	0,159	0,834	0,257	0,081	0,127	0,130	0,180	0,171	0,169	0,169	0,183	0,160	0,180	0,177	0,171	0,186	0,177
l	0,141	0,206	0,120	0,211	0,111	0,310	0,116	0,227	0,356	0,279	0,251	0,714	0,117	0,154	0,107	0,199	0,186	0,231	0,117	0,283	0,146	0,173	0,116	0,190	0,186	0,189
m	0,201	0,171	0,160	0,184	0,146	0,124	0,151	0,220	0,139	0,101	0,087	0,117	0,846	0,296	0,146	0,177	0,164	0,161	0,143	0,111	0,241	0,139	0,149	0,114	0,134	0,107
n	0,277	0,229	0,220	0,256	0,207	0,177	0,187	0,351	0,200	0,139	0,116	0,160	0,273	0,794	0,223	0,233	0,226	0,231	0,204	0,163	0,387	0,194	0,186	0,126	0,137	0,154
o	0,303	0,299	0,317	0,276	0,374	0,121	0,153	0,199	0,126	0,120	0,124	0,107	0,151	0,246	0,669	0,261	0,269	0,144	0,179	0,130	0,269	0,151	0,126	0,100	0,091	0,107
p	0,236	0,381	0,220	0,419	0,236	0,193	0,197	0,269	0,189	0,171	0,191	0,193	0,183	0,307	0,279	0,714	0,387	0,160	0,147	0,190	0,230	0,184	0,144	0,173	0,177	0,137
q	0,296	0,386	0,214	0,354	0,224	0,196	0,241	0,259	0,173	0,186	0,177	0,186	0,176	0,237	0,269	0,410	0,680	0,156	0,151	0,190	0,200	0,163	0,134	0,163	0,167	0,141
r	0,209	0,160	0,196	0,186	0,164	0,244	0,093	0,186	0,267	0,166	0,180	0,203	0,150	0,226	0,139	0,166	0,150	0,640	0,143	0,260	0,207	0,197	0,154	0,219	0,169	0,197
s	0,287	0,163	0,221	0,154	0,263	0,129	0,161	0,199	0,157	0,079	0,197	0,117	0,143	0,267	0,207	0,147	0,157	0,149	0,509	0,190	0,216	0,201	0,183	0,186	0,124	0,234
t	0,176	0,184	0,184	0,191	0,154	0,310	0,121	0,203	0,311	0,239	0,223	0,317	0,111	0,169	0,124	0,184	0,190	0,289	0,201	0,566	0,191	0,217	0,146	0,191	0,184	0,239
u	0,307	0,246	0,237	0,253	0,226	0,163	0,204	0,293	0,173	0,147	0,166	0,151	0,247	0,404	0,269	0,219	0,189	0,207	0,216	0,180	0,714	0,207	0,184	0,147	0,151	0,164
v	0,184	0,193	0,146	0,163	0,147	0,176	0,146	0,151	0,187	0,146	0,197	0,173	0,133	0,206	0,163	0,184	0,163	0,197	0,179	0,211	0,190	0,720	0,287	0,191	0,286	0,194
w	0,207	0,197	0,151	0,156	0,147	0,121	0,139	0,159	0,107	0,094	0,177	0,116	0,154	0,197	0,120	0,133	0,129	0,143	0,154	0,146	0,173	0,236	0,846	0,153	0,176	0,139
x	0,164	0,156	0,119	0,156	0,104	0,154	0,151	0,156	0,177	0,156	0,200	0,190	0,109	0,143	0,100	0,173	0,163	0,196	0,163	0,186	0,147	0,203	0,153	0,731	0,187	0,270
y	0,156	0,163	0,137	0,167	0,137	0,187	0,147	0,129	0,147	0,163	0,191	0,180	0,129	0,143	0,091	0,177	0,167	0,151	0,107	0,173	0,157	0,331	0,170	0,187	0,811	0,179
z	0,156	0,163	0,147	0,157	0,147	0,196	0,137	0,137	0,189	0,170	0,189	0,189	0,113	0,160	0,119	0,143	0,141	0,237	0,251	0,233	0,176	0,229	0,156	0,344	0,224	0,537

Рисунок 2.7 – Результуюча матриця для нижнього регістру

Маючи ці результати можна виділити для кожної букви ряд схожих на неї букв. Середнє значення схожості букви верхнього регістру саму на себе S(A,A); S(B,B); …; становить ~0,667. Позаяк середнє значення бралось на основі інших робіт, у яких значення схожості «1» майже не було, то і середнє значення не є одиницею. Це значення можна розділити на 2 щоб отримати реалістичний поріг значення яке має пройти деяка літера, щоб потрапити до списку схожих, це значення буде ~0,333. Для символів нижнього регістру відповідні значення становлять 0,700 і 0,350 відповідно. Після взяття всіх букв які мають значення схожості відповідних порогів отримаємо наступні пари «символ-можлива заміна»:

1 – l, i, I;	4 – a, A;	7 – t, T;	0 – o, u, O, U;
2 – Z;	5 – S;	8 – B;	
3 – e, E;	6 – b, G;	9 – g, q, R;	

B – D, G;	H – N;	O – G, D, C, U;	S – Z;
C – G, O;	I – l, i, J, L;	P – F;	Y – V;
D – B, O;	J – l, i, I;	U – O;	Z – S;
F – P;	L – I;	V – Y, M;	
G – O, C;	N – H;	W – V;	

b – h, p, q, g;	c – e;	e – c;	g – q, b;
h – b;	i – I, J, l, j;	j – i;	l – l, I, J, i;
o – u;	p – b;	q – b, g;	s – z;
u – o;	v – w;	w – v;	z – s;

Важливо зауважити, що у показаних вище парах не вказані копії символів, адже очевидно, що будь який символ схожий сам на себе; всі інші символи розставлені у порядку спадання коефіцієнту схожості.

Отже, отримавши ці дані можна їх використати у майбутньому при розробці програмного забезпечення, аби додати їх у програму, щоб та мала з чим працювати.

2.2 Розробка алгоритму обробки матриць сплутування

Для того, аби раніше створені матриці сплутувань мали хоча б який-небудь сенс, то необхідно написати алгоритм, що буде використовувати ці матриці для пошуку можливих замін у слові. Серед можливих алгоритмів можна навести наступні: backtracking algorithm (бектрекінг алгоритм), combinatorial approach (комбінаторний підхід), recursive enumeration (рекурсивний перебір), graph-based approach (підхід за допомогою графа як структури даних). Огляд кожного з алгоритмів наведено нижче:

- Backtracking algorithm

  Цей алгоритм належить до класу алгоритмів для пошуку частини або всіх розв’язків деякої задачі шляхом поступової побудови кандидатів і відсіювання тих кандидатів які не можуть бути правильним завершеним розв’язком заданої проблеми [23]. Алгоритм працює в чотири кроки:

  1. Першим кроком є вибір елементу з якого потрібно почати розв’язок або ж вибір стану з якого продовжиться вже існуюче

рішення. Цей крок визначає потенційні варіанти на наступному етапі алгоритму;

2. Другим кроком є перевірка, де поточний обраний елемент або стан, де це об’єкт, перевіряється на відповідність критеріям та обмеженням заданої проблеми. Якщо перевірка успішна, то тоді алгоритму переходить на крок 3, якщо ж об’єкт ці перевірку не проходить, то алгоритм його відсіює, щоб вибрати інший елемент та знову повторити перевірку, що допомагає уникнути нескінченних циклів.
3. Третім кроком є повернення до попереднього вибору і перевірки, коли всі можливі варіанти закінчились. Повернення відбувається у тому випадку, коли поточний об’єкт не веде до розв’язання проблеми або ж до моменту, коли всі можливі варіанту вибору вичерпані. Якщо повернення відбувається, то це означає повне відсіювання цього «шляху» і повернення до попереднього стану аби пробувати інший «шлях».
4. Четвертий крок у разі успішних перевірок на кроці 2. Цей крок алгоритму продовжує вирішення проблеми у заданому «шляху» і рухається вперед до моменту знаходження наступного можливого варіанту або ж до моменту вирішення задачі. Цей крок може повторюватись багато разів, доки не будуть знайдені усі можливі варіанти або ж єдиний розв’язок.

Таким чином ці чотири кроки є ключовими для пошуку рішень за допомогою цього алгоритму та кожний з них грає важливо роль під час процесу пошуку. Всі ці кроки у вигляді рисунку наведені на рис. 2.8.

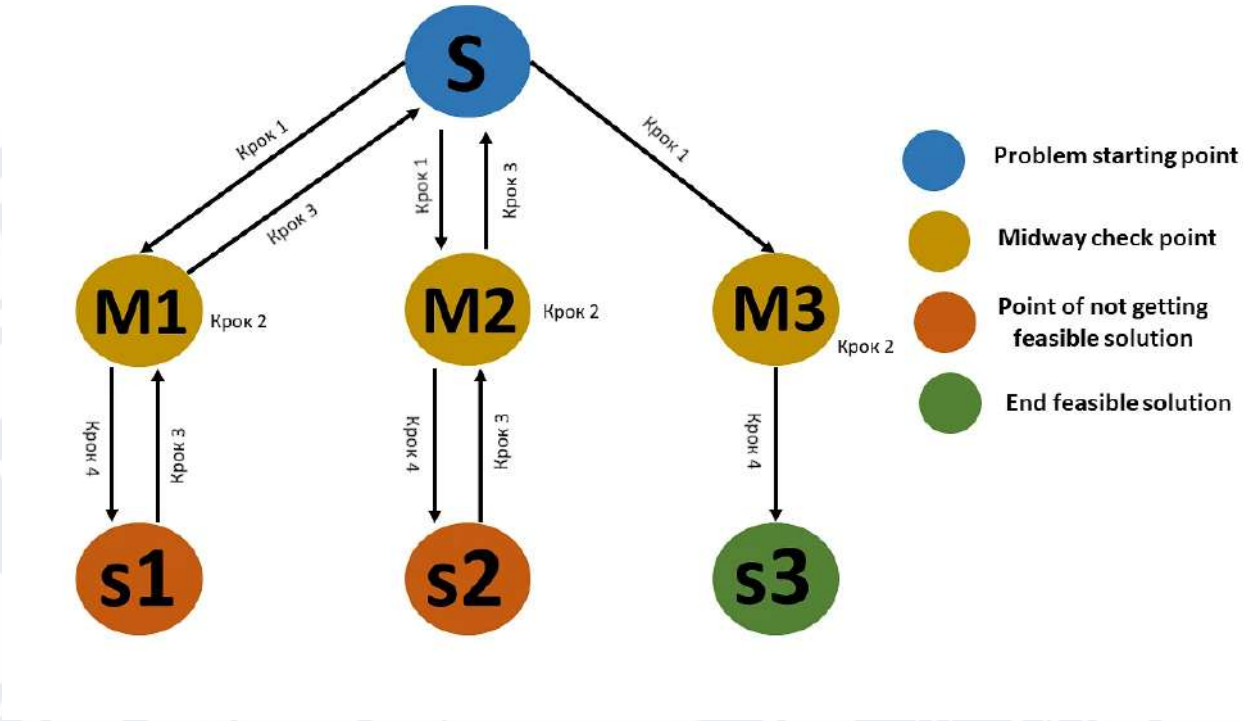


Рисунок 2.8 – Приклад роботи backtracking алгоритму

Базуючись на даних про цей алгоритм, можна визначити наступні плюси цього алгоритму:

- Ефективність при генерації комбінацій, адже алгоритм рекурсивно або ітеративно сам обирає обирає всі можливі комбінації;
- Можливість раннього відсіювання, адже алгоритм постійно перевіряє «гілки» на правильність за заданими критеріями;
- Легкість додання нових критеріїв, до кожного кроку можна додавати будь-які нові умови згідно із заданою задачею.

Серед мінусів цього алгоритму можна виділити наступне:

- Експоненціальна складність, адже у разі якщо існує особливо велика кількість можливих комбінацій, то алгоритму може мати таку складність. Складність становить $O(2^n)$, де n – довжина слова;

- Складність реалізації, адже потрібно мати чіткі правила та логіку за якими цей алгоритм буде працювати, інакше можлива велика кількість помилок;

- Combinatorial approach

  Комбінаторний підхід, або ж комбінаторна оптимізація це підрозділ математичної оптимізації, що полягає у знаходженні оптимального об'єкта зі скінченної множини об'єктів, коли множина допустимих рішень є дискретною або може бути зведена до дискретної множини [24]. Комбінаторна оптимізація в основному використовується для розв'язання дискретних оптимізаційних задач за допомогою комбінаторних методів. Ці задачі дискретної оптимізації - це, по суті, задачі, які мають на меті знайти найкраще рішення, яке можливе з кінцевого набору можливих варіантів [25]. Серед плюсів цього варіанту можна навести наступні переваги:

  - Ефективність обчислень, адже дає змогу знаходити перестановки безпосередньо, без необхідності перевіряти всі варіанти;
  - Можливість задавати обмеження, щоб результуючі варіанти відповідали певним умовам;

  Серед мінусів можна виділити наступні:

  - Складність обчислювання, адже якщо слово досить довге, то складність росте експоненціальної складності обчислень;
  - Не ефективність при великій кількості можливих замін, адже це призведе до великих витрат по ресурсам під час роботи алгоритму;
  - Створення великої кількісті непотрібних варіантів, що не відповідають умовам задачі.

- Recursive enumeration

  Цей підхід є алгоритмом, що шукає всі можливі варіанти рішення проблеми виконуючи повторення завдань, що накопичують рішення при повторних викликах функції, де кожний останній виклик функції визначає початковий стан для наступного запиту [26]. Цей підхід розглядає задачу генерації варіації слів як набір підзадач, перевіряючи кожний окремий елемент, де для кожного елемента (літери) вирішується, чи буде він включений у вибірку чи ні.

  Таким чином, алгоритм проходить деяку множину у визначеному порядку будуючи слово, обираючи або відкидуючи літеру-альтернативу на кожному кроці для поточної базової літери із множини k можливих замін. Ілюстрація цього алгоритму наведена на рис. 2.9.

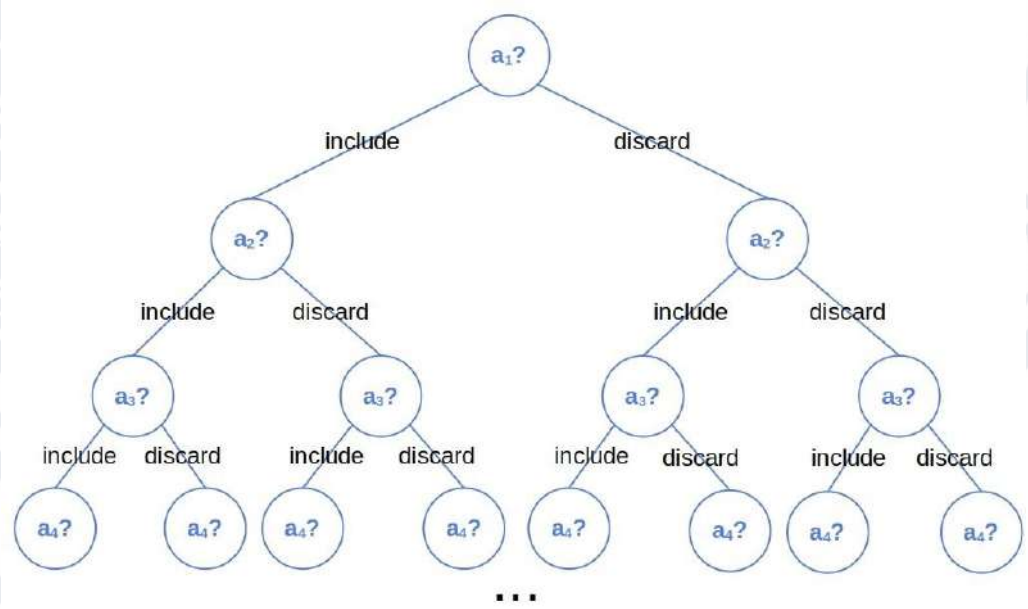


Рисунок 2.9 – Приклад роботи Recursive enumeration алгоритму

Алгоритм може перерахувати усі комбінації, обходячи все дерево і збираючи його листя, відстежуючи на кожному кроці поточної активної комбінації і спускаючись від верху дерево до низу запам’ятовуючи рішення при включення елемента вершини; при процесі підняття алгоритм рішення пов’язане з цією вершиною – змінюється.

Серед плюсів цього алгоритму можна навести наступні:

- Простота реалізації;
- Потужність при роботі з малими обсягами даних;

Серед мінусів для цього алгоритму можна виділити наступні:

- Висока складність, адже для великих обсягів даних кількість можливих комбінацій може бути дуже великою, що призводить до експоненціальної складності обчислень.
- Займає багато пам’яті при роботі з великим обсягом даних;
- Глибокі рекурсивні виклики можуть переповнити стек, що призведе до передчасного аварійного закінчення програми.

- Graph-based approach

Графи є часто використовуваним типом даних, що зберігає данні у виглядів вершин із зв’язками між ними, що точно відображають взаємозв’язок вхідних даних. У такому графі кожна вершина є літерою яку можна вставити у комбінацію і кожне ребро відображає логічний порядок вставки символів. Починаючи з першого вузла графа йде рух вздовж ребер вибираючи можливий наступний символ у комбінації, і цей рух продовжується доки не буде досягнутий останній вузол [27]. Найпопулярнішим варіантом побудови слів за допомогою графа є 26-шляхове дерево (trie), що дає

змогу швидко перевіряти чи є поточний шлях із букв, що представляється зліва направо, потенційним варіантом [28]. Як приклад використання такого графа можна навести варіант, коли він використовується для побудови слів або фраз із заданого числа букв і можливих пар. Якщо взяти наступний набір «heahrhn», то граф буде мати вигляд як показано на рис. 2.10.

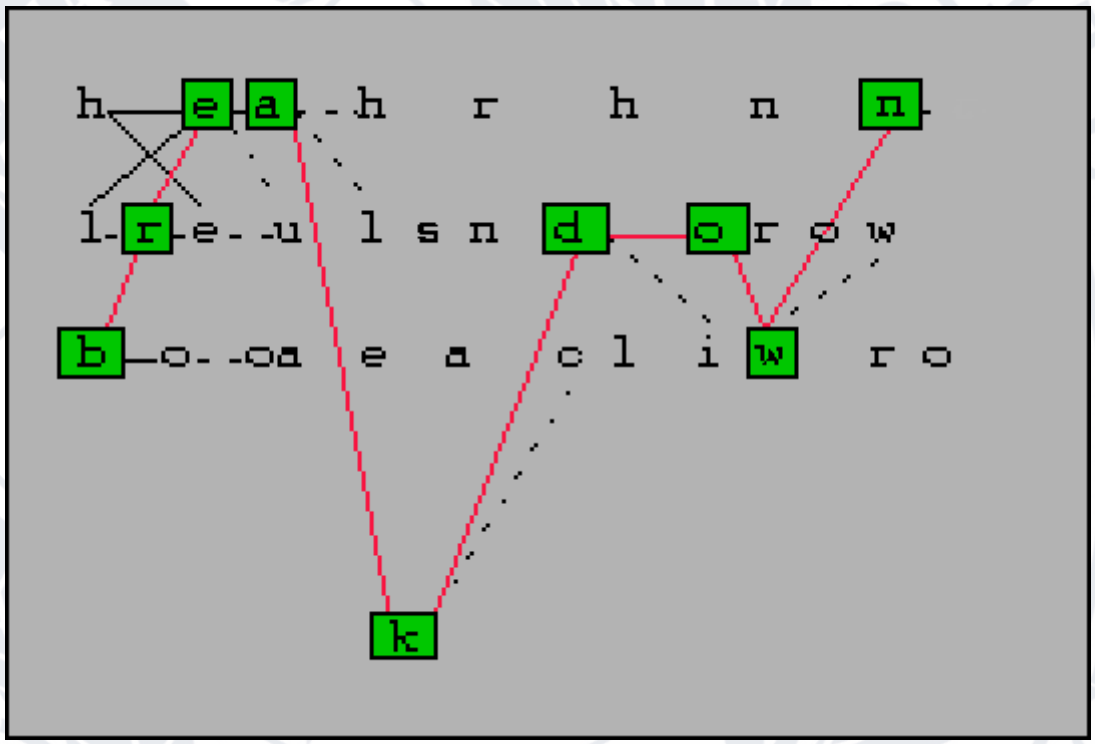


Рисунок 2.10 – Приклад побудови фрази із заданих букв та можливих пар

Серед плюсів цього підходу можна виділити наступні:

- Більше зрозуміла структура для людського сприйняття, адже такі графи можна відобразити у вигляді рисунку, що легко читати;
- Універсальність, цей в загальному цей алгоритм підходить для вирішення широко спектру різних задач.

До мінусів цього підходу можна віднести наступні пункти:

- Складність побудови, адже створення самої структури для кожної задачі часто є нетривіальною проблемою і їх оновлення або адаптація під інші вхідні дані потребує внесення великої кількості змін;
- Вимогливість до пам’яті, адже зберігання даних та пошук по ним може займати багато пам’яті, особливо на великих графах;

Потреба використовувати вузько-спеціалізовані бібліотека та фреймворки, які не завжди мають достатній рівень документації.

Підбиваючи підсумки огляду можливих підходів для реалізації початкового задуму, можна виділити backtracking алгоритм, адже він хоча й не є найбільш ефективним з точки зору обчислювань, але має помірну складність реалізації і дуже легко піддається модифікаціям та до нього легко додати нові обмеження, аби відповідати новим завданням. Цей алгоритм буде обраний для подальшої розробки програмного забезпечення.

Висновок до розділу 2

В цьому розділі буд наведений опис існуючих матриць сплутувань та виведені власні матриці на основі попередніх, а також були розглянуті можливі алгоритми для подальшої реалізації у програмному забезпеченні і обраний серед них той, що найбільше відповідає заданим критеріям.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір технологій

Java – об'єктно-орієнтована мова програмування, випущена 1995 року компанією «Sun Microsystems» як основний компонент платформи Java. З 2009 року мовою займається компанія «Oracle», яка того року придбала «Sun Microsystems». В офіційній реалізації Java-програми компілюються у байт-код, який при виконанні інтерпретується віртуальною машиною для конкретної платформи [13].

Ця мова програмування була обрана завдяки декільком ключовим критеріям, такими як висока продуктивність виконання, відносно простий і зрозумілий синтаксис, що значно спрощує процес читання коду, а також велика кількість різноманітних спільнот, бібліотек та іншого, що допоможе вирішити поставлену задачу.

Також ця мова є в топ 3 найпопулярніших мов програмування згідно із GitHub станом на 2022 рік [14].

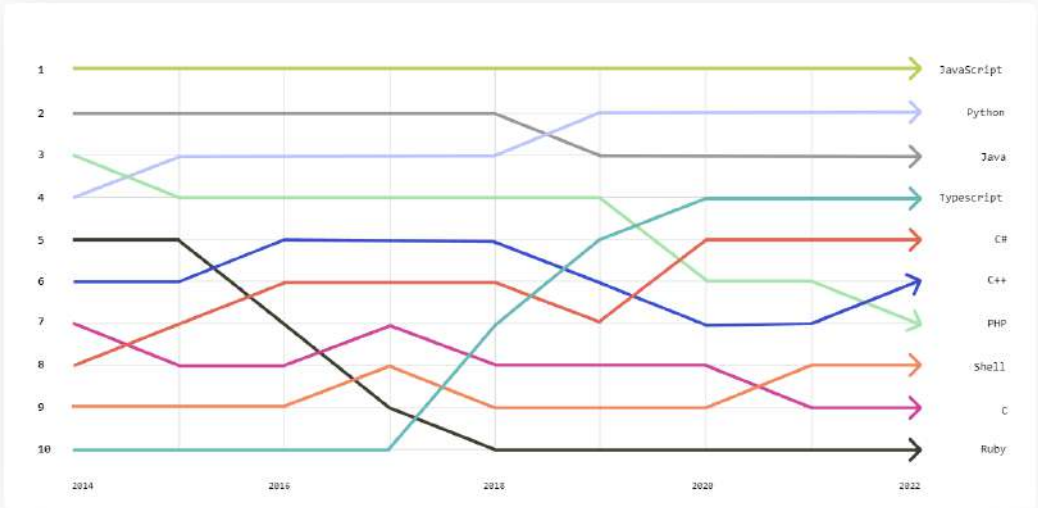


Рисунок 3.1 – найпопулярніші мови програмування на GitHub

Для зручної роботи із цією мовою програмування було обрано IDE IntelliJ IDEA. Ця IDE підтримує дуже широкий спектр інструментів за допомогою плагінів, такі як система тестування JUnit, система контролю

версій Git, застосунки для збірки як Maven або Gradle. Вбудовані редактори для XML/JSON файлів, регулярних виразів, вбудована система перевірки синтаксису та коректності коду. Ці всі плагіни досить суттєво зберігають час та сили під час написання та рефакторингу коду, тестування програми в цілому, або її окремих модулів, збірку та компіляції всього програмного забезпечення та подальше його розгортання на сервері, чи у хмарному середовищі.

Підсумувавши інформацію вище, може виділити наступні переваги такої зв'язки як IntelliJ IDEA та Java:

a) Підтримка широкого спектру типів текстових файлів;
b) Потужний, розумний редактор коду, що надає підказки як краще написати ту чи іншу функції та підсвічує змінні, методи, класи, що не використовуються ніде у коду програмного забезпечення;
c) Можливість розробляти програмне забезпечення яке завдяки Java-машині буде працювати майже на будь якому пристрої;
d) Широкий спектр можливостей для тестування коду;
e) Потужна система збору проектів Maven, що також дозволяє підключати бібліотеки та плагіни до коду буквально за секунди;
f) Дуже велика спільнота розробників, велика кількість форумів, посібників та інших навчальних матеріалів, що допомагають як і на початкових етапах, так і на більше пізніх.

Хоча мова є потужною і все ще оновлюється, вона також має низку недоліків, таких як:

a) Популярність виключно у back-end розробці;

b) Більшість фреймворків та бібліотек потрібно вивчати не менше ніж саму мову, настільки вони великі та багатофункціональні;
c) Досить часто навчальні матеріали написані для старих версій мови, і ці матеріали через це частково втрачають актуальність.

IDE IntelliJ IDEA є, фактично, ідеальним варіантом для розробки, і серед недоліків можна виділити хіба що, те, що на відносно старих ПК процес збірки і компіляції коду за допомогою цієї IDE може займати немало часу.

Також для розробки даного програмного забезпечення також був обраний такий фреймворк як Spring. Spring Framework (Spring) - це фреймворк додатків з відкритим вихідним кодом, який надає інфраструктурну підтримку для розробки Java-додатків. Один з найпопулярніших фреймворків Java Enterprise Edition (Java EE), Spring допомагає розробникам створювати високопродуктивні додатки, використовуючи Plain Old Java Objects (POJO). Цей фреймворк - це великий масив заздалегідь визначеного коду, до якого розробники можуть додавати код для вирішення проблеми у певній галузі [15].

Також застосунок буде мати форму веб-додатку, а саме - сервера з API, що приймає на вхід деякий текст, а повертає його ж, але відфільтрованим, разом із словами-тригерами. Цей застосунок буде мати архітектуру MVC (Model-View-Controller). Spring Boot надає можливість дуже швидко, почати розробку подібного додатку завдяки тому, що майже всю роботу по конфігурації серверу, як локально так і глобально, фреймворк бере на себе і виконує свою роботу поза межами уваги розробника. Хоча й має можливість детального конфігурування залежно від обставин [16-17].

Серед переваг використання Spring можна виділити наступне:

a) Створення додатків за допомогою POJO, що дає змогу не використовувати ніякі треті сервери;
b) Spring має модульну форму, тому всі непотрібні сервіси можна відключити і не турбуватись про них, адже вони не будуть ніяк навантажувати сам проект;
c) Тестування проекту значно спрощене тим, що майже весь код, що відповідає за середовище перенесений до самого фреймворка, і це дає змогу зосередитись на бізнес логікі проекту;
d) Веб-розробка за допомогою Spring має архітектуру MVC, що дає проекту ще більше модульності.

Хоча цей фреймворк є найбільшим та надпотужним для розробки майже будь-яких додатків мовою програмування Java, але й він не є ідеальним, і має деякі недоліки. Серед них можна виділити наступні:

a) Складність самого фреймворка. Його потрібно додатково вивчати, розробнику потрібно мати досить багато навичок для ефективного використання цього інструменту;
b) Висока залежність від XML, хоча починаючи від версії 3.0.0 розробники фрейморку надали можливість частково, а й іноді повністю, замінити конфігурування фреймворка з XML на анотації.

Маючи цю інформацію, можна зробити висновок, що хоча фреймворк і не є бездоганним, але є досить потужним та гнучким для розробки проектів будь якого масштабу, максимально зводячи процес розробки до саме бізнес-логіки, оминаючи досить складні і часто незрозумілі конфігурації застосунку [18].

Власне, маючи всі ці дані можна сказати, що навіть якщо не існує бібліотеки для вирішення тієї чи іншої проблеми завжди існує багато

способів створити рішення самому завдяки надпотужним інструментам, що надаються як і мовою, так і вже існуючими рішеннями а також великий базі розробників, які щодня продукують «контент» і можуть допомогти з проблемами.

Також для збірки проекту та зручного підключення сторонніх бібліотек був обраний такий інструмент як Maven. Maven - це інструмент автоматизації та управління, розроблений Apache Software Foundation. Maven - це потужний інструмент управління проектами, який базується на POM (project object model). Він використовується для побудови проектів, визначення залежностей та документування [19].

Серед переваг цього продукту можна визначити наступні:

a) Легко підключати нові бібліотеки, все, що для цього треба це лише записати код залежності у файл конфігурації;
b) Дозволяє легко збирати проект у будь-яких середовищах, без потреб перейматись про залежності, процеси, тощо;
c) Надає вичерпну інформацію щодо кожного кроку який виконується під час збірки та розгортання, що значно спрощує процес дебагінгу;
d) Дуже велика база даних фреймворків;
e) Підвищує загальний рівень продуктивності проекту.

Проте, на жаль, цей інструмент має і деякі недоліки, що час від часу створюють проблеми, хоча і не достатньо суттєві аби не обирати Maven. Серед потенційних недоліків можна визначити наступні:

a) Maven вимагає окремого встановлення на ОС, щоб працювати;
b) Неможливі використати його для додавання бібліотек та фреймворків які не присутні в його базі даних;
c) Maven відносно повільний.

Можна зробити висновок, що Maven є досить потужним варіантом для подальшої розробки програмного забезпечення.

3.2 Архітектура системи

Архітектурним рішенням для заданої системи буде MVC (Model-View-Controller, Модель-Представлення-Контролер) паттерн. Шаблон проектування MVC визначає, що додаток складається з моделі даних, інформації для представлення та інформації для керування. Шаблон вимагає, щоб кожна з цих частин була відокремлена в окремі об'єкти [29]. Цей шаблон є одним із найбільш розповсюджених стандартів для розробки веб-додатків адже він дозволяє прозоро розділяти зони впливу під час розробки та масштабування. Схема взаємодії користувача із системою наведена на рис. 3.2.

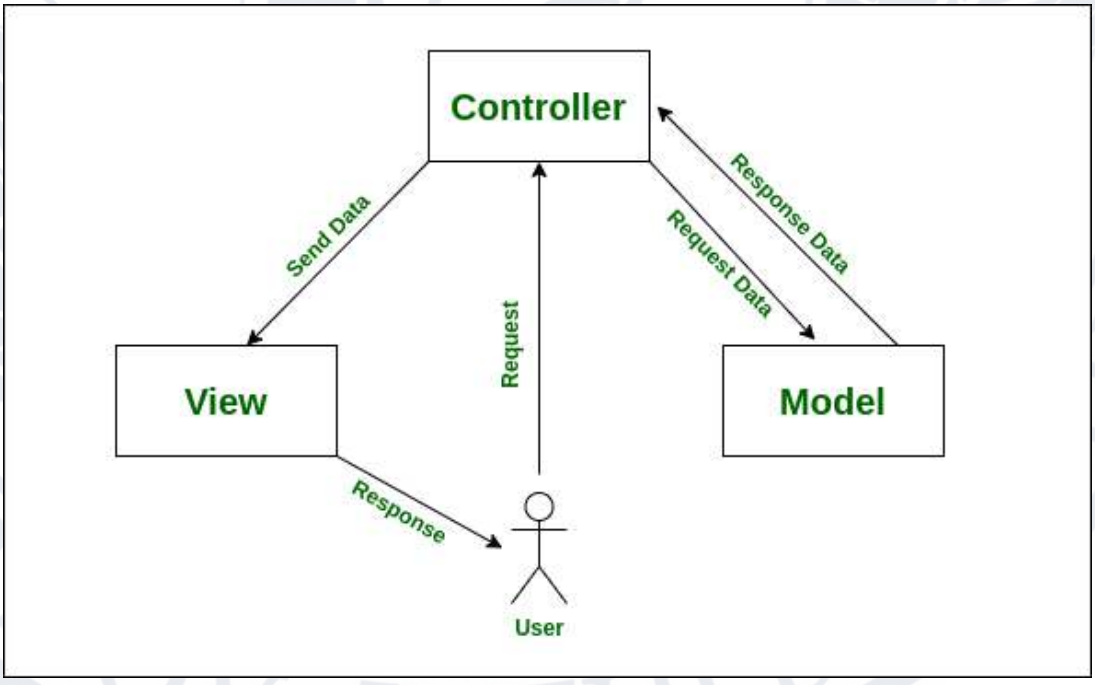


Рисунок 3.2 – взаємодія користувача із системою на базі MVC

Рівень «Модель» містить чисті дані додатку та логіку, що описує як ці дані мають бути представлені. Це просто чисті дані, що передаються по

всьому додатку, наприклад, від серверу до бази даних або від інтерфейсу до серверу. У цій конкретній програмній реалізації роботу збереження та передачі інформації буде виконувати так зване POJO (Plain-old-java-object), що є, по суті, звичайний об’єктом на мові програмування Java.

Рівень «Представлення» представляє дані моделі користувачеві. Цей рівень знає як і де отримати доступ до даних, але цей рівень не знає, що ці дані означають і що кінцевий користувач може з ними робити. Все ще робить цей рівень це просто відображає інформацію. У цій реалізації не буде ніяких html або jsp сторінок для відображення, але будуть json файли. Приклад такого представлення наведений на рис. 3.3.

Рисунок 3.3 – представлення даних у вигляді JSON-файлу

Рівень «Контролер» існує як поєднання між моделлю та представленням. Він «слухає» події, що викликані певними діями і виконує певні дії пов’язані з цією подією. Здебільшого контролер викликає відповідний метод для опрацювання моделі. Оскільки представлення і модель через механізм повідомлень, результат виклику контролера відображається майже одразу. В цій реалізації будуть розроблені звичайні методи для роботи з даними.

3.3 Модуль «Моделі»

Як вже було зазначено раніше, ця частина програми служить для зберігання та відправки різноманітних даних. Майже всі моделі у програмного забезпеченні є незалежними одна від іншої, за окремими виключеннями. Серед існуючих моделей представленні наступні: FoundProfanityDictModel, InputModel, OutputModel, PongModel, StaticDataInitModel, WordInputModel, WordOutputModel. Як видно із рис. 3.4, зв’язок між існує виключно між моделями OutputModel та FoundProfanityDictModel, де друга є частиною першої.

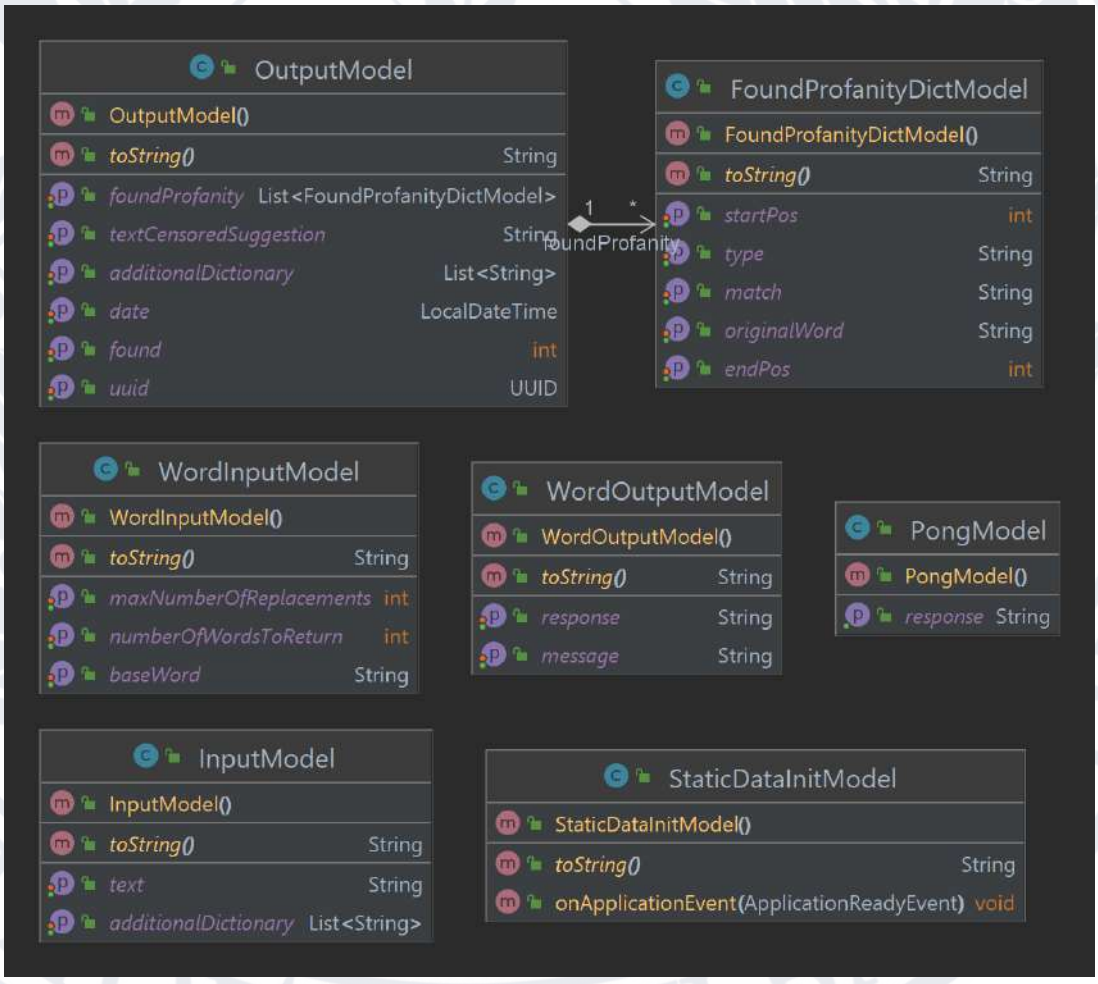


Рисунок 3.4 – Рівень «моделі»

Власне, кожна із заданих моделей буде розглянута окремо та більш детально, адже діаграма не відображає все у деяких випадках. Опис моделей буде наведений нижче:

- FoundProfanityDictModel

  Ця модель відповідає за зберігання інформації про словник виявлених лайливих слів. Словник складається із наступних аргументів:

  - private String originalWord – це аргумент зберігає оригінальне слово, а саме те слово, що було у деякому заданому початковому тексті;
  - private String match – це аргумент, що зберігає інформацію про слово за яким було виявлено лайку. Це, по суті, пара до минулого аргументу. Наприклад, якщо минуле слово було «m0r0n», то його парою за якою воно було виявлено буде слово «moron»;
  - private String type – цей аргумент зберігає інформацію про класифікацію лайливого слова. Слова підпадають під одну із п’яти наступних категорій: «adult», «insult», «intolerance», «global», «custom»;
  - private int startPos – цей аргумент зберігає початкову координату у тексті конкретного слова. Ця координата це номер початкового символу цього слова відносно початку тексту;
  - private int endPos – аналогічно минулому, цей аргумент зберігає координату слова, але вже кінцеву відносно початку тексту.

- InputModel

  Ця модель відповідає за збереження інформації про вхідні до програми значення, якщо точніше, то за дані що надходять для метода, що відповідає за пошук лайливих слів. Складається із наступних аргументів:

  - private List<String> additionalDictionary – цей аргумент зберігає масив рядків які користувач може сам визначити, аби програмне забезпечення також їх вважало лайкою та знаходило. Існування цього масиву зумовлене не ідеальністю програми та бажанням надати більшої гнучкості кінцевому користувачеві. Цей вхідний параметр не є обов'язковим і за умов його відсутності програма працюватиме коректно. Слова, що були виявлені за допомогою цього масиву будуть позначені у результаті під типом «custom»;
  - private String text – цей аргумент зберігає для подальшого опрацювання вхідний текст. На вхідний текст накладені деякі обмеження які будуть розглянуті пізніше, під час розгляду логіки валідації даних. Також варто зауважити, що цей параметр є обов'язковим для вхідної моделі і без нього програма не працюватиме, але лише поверне відповідну помилку, що вказує на відсутність тексту.

- OutputModel

  Ця модель відповідає за зберігання інформації, що утворюється під час пошуку лайки у тексті, та складається із наступних полів:

  - private UUID uuid – унікальний ID, що надається кожному успішному виклику методу для полегшення можливого пошуку минулих результатів або є відладки під час розробки;

- private LocalDateTime date – цей аргумент зберігає точну дату у форматі «yyyy-MM-dd'T'HH:mm:ss.SSSXXX» успішного виклику методу;
- private String textCensoredSuggestion – цей параметр є текстом із застосованою до нього цензурою (заміна усієї знайденої лайки на символ «*»). Слова, що не підпадають під визначення лайки цій цензурі не піддаються;
- private int found – цей параметр зберігає загальну кількість лайливих слів, що були знайдені у заданому тексті;
- private List<FoundProfanityDictModel> foundProfanity – цей аргумент зберігає данні по кожному окремому знайденому лайливому слову у тексті. Детальна структура цього масиву була описана вище;
- private List<String> additionalDictionary – цей аргумент відповідає за повернення власного словника користувача. Детальніше про цей словник також було написано вище.

- PongModel

  Ця модель існує виключно для більш простої перевірки, чи працює взагалі програмне забезпечення. Модель складається із одного поля:
  - private String response – це завчасно створена відповідь сервера на будь який запит до контролера пов’язаного із моделлю.
- StaticDataInitModel

  Хоча ця модель і не має жодних аргументів у класичному їх розумінні, проте вона відіграє не менш важливу роль тим, що відповідає за ініціалізації даних з якими потім буде працювати програма. До таких даних входить словник слів, його розбиття на підсловники а також завантаження пар візуально схожих символів, які є ключової ідеєю програмного забезпечення. Після цієї ініціалізації програма буде здатна звертатись у read-only режимі до

таких глобальних аргументів як «globalDictionary» та «visuallySimilarCharacters».

- WordInputModel

  Ця модель відповідає за збереження вхідних даних для використання їх для побудови слова, що базується на введеному, але містить з собі деяку кількість замін аби мати більші шанси обійти фільтри інших сервісів. До аргументів належать наступні:

  - private String baseWord – цей параметр це і є вхідне слово, що надається користувачем та потім піддається замінам символів на візуально схожі;
  - private int maxNumberOfReplacements – цей аргумент відповідає за бажану кількість замін у базовому класі, хоча варто зауважити, що якщо це значення завелике, то воно буде проігнороване.

- WordOutputModel

  Ця модель відповідає за збереження результату роботи відповідного метода під час творення слова із замінами із деякого базового слова та складається із наступних аргументів:

  - private String response – це результуюче слово, що утворилось під час замін;
  - private String message – цей аргумент є свого роду попередженням, яке повертається у випадку, якщо нічого із словом зробити не вдається. Зазвичай він є пустим.

Власне, на цьому огляд моделей закінчено. Для кожної із існуючих був наведений детальний опис з поясненням про кожний існуючий аргумент та як він пливає на роботу і які ціль має.

3.4 Модуль «Контролери»

Контролери застосовуються для виклика відповідних методів бізнес-логіки програмного забезпечення. У програмі існує 2 контролера: PingController та ProfanityFilterController. Вони абсолютно незалежні один від одного і виконують повністю різні функції. Діаграма цих контролерів наведена на рис. 3.5.

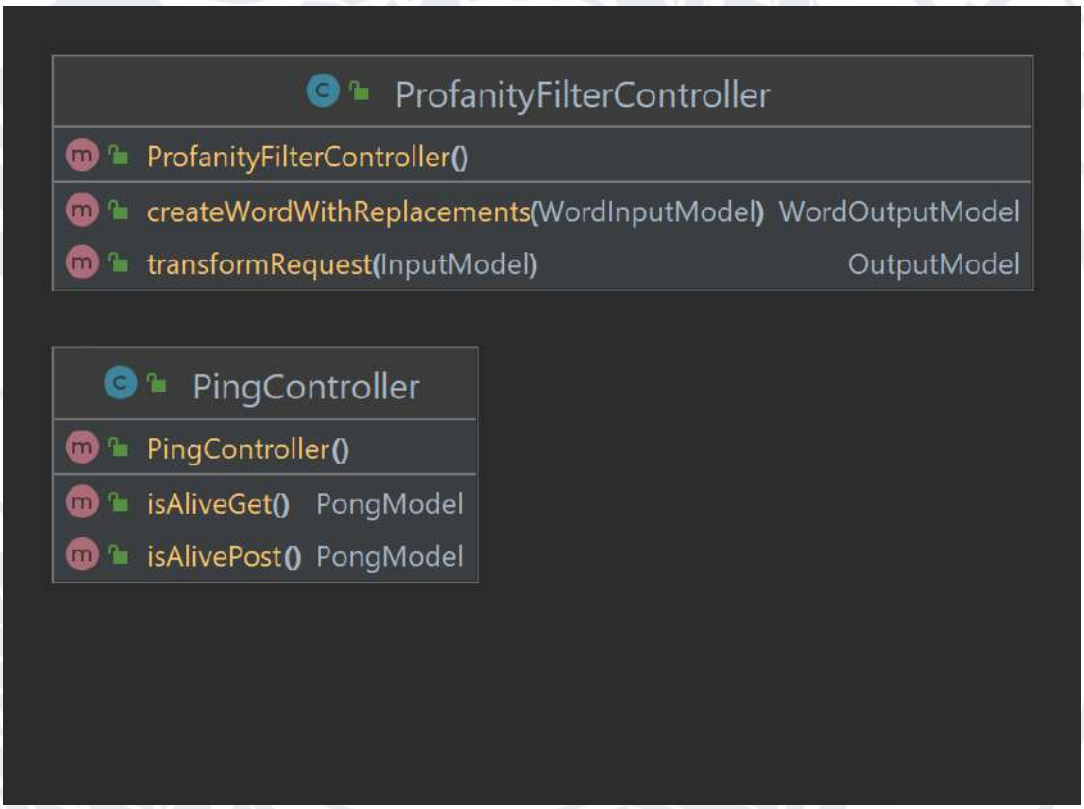


Рисунок 3.5 – Рівень «контролери»

Як видно із рис. 3.5, ці контролери містять по 2 методи кожний. Для контролера PingCotroller ці 2 методи виконують ідентичні функції перевірки чи сервер працює шляхом виконання дуже простого і швидкого запиту на відповідні кінцеві точки, тому їх вони не потребують детального огляду.

Контролер ProfanityFilterController має 2 методи – createWordWithReplacements та transformRequest. Перший викликає блок валідації даних, після чого йде попередня обробка даних і тільки після цього йде основний код бізнес-логіки виявлення замаскованих слів. Всі ці 3 етапи

будуть розглянуті більш детально у майбутніх розділах. Другий ж метод є дещо простішим, адже складається всього з двох кроків – валідації та бізнес логіки, ніякої попередньої обробки у ньому не передбачено. Цих 2 кроки також будуть розглянуті більш детально у майбутніх розділах.

3.5 Модуль «Виключення» або обробник помилок

Цей модуль виконує функції створення і відправлення повідомлень про помилки у легко формі, що легко буде зчитуватись користувачем з відповідним HTTP кодом, щоб спростити процес пошуку, що саме пішло не так та як з цим можна розібратись. Також важливою частиною цього модулю є клас, що наслідує клас ResponseEntityExceptionHandler адже завдяки цьому взагалі є можливість відображати власні помилки замість стандартних вбудованих, що не завжди відповідають контексту і не дуже інформативні. Цей клас-наслідник, ControllerAdvisor, містить інформацію про всі можливі власні помилки створені в рамках проекту та допомагає створювати ту структуру відповіді, що потребується.

У програмному забезпеченні реалізовані 3 власних типи помилок: ParamDoesNotExist, ParamTooLongException, TextException. Діаграма помилок та їх обробника наведена на рис. 3.6.

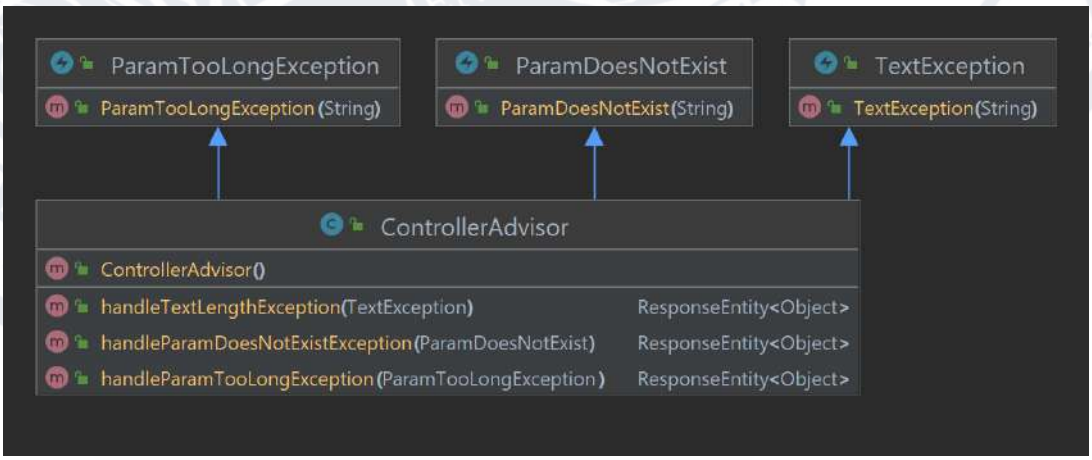


Рисунок 3.6 – Обробник помилок та власні помилки

Ці помилки використовуються під час валідації вхідний даних та майже ніколи у інших місцях програми. Кожна із наведених помилок сигналізує користувачеві про певну проблему із даними. Помилка ParamDoesNotExist дає знати користувачеві, що введений параметр є занадто довгим і використовується в обох методах основного контролера для валідації тексту, що має ліміт 160 символів, та базового слова, що має ліміт у 15 символів. Приклад помилки із HTTP кодом 400 Bad Request яку можливо отримати наведений на рис. 3.7.

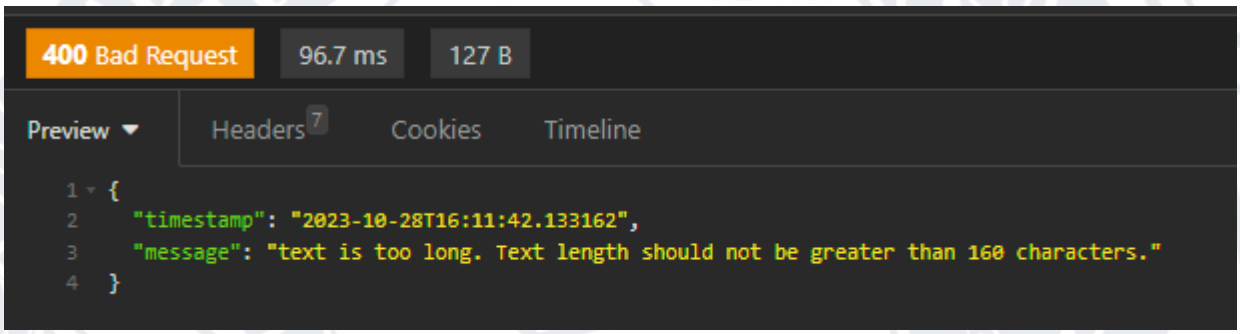


Рисунок 3.7 – Помилка «TextException»

У випадку якщо обов’язкового параметру не буде надано і робота програми буде неможлива, то буде повернута помилка із HTTP кодом 422 Unprocessable Entity як наведено на рис. 3.8.

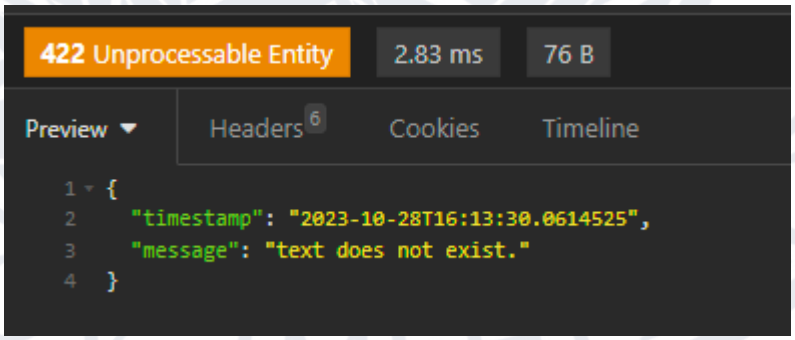


Рисунок 3.8 – Помилка «ParamDoesNotExist»

У випадку деяких помилок, як наприклад Timeout буде використаний вбудований обробник, що поверне доволі загальний опис ситуації, і у випадку непередбачуваних критичних помилок буде повернута помилка 500 Internal Server Error. Хоча в ідеалі такого не повинно ставитись, і всі можливі ситуації мають бути покриті власними помилками, все одно іноді можуть виникнути баги зі сторони фізичного серверу, що приведе до чого критичного. Загалом, обробник створений для спрощення взаємодії користувача із сервісом, адже у випадку будь-якої помилки її можливо побачити у логах серверу, куди користувачам доступу немає.

3.6 Модуль «Utils» або валідація та бізнес-логіка

У цьому модулі існує загалом 6 класів: ModelValidator, PluralsSingulars, PreProcessor, TextProcessor, WordReplacementModelValidator, WordsCreator. Рис. 3.9 показує структуру цих класів та їх взаємодію. із цих класів виконує свою функцію та працює у синергії з деякими іншими.

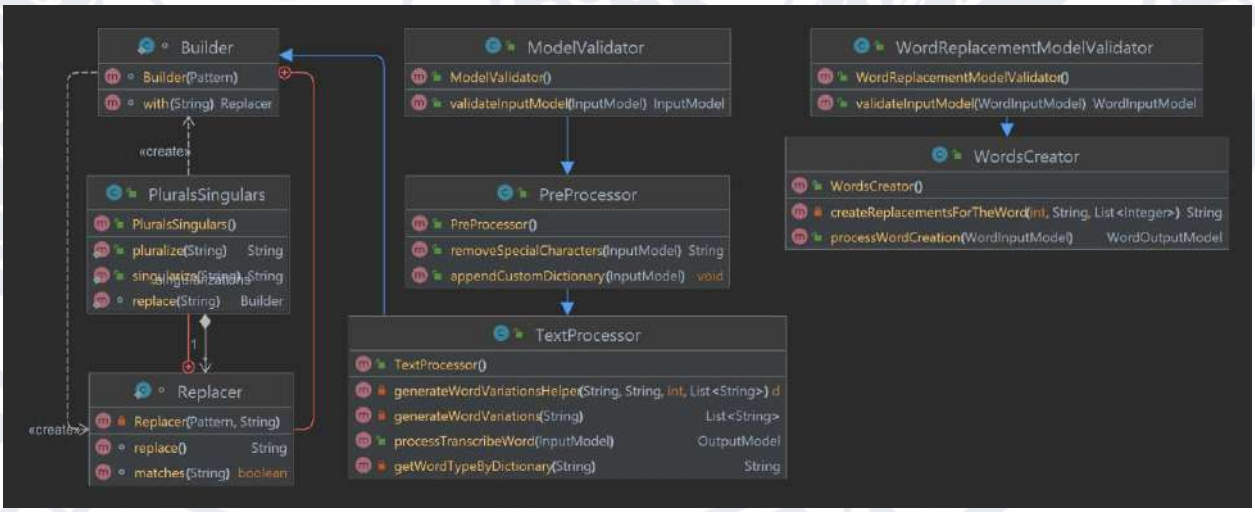


Рисунок 3.9 – Структура класів бізнес-логіки та їх взаємодія

Так, для контролера «ProfanityFilterController», що був вже загданий раніше, існує 2 методи: transformRequest (що виконує пошук прихованих

лайливих слів) та createWordWithReplacements(що самостійно створює слово із замінами). Цих 2 методи будуть розглянуті на рівні їх логіки. Перший з них використовує наступні класи на рівні свого методу (перераховані у порядку виклику): ModelValidator, PreProcessor та TextProcessor. Власне в цьому порядку вони і будуть розглянуті.

ModelValidator – це клас, який, зрозуміло із назви, валідує вхідний об’єкт на відповідність до певних норм. Серед таких правил наявні всього 2, перше це перевірка на наявність тексту у вхідній моделі даних, і у випадку якщо його немає, то викид помилки ParamDoesNotExist, адже текст є обов’язковим для подальшої роботи програми; наступна частина валідації перевіряє чи текст не є більшим ніж 160 символів, адже як було наведено у розділі 1 – 160 символів є цілком достатнім для переважної більшості повідомлень. У випадку, якщо ж це правило не виконується, то викидується помилка TextException. У разі якщо всі перевірки пройшли успішно, то валідатор просто повертає всю вхідну модуль назад без будь яких змін.

PreProcessor – це клас, що виконує декілька попередніх дій, що мають бути виконані для наступних етапів роботи. Перша дія – видалення усіх спеціальних символів із тексту за допомогою наступного регулярного виразу: "[@#$\\\\/<>_`+|=\"№%^\\[\\]&*]", усі символи перераховані між першою та останньою квадратними дужками є тими, що перестануть існувати. Наступною дією є додавання користувацького словника (якщо то існує та має довжину не менше 1 елементу) до основного словника на час виконання програми. Очищення від цих даних відбувається після пошуку слів і перед поверненням результату користувачеві.

TextProcessor – це клас, що відповідає за основну логіку всього програмного продукту і має лише 1 публічний метод processTranscribeWord, що виконує більшість необхідних перетворень перед запуском бектрекінг алгоритму. Серед таких перетворень є розбиття тексту на слова, або ж

токени, шляхом застосування до нього наступного регулярного виразу "[-|\\.||_|,|+|;|:]+", що означає, що всі символи між знаками «|» будуть розцінюватись як сепаратори у тексті, і знак плюс «+» у кінці означає, що неважливо скільки цих символів буде йти один за одним, усі вони будуть розцінюватись як один сепаратор, це було зроблено аби уникнути ситуацій де могли виникати такі токени як просто пробіл « », або просто кома «,» та інші. Наступним кроком після токенізації йде перевірка чи є токени «схожими» на слова із словника шляхом визначення відстані Левенштейна, і якщо вона більша аніж максимальне її значне поділене на два, то тоді слово вважається не схожим і пропускається, аби не перевантажувати алгоритм непотрібними даними. Далі, під час аналізу цих слів вони зводяться до однини, якщо вони у множині за допомогою класу PluralsSingulars, що містить у собі логіку перетворення одними у множина та навпаки для англійської мови, а також списки виключенням із правил, що перетворюються бо своїм унікальним алгоритмам та список тих слів, які не можуть бути переведені в з одного стану у інший. Далі, шляхом виконання бектрекінг алгоритму, зі слова вилучаються заміни на символи, що найбільше схожі одні на інших та визначається чи представлене отримане слово у словнику лайки. У випадку, якщо ні, то програма переходить до наступного токену, але у випадку, якщо так, то це слово йдуть наступні етапи опрацювання слова, серед яких визначення його приналежності до одного із п’яти можливих типів, визначення його координат відносно початку тексту, запис цієї інформації до результуючого словника а також до масиву які мають бути зацензуровані перед поверненням результату. Після того як слова були опрацьовані, то виконується фінальні етапи побудови відповіді для користувача, такі як заміна лайки на символ «*» а також заповнення модулі відповіді датою, унікальним ID та іншим. Після цього масив користувацького словника очищується і результати роботи повертається.

Описані вище 4 класи реалізовують основну функцію програмного забезпечення, проте існує ще одна функція, що було згадана раніше – створення замаскованих слів. Використовує цей функціонал тільки 2 класи: WordReplacementModelValidator та WordsCreator. Ці класи та їх логіку буде описано нижче.

WordReplacementModelValidator – цей клас аналогічний за призначенням першому описаному класу минулого контролера. У цьому випадку, валідації відбувається за наступними критеріями – перевірка чи у вхідній моделі наявне базове слово, без якого подальша робота неможлива, і якщо ні, то викидується помилка ParamDoesNotExist. Далі йде перевірка чи базове слово не є довшим за 15 символів, інакше буде повернута помилка ParamTooLongException. У випадку, якщо всі перевірки пройшли успішно, то як у аналогічному класі – буде повернута вхідна модель без будь яких змін.

WordsCreator – це клас, власне, створює заміни у слові на основі жадібного рандомізованого алгоритму. Це відбувається у декілька кроків, серед яких є наступні: визначення загальної кількості літер, що можна замінити у конкретному слові, порівняння цього значення із значенням maxNumberOfReplacements, що вводиться користувачем за потреби, і у разі якщо бажана кількість заміна є вищою ніж можлива, то тоді це значення ігнорується і програма виконує maxPossibleNumbersOfReplacements/2 кількість замін. Розділення на 2 виконується аби не спотворити слово, адже завелика кількість можна привести слово до стану, що його неможливо прочитати. У випадку, якщо кількість можливих замін є рівною нулю, то повертається попередження про це. Якщо все ж добре, то виконується N замін у слові. Логіка замін є наступною: обирання випадкової літера яку можливо замінити та підставляння замість неї іншої, схожої на неї із вибором із впорядкованого масиву схожих символів, далі індекс цієї літери позначається аби уникнути того, що він ще раз буде обраний для подальших

замін. Після проходження цього алгоритму, повертається вихідна модель користувачеві.

Висновок до розділу 3

У цьому розділі були розглянуті усі елементи програмної реалізації рішення зазначеної проблеми, було детально описано кожний модуль, клас та метод, що використовувався для досягнення визначеної мети.

РОЗДІЛ 4
ЕКСПЕРЕМЕНТАЛЬНІ ДОСЛДЖЕННЯ

4.1 Дослідження модулю пошуку прихованої лайки

Для тестування роботи програмного забезпечення, а саме модуля пошуку лайки у тексті, буде згенеровано 3 речення, що міститимуть деяку приховану та не приховану лайку. Для кращого порівняння, результати роботи власного продукту будуть порівнянні із результати для тих самих речень сервісів, що було розглянуті раніше, за виключенням API Ninjas адже він не може знайти найпростішу заміну. Для тестування системи були згенеровані наступні речення: «A fuck1ng man out a focklnq blue sweater fat ass sat at zh1tt1ng the st0p1d fuckinb desk.» (6 лайливих слів), «The ztubid fuekJng infant bItcb ass fock crawled into my ass the stupid fucking dense grass.» (8 лайливих слів), «Dude tried to fuekinq impress her ugly bitch face and fuekinb failed because the stubJd fucking c00n of is an Idjot.» (8 лайливих слів).

Результати перевірки першого речення:

- Readable.com

  Цей сервіс знайшов 4 лайливих слова та виділив їх. Результат наведений на рис. 4.1.

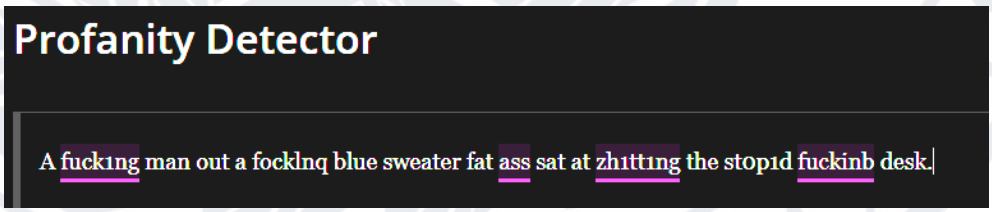


Рисунок 4.1 – результат першого речення за допомогою Readable.com

- Sightengine

  Цей сервіс знайшов 3 слова ти виділив їх. Результат наведений на рис. 4.2.

```
{
  "status": "success",
  "request": {
    "id": "req_fatxrzGlIlcQHOsetntPi",
    "timestamp": 1698257997.159814,
    "operations": 1
  },
  "profanity": {
    "matches": [
      {
        "type": "sexual",
        "intensity": "high",
        "match": "fucking",
        "start": 2,
        "end": 8
      },
      {
        "type": "insult",
        "intensity": "medium",
        "match": "fatass",
        "start": 41,
        "end": 47
      },
      {
        "type": "sexual",
        "intensity": "high",
        "match": "fuck",
        "start": 76,
        "end": 82
      }
    ]
  },
  "personal": {
    "matches": []
  },
  "link": {
    "matches": []
  }
}
```


Рисунок 4.2 – результат першого речення за допомогою Sightengine

- Webpurify

  Цей сервіс знайшов 3 слова та замінив їх на зірочки. Результати наведені на рис. 4.3.

```
{
  "rsp": {
    "@attributes": {
      "stat": "ok"
    },
    "method": "webpurify.live.replace",
    "lang": "en",
    "format": "rest",
    "found": "3",
    "text": "A ******* man out a focklnq blue sweater fat *** sat at zhitting the st0p1d ******* desk.",
    "api_key": "0d84cd89b362f09ab32d67edecd77d2b"
  }
}
```


Рисунок 4.3 – результат першого речення за допомогою Webpurify

- PurgoMalum

  Цей сервіс також знайшов 3 слова та замінив їх зірочками. Результат наведений на рис. 4.4.


```
{
  "result": "A ******* man out a fockinq blue sweater fat *** sat at zh1tt1ng the st0p1d ****inb desk."
}
```


Рисунок 4.4 – результат першого речення за допомогою PurgoMalum

- Власний застосунок

  Власний застосунок, у свою чергу, виявив 5 слів у заданому реченні та замінив їх зірочками. Результат наведений на рис. 4.5.


```
{
  "uuid": "2334d446-8736-4a5e-afc8-5e3d18d86367",
  "date": "2023-10-31T15:10:48.5548797",
  "textCensoredSuggestion": "A ******* man out a ******* blue sweater fat ass sat at ******** the ****** ******* desk.",
  "found": 5,
  "foundProfanity": [
    {
      "originalWord": "fuck1ng",
      "match": "fucking",
      "type": "insult",
      "startPos": 2,
      "endPos": 9
    },
    { ↔ 5 ↦ },
    { ↔ 5 ↦ },
    { ↔ 5 ↦ },
    { ↔ 5 ↦ }
  ],
  "additionalDictionary": []
}
```


Рисунок 4.5 – результат першого речення за допомогою власного застосунку

Результат перевірки другого речення:

- Readable.com

  Цей сервіс знайшов 4 лайливих слів та виділив їх. Результат наведений на рис. 4.6.

Рисунок 4.6 – результат першого речення за допомогою Readable.com

- Sightengine

  Цей сервіс знайшов 5 слів ти виділив їх. Результат наведений на рис. 4.7.

Рисунок 4.7 – результат першого речення за допомогою Sightengine

- Webpurify

  Цей сервіс знайшов 4 слова та замінив їх на зірочки. Результати наведені на рис. 4.8.

```
{
  "rsp": {
    "@attributes": {
      "stat": "ok"
    },
    "method": "webpurify.live.replace",
    "lang": "en",
    "format": "rest",
    "found": "4",
    "text": "The ztubid fuekJng infant bItcb *** **** crawled into my *** the stupid ******* dense gr***.",
    "api_key": "0d84cd89b362f09ab32d67edecd77d2b"
  }
}
```


Рисунок 4.8 – результат першого речення за допомогою Webpurify

- PurgoMalum

  Цей сервіс знайшов 3 слова та замінив їх зірочками. Результат наведений на рис. 4.9.


```
{
  "result": "The ztubid fuekJng infant bItcb *** fock crawled into my *** the stupid ******* dense grass."
}
```


Рисунок 4.9 – результат першого речення за допомогою PurgoMalum

- Власний застосунок

  Власний застосунок, у свою чергу, виявив 6 слів у заданому реченні та замінив їх зірочками. Результат наведений на рис. 4.10.


```
{
  "uuid": "204fa526-57b7-49d1-b757-0712c2ee4dd9",
  "date": "2023-10-31T15:01:57.0080459",
  "textCensoredSuggestion": "The ****** ******* infant ***** ass **** crawled into my ass the ****** ******* dense grass.",
  "found": 6,
  "foundProfanity": [
    {
      "originalWord": "ztubid",
      "match": "stupid",
      "type": "insult",
      "startPos": 4,
      "endPos": 10
    },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ }
  ],
  "additionalDictionary": []
}
```


Рисунок 4.10 – результат першого речення за допомогою власного застосунку

Результати перевірки третього речення:

- Readable.com

  Цей сервіс знайшов 3 лайливих слова та виділив їх. Результат наведений на рис. 4.11.

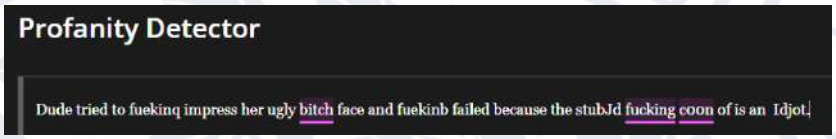


Рисунок 4.11 – результат першого речення за допомогою Readable.com

- Sightengine

  Цей сервіс знайшов 2 слова ти виділив їх. Результат наведений на рис. 4.12.

Рисунок 4.12 – результат першого речення за допомогою Sightengine

- Webpurify

  Цей сервіс знайшов 3 слова та замінив їх на зірочки. Результати наведені на рис. 4.13.

```
{
  "rsp": {
    "@attributes": {
      "stat": "ok"
    },
    "method": "webpurify.live.replace",
    "lang": "en",
    "format": "rest",
    "found": "3",
    "text": "Dude tried to fuekinq impress her ugly ***** face and fuekinb failed because the stubJd ******* **** of is an  Idjot.",
    "api_key": "0d84cd89b362f09ab32d67edecd77d2b"
  }
}
```


Рисунок 4.13 – результат першого речення за допомогою Webpurify

- PurgoMalum

  Цей сервіс знайшов 3 слова та замінив їх зірочками. Результат наведений на рис. 4.14.


```
{
  "result": "Dude tried to fuekinq impress her ugly ***** face and fuekinb failed because the stubJd ******* **** of is an  Idjot."
}
```


Рисунок 4.14 – результат першого речення за допомогою PurgoMalum

- Власний застосунок

  Власний застосунок, у свою чергу, виявив 7 слів у заданому реченні та замінив їх зірочками. Результат наведений на рис. 4.15.


```
{
  "uuid": "a0669660-cf0e-40d2-aaaa-eefefb23f1a7",
  "date": "2023-10-31T15:08:45.7191887",
  "textCensoredSuggestion": "Dude tried to ******* impress her ugly ***** face and ******* failed because the ****** ******* **** of is an  *****.",
  "found": 7,
  "foundProfanity": [
    {
      "originalWord": "fuekinq",
      "match": "fucking",
      "type": "insult",
      "startPos": 14,
      "endPos": 21
    },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ },
    { ↔ 5 ↔ }
  ],
  "additionalDictionary": []
}
```


Рисунок 4.15 – результат першого речення за допомогою власного застосунку

Можна підбити підсумки цього дослідження зробивши таблицю та порівнявши відсоток «попадання» по словам із прихованою та неприхованою лайкою. Данна таблиця 4.1 наведена нижче:

Таблиця 4.1 – Порівняння результатів перевірки власного сервісу та інших аналогічних

	Речення 1, 6 слів	Речення 2, 8 слів	Речення 3, 8 слів	% попадання
Readable.com	4/6	4/8	3/8	50%
Sightengine	3/6	5/8	2/8	45.5%
Webpurify	3/6	4/8	3/8	45.5%
PurgoMalum	3/6	3/8	3/8	41%
Власний застосунок	5/6	6/8	7/8	82%

Як видно із таблиці 4.1, власний застосунок хоча і не є ідеальним, але показує результати значно вище ніж ті, що можна отримати використовуючи сервіси, що є першими під час видачі результатів від Google.

4.2 Дослідження модулю заміни символів у словах

Наступним об’єктом перевірки є та частинам програми, що відповідає за створення слів із заміною. Знову ж таки будуть використані ті самі сервіси, що використовувались і раніше. Буде обрано 4 слів із словника з яким працює власний продукт, для кожного із цих слів буде згенеровано по 5 варіацій, даючи в результаті 20 слів, і ці слова будуть відправлені відповідним сервісам на перевірку та виявлення прихованої лайки.

Отже, серед базових слів будуть обрані наступні: «faggot», «bitch», «motherfucker», «dickhead», «retard». Їх було обрано базуючись на частоті використання та кількості букв у слові, адже не всюди є можливість згенерувати 5 варіацій. Отримані варіації вхідних базових слів наведені на рис. 4.16.

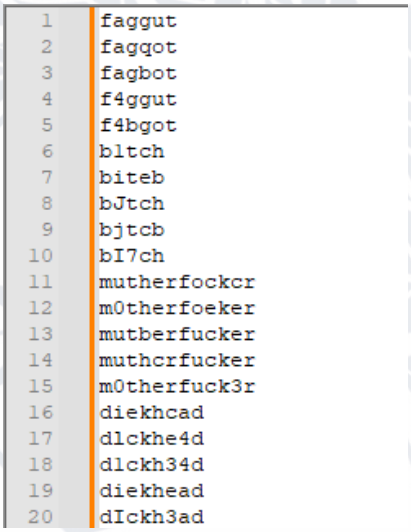


Рисунок 4.16 – результат генерування прихованої лайки

Отримавши ці результати, можна тепер відправляти їх у раніше згадані сервіси. Результати роботи цих сервісів будуть наведені нижче.

Readable.com виконав задачу доволі дивно, адже повністю пропустив всі варіації слова «faggot», можливо через те, що взагалі не розпізнає це слово як лайку а можливо з інших причин. В будь якому випадку цей сервіс знайшов всього 7 слів із 20. Результат наведений на рис. 4.17.

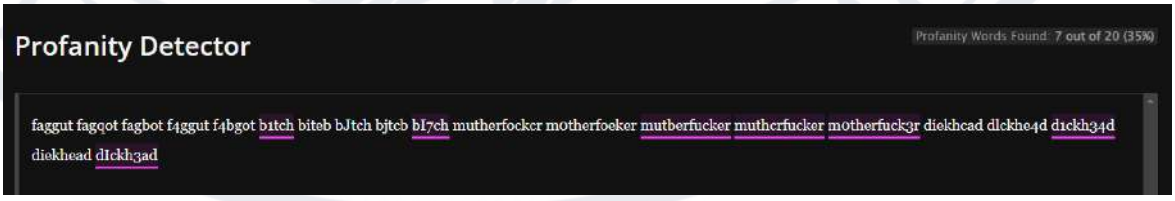


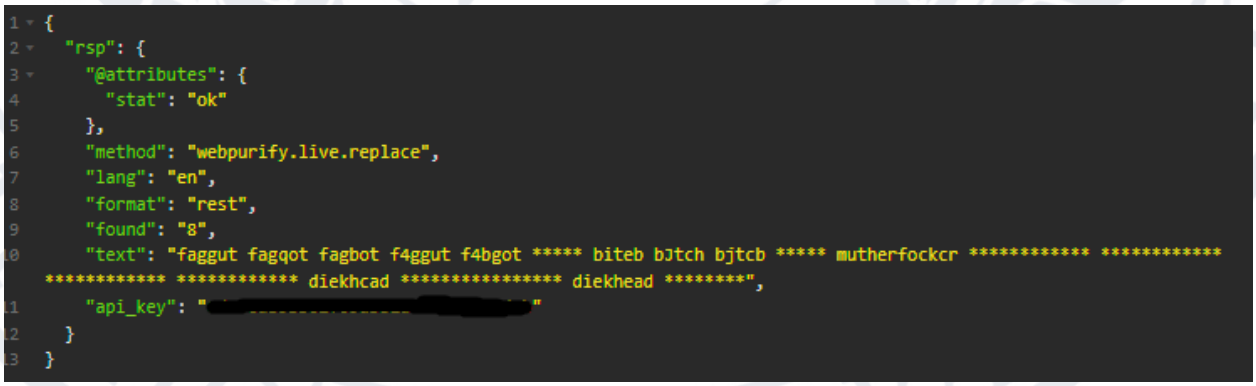
Рисунок 4.17 – результат роботи Readable.com у пошуку лайки серед згенерованих слів

Sightengine виконав задачу дещо краще, розпізнаючи слово «faggot» і знайшов загалом 10 слів. Результати наведені на рис. 4.18.

```
{
  "status": "success",
  "request": {
    "id": "req_f5Y19gPBWNoiqCBpKZx0n",
    "timestamp": 1699100475.395013,
    "operations": 1
  },
  "profanity": {
    "matches": [ ↤ 10 ↦ ]
  },
  "personal": {
    "matches": []
  },
  "link": {
    "matches": []
  }
}
```


Рисунок 4.18 – результат роботи Sightengine у пошуку лайки серед згенерованих слів

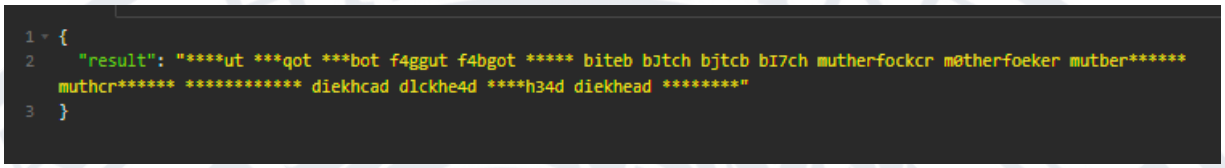
Webpurify також виконав задачу доволі посередньо через те, що взагалі не розпізнавав слово «faggot», що є досить дивним адже слово однозначно лайливе. Загалом сервіс знайшов 8 слів із 20. Результати на рис. 4.19.



```
{
  "rsp": {
    "@attributes": {
      "stat": "ok"
    },
    "method": "webpurify.live.replace",
    "lang": "en",
    "format": "rest",
    "found": "8",
    "text": "faggut fagqot fagbot f4ggut f4bgot ***** biteb bJtch bjtcb ***** mutherfockcr ************ ************ ************ ************ diekhcad **************** diekhead ********",
    "api_key": "[redacted]"
  }
}
```


Рисунок 4.19 – результат роботи Webpurify у пошуку лайки серед згенерованих слів

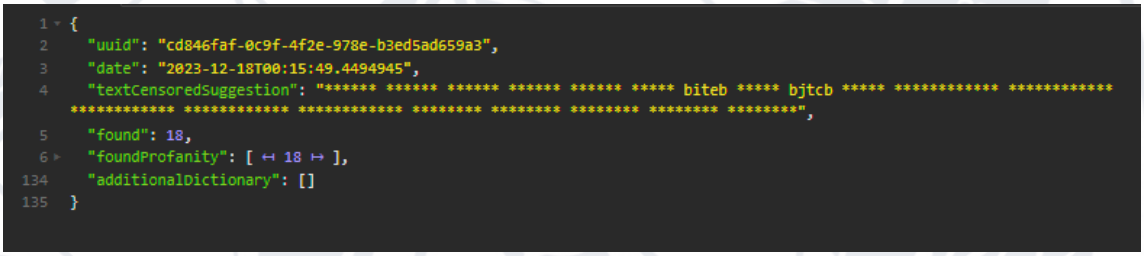
PurgoMalum зміг розпізнати слова на базі «faggot», проте мав суттєві проблеми із розумінням деяких інших слів, що дало результат у 9 слів із 20. Результат наведений на рис. 4.20.



```
{
  "result": "****ut ***qot ***bot f4ggut f4bgot ***** biteb bJtch bjtcb bI7ch mutherfockcr m0therfoeker mutber******
muthcr****** ************ diekhcad dlckhe4d ****h34d diekhead ********"
}
```


Рисунок 4.20 – результат роботи PurgoMalum у пошуку лайки серед згенерованих слів

Власний розроблений веб-додаток впорався майже з усім, знайшовши 18 із 20 таких слів.



```
{
  "uuid": "cd846faf-0c9f-4f2e-978e-b3ed5ad659a3",
  "date": "2023-12-18T00:15:49.4494945",
  "textCensoredSuggestion": "****** ****** ****** ****** ****** ***** biteb ***** bjtcb ***** ************ ************
************ ************ ************ ******** ******** ******** ******** ********",
  "found": 18,
  "foundProfanity": [ ↔ 18 ↔ ],
  "additionalDictionary": []
}
```


Рисунок 4.21 – результат роботи власної розробки у пошуку лайки серед згенерованих слів

Назагал, для підбиття підсумків та порівняння отриманих результатів можна побудувати таблицю із відсотком знайдених слів для отримання результату для кожного із сервісів та середнього для них, а також результатів власної розробки.

Таблиця 4.2 – Порівняння результатів пошуку згенерованої прихованої лайки

	Readable.com	Sightengine	Webpurify	PurgoMalum	Розробок
Відношення знайдених до загальної кількості	7/20	10/20	8/20	9/20	18/20
Результат і відсотках	35%	50%	40%	45%	90%
Середній результат у відсотках	42.5%				90%

Як видно із таблиці 4.2 – більшість сервісів не була спроможна знайти навіть половину із запропонованих слів, з тих чи інших причин. Хоча загалом результати і можна вважати непоганими, але цього однозначно буде недостатньо для якісного аналізу тексту на меті якого є пошук саме прихованої лайки, а не очевидної.

4.3 Тестування модулю пошуку прихованої лайки на великих обсягах даних

Для більш ґрунтовного тестування розробленого веб-сервісу був взятий розмічений список «Toxic Comments» від «Kaggle Toxic Comment Classification Challenge» [30]. Цей датасет має декілька полів які характеризують його приналежність до деякого типу (типів), серед таких полів наступні: «toxic», «severe_toxic», «obscene», «threat», «insult», «identity_hate». У цих полях значення 0 вказує, коментар не належить до

заданого типу, 1 означає, що належить. Важливо зауважити, що розміткою займались реальні люди, які можуть розуміти контекст написаного. На основі 116337 коментарів було отримано наступні результати щодо їх токсичності:

toxic = 1 коментарів – 11147;

severe_toxic = 1 коментарів – 1156;

obscene = 1 коментарів – 6171;

threat = 1 коментарів – 358;

insult = 1 коментарів – 5748;

identity_hate = 1 коментарів – 996;

Оскільки кожен коментар може мати одночасно більше ніж один такий параметр зі значенням 1, тому просто порахувати суму зазначених чисел вище не є коректним шляхом. Назагал коментарів які мають хоча б один параметр зі значенням 1 – 11827, тобто приблизно 10% від загальної кількості. Власне, наступним кроком буде пропуск цих речень через розроблений веб-сервіс та порівняння кількості вхідних речень із кількістю речень, що виявить розроблений сервіс. Оскільки початковий датасет не вказує на конкретні слова у реченнях, а лише подає загальну характеристику речення, то окремі слова порівнюватись не будуть. Запуски відбувались вже на реальних деплоях за допомогою сервіса Make. Також тестування було розбите на дві частини, де у першій частині тестувались всі коментарі які мали значення 1 для наступних категорій: obscene, threat, insult, identity_hate. Ці категорії були обрані тому, що вони майже нерозривно пов’язані із нецензурними словами. Інша частина була присвячена двом іншим категоріям: toxic та severe_toxic адже токсичність зовсім не зобов’язує використовувати лайку, навіть приховану. Спеціально для тестування ліміт по символам було збільшено до 500 символів. Збільшення його ще більше може негативно вплинути на продуктивність.

Кількість коментарів, що були взяті для першої частини тестування дорівнює 7673, із них 6822 пройшло перевірку на максимально 500 символів, і саме з ними далі працював розроблений веб-сервіс. У сервісі, що був використаний для тестування було створено 2 бази даних – для текстів, у яких власний застосунок успішно виявив щонайменше 1 лайливе слово (нижня) і для тих, де він не впорався з тих чи інших причин (верхня). Як результат, що показаний на рис. 4.22, загальна кількість коментарів де було виявлено лайку складає 5156, а кількість де не було – 1666. Із цих цифр видно, що виявлено лайку було приблизно у 75.6% коментарів, відповідно 24.4% речень були пропущені. Серед пропущених, з тих чи інших причин, коментарів досить показовим є найперший пропущений, що має такий вигляд: «Hi! I am back again! Last warning! Stop undoing my edits or die!». З цього коментаря видно, що жодне слово не є лайливим у класичному розумінні, проте люди, які розмічали датасет, як було зазначено раніше, прочитали контекст повідомлення та зрозуміло, що воно цілком підпадає під тип «threat», проте програма не може аналізувати контекст і тому пропустила це речення.

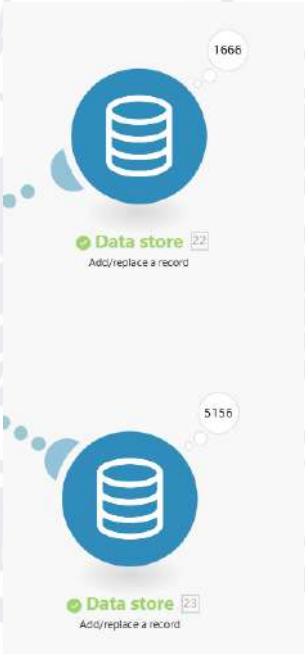


Рисунок 4.22 – результат аналізу першої частини коментарів

Наступною частиною є перевірка коментарів, що підпадають під інших 2 категорії. Розміщення баз даних залишилось незмінним. Кількість коментарів, що підпали під умову, що значення toxic та/або severe_toxic дорівнює 1 сягає 11147 одиниць. Після відкидання коментарів де довжина більше ніж 500 символів залишилось 9812 коментарів, вони і були пропущені через розроблений сервіс. Оскільки, як були зазначено раніше, токсичність ще зовсім не означає наявність лайливих слів і сильно залежить від контексту, то й результати роботи не є вражаючими. Кількість текстів із виявленою лайкою складає 5975 одиниць, відповідно кількість пропущених складає 3837 одиниць, що у відсотках приблизно дорівнює 61% та 39% відповідно. Результати показані на рис. 4.23. Серед пропущених речень багато на кшталт наступних: «Well you are ridiculous, in fact I suspect that you are Calton, please block me, I dont care....», «Don't post any garbage on my page!» та інших. Це речення про контекст, а не про просто написання лайки, можливо навіть прихованої, і тому є цілком очевидним те, що розроблений веб-сервіс їх пропускає.

Рисунок 4.23 – результат аналізу другої частини коментарів

Також перевірені і нетоксичні коментарі. Узагальнені результати перевірки наведено в табл. 4.3. З таблиці видно, що запропонований алгоритм рідко коли виявляє лайливе слово в нейтральних коментарях – коментарях третьої групи. Образливі та погрозливі коментарі – коментарі з другої групи у 75.6% випадків містять лайливе слово, яке виявляється запропонованим алгоритмом. Більшість токсичних коментарів першої групи також містить правильні або спотворені лайливі слова. Таким чином, виявлені за запропонованим алгоритмом приховані лайливі слова можуть розглядатися як додатковий інформативний атрибут для створення моделей виявлення токсичного контенту в соціальних мережах. Принциповою відмінністю запропонованого алгоритму є те, що він базується на аналітичному підході, а не на індуктивному. Тому, він здатен виявляти не лише відомі спотворені лайливі слова, але і нові, які будуть створені на основі нових варіантів заміни символів.

Таблиця 4.3 – Результати тестування на датасеті Kaggle Toxic Comment

Група	Тип коментаря	Кількість коментарів	Виявлено лайливе словом	Не виявлено лайливого слова
1	Toxic Severe toxic	9812	5975 61%	3837 39%
2	Obscene Threat Insult Identity hate	6822	5156 75.6%	1666 24,4%
3	Neutral	50030	1604 3.2%	48426 96.8%

Висновок до розділу 4

У цьому розділі були протестовані результати, що надає власне програмне забезпечення, а також були порівнянні ці результати із тими, що можна отримати використавши аналогічні сервіси. Було протестоване власне програмне забезпечення на великих обсягах даних з датасету від Kaggle.

За результатами розділу опубліковано тези [31].

ВИСНОВКИ

В результаті виконання цієї роботи були розглянуті існуючі сервіси, що виконують функції модерації тексту, в тому числі із можливістю знаходження прихованих лайливих слів.

Були проаналізовані існуючі матриці сплутувань символів англійського алфавіту та на їх основі створені власні для наступних пар: «верхній-верхній регістри», «нижній-нижній регістри», «верхній-нижній регістри», «верхній регістр-цифри», «нижній регістр-цифри». Базуючись на власних матрицях кожній парі символів було наданий коефіцієнт, що визначає наскільки символи у цій парі візуально схожі.

Після оцінки існуючих рішень та побудови матриць сплутувань були розглянуті алгоритми для побудови слів, що могли б бути використані при розробці власного програмного забезпечення і після аналізу існуючих методів був обраний той, що підходить найкраще.

Після визначення із алгоритмом розпочалась розробка веб-сервісу із підтримкою REST для взаємодії із користувачами та іншими системами. Веб-сервіс був розроблений на мові програмування Java із використанням Spring Boot та Maven.

Після розробки сервісу, результати його роботи були протестовані та порівнянні із результатами роботи аналогічних, раніше розглянутих, сервісів а також розроблений веб-сервіс був протестований на великій базі коментарів взятій у Kaggle Toxic Comment Classification Challenge.

СПИСОК ЛІТЕРАТУРИ

1. When Was Texting Invented: The History of Texting [Електронний ресурс] – Режим доступу до ресурсу: https://www.textmagic.com/blog/the-history-of-texting-from-telegraphs-to-enterprise-sms/
2. A Brief History of Text Messaging [Електронний ресурс] – Режим доступу до ресурсу: https://www.mobivity.com/mobivity-blog/a-brief-history-of-text-messaging
3. Cyberbullying, Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Cyberbullying
4. “Students who ended their lives in «cyber violence»... The perpetrator who mocked even death”, SBS News, 9 вересня 2018 року [Електронний ресурс] – Режим доступу до ресурсу: https://news.sbs.co.kr/news/endPage.do?news_id=N1004932661%20
5. Investigating the role of swear words in abusive language detection tasks [Електронний ресурс] – Режим доступу до ресурсу: https://link.springer.com/article/10.1007/s10579-022-09582-8
6. M. Perea, Duñabeitia J. A., M. Carreiras. R34D1Ng W0Rd5 W1Th Numb3R5. // Journal of Experimental Psychology: Human Perception and Performance, Vol. 34, p. 237–241, 2008.
7. S. Saha, S. Basu, M. Nasipuri. Automatic Localization and Recognition of License Plate Characters for Indian Vehicles. August 2011 [Електронний ресурс] Режим доступу до ресурсу: https://www.researchgate.net/publication/261760841_Automatic_Localization_and_Recognition_of_License_Plate_Characters_for_Indian_Vehicles
8. Jack M. Loomis. Analysis of tactile and visual confusion matrices. // Perception & Psychophysics, Vol. 31, p. 41-52, 1982.

9. J. T. Townsend. Theoretical analysis of an alphabetic confusion matrix. // Perception & Psychophysics, Vol. 9, p. 40-50, 1971.
10. L.R.Geyer. Recognition and confusion of the lowercase alphabet. // Perception & Psychophysics, Vol. 22, p. 487-490, 1977.
11. Dunn-Rankin, P., Leton, D. A., Shelton, V. F. Congruency factors related to visual confusion of English letters. // Percept Mot Skills, Vol. 26, p. 659–666, 1968.
12. Urban dictionary [Електронний ресурс] – Режим доступу до ресурсу: https://www.urbandictionary.com/
13. Java, Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Java
14. The top programming languages [Електронний ресурс] – Режим доступу до ресурсу: https://octoverse.github.com/2022/top-programming-languages
15. Spring Framework [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework
16. MVC Design Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/mvc-design-pattern/
17. The Model View Controller Pattern – MVC Architecture and Frameworks Explained [Електронний ресурс] – Режим доступу до ресурсу: https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/
18. What are Java Spring framework advantages and disadvantages? [Електронний ресурс] – Режим доступу до ресурсу: https://www.educative.io/answers/what-are-java-spring-framework-advantages-and-disadvantages

19. What is Maven? [Електронний ресурс] – Режим доступу до ресурсу: https://www.learntek.org/blog/what-is-maven/
20. “Definition of Profanity” [Електронний ресурс] – Режим доступу до ресурсу: https://www.merriam-webster.com/dictionary/profanity
21. G.Patrick Nerbonne, Nicholas M. Hipskind. The use of profanity in conversational speech. // Journal of Communication Disorders, Vol. 5, p. 47-50, 1972.
22. C. Emmery, B. Verhoeven, G. De Pauw, G. Jacobs, C. Van Hee, E. Lefever, B. Desmet, V. Hoste, W. Daelemans. Current limitations in cyberbullying detection: On evaluation criteria, reproducibility, and data scarcity. // Lang Resources & Evaluation, Vol. 55, p. 597-633, 2021.
23. E. Gurari, “CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms” [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20070317015632/http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html#QQ1-51-128
24. Beasley, J. E. "Integer programming" [Електронний ресурс] – Режим доступу до ресурсу: http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html
25. Combinatorial optimization [Електронний ресурс] – Режим доступу до ресурсу: https://www.engati.com/glossary/combinatorial-optimization
26. Recursive Backtracking and Enumeration [Електронний ресурс] – Режим доступу до ресурсу: https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1218/lectures/11-backtracking1/Lecture11Slides.pdf
27. L. Filipe Rodrigues Ribeiro, “Graph-based Approaches to Text Generation” [Електронний ресурс] – Режим доступу до ресурсу: https://tuprints.ulb.tu-darmstadt.de/21498/5/PhD_thesis_Leonardo_Ribeiro.pdf

28. Graph-based word hypotheses generation, [Електронний ресурс] – Режим доступу до ресурсу: https://www.ai.rug.nl/~lambert/recog/hwr-tutor/word-hyps.html
29. MVC Design Pattern, [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/mvc-design-pattern/
30. Toxic Comment Classification Challenge [Електронний ресурс] – Режим доступу до ресурсу: https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
31. Бездушний В.О., Штовба С.Д. Виявлення прихованої лайки в текстових повідомленнях за аналізом візуально-подібних символів. Матеріали Четвертої всеукраїнської науково-практичної конференції «Комп'ютерні технології обробки даних», Вінниця: ДонНУ імені Василя Стуса, 2023. *Прийнято до друку.*

Додаток 2 до наказу
від «31» березня 2023 року
№119/05

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)