КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

«На правах рукопису»

Робота допущена до захисту в ЕК
рішенням кафедри радіотехніки та радіоелектронних систем
від _____ 2024 року, протокол № ___.
Завідувач кафедри доктор фіз.-мат. наук, професор
_____ Ігор АНІСІМОВ

ДИПЛОМНА РОБОТА МАГІСТРА
на тему:
«ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ АНАЛІЗАТОРА ВИХІДНИХ ТЕКСТІВ ПРОГРАМ»

Виконав:
студент 2-го курсу магістратури
денної форми навчання
спеціальності 172 - Телекомунікації та радіотехніка
ОНП «Інформаційна безпека телекомунікаційних систем і мереж»
Бакаєв Родіон Олександрович

Науковий керівник:
доктор тех. наук, професор
Погорілий Сергій Дем'янович

Рецензент:
кандидат фіз.-мат. наук, доцент
Шаповалов Андрій Петрович

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент ___ Родіон БАКАЄВ

Київ 2024

РЕФЕРАТ

Дипломна робота магістра: 31 с., 13 рисунків, 6 джерел, 4 додатки.

АЛГОРИТМ, АНАЛІЗАТОР, ШТУЧНИЙ ІНТЕЛЕКТ(ШІ), PYTHON, ВИХІДНИЙ КОД, ЕМПІРИЧНЕ ДОСЛІДЖЕННЯ

Об’єкт розроблення – програмна реалізація аналізатора вихідного коду програм.

Мета роботи – розробка системи для автоматичного зчитування вхідний код та перетворення його в природну мову за допомогою алгоритмів або штучного інтелекту.

ЗМІСТ

ВСТУП

Проблематикою всього дослідження було відслідковування еволюції програмного забезпечення. Через нехтування програмістами написання точних коментарів до внесених змін контроль історії оновлень для подальшого її аналізу стає складнішим. Для реалізації генератору коментарів за допомогою алгоритму або з використанням штучного інтелекту потрібно розробити систему, яка буде зчитувати вхідний код та перероблювати його в природню мову. Таким чином з основної проблематики всього дослідження було визначена тема роботи «Дослідження та реалізація аналізатора вихідних текстів програм».

1. ОСНОВНІ ПРИНЦИПИ РОБОТИ АНАЛІЗАТОРА

1.1 Що повинен вміти аналізатор

Аналізатор повинен вміти зчитувати код та перетворювати логічні та математичні оператори, синтаксичні конструкції мови програмування та самодокументовані назви змінних, функцій, методів, класів тощо. у більш зрозумілу для аналізу алгоритмом або штучним інтелектом(ШІ) Natural Language(NL). Natural Language відноситься до мови, яку люди використовують для спілкування між собою, таку як українська, англійська, іспанська, китайська та багато інших. В інформаційних технологіях термін "Natural Language" часто пов'язується із обробкою природної мови (Natural Language Processing, NLP).

Також аналізатор повинен вміти ігнорувати NL яка написана людиною для людини, а саме строкові змінні з певним текстом, певні символи, коментарі та іншу інформацію яка повинна надаватись людині в тому вигляді, в якому вона є.

1.2 Правила написання коду для аналізатора

Самодокументовані назви сутностей правильно будуть оброблятися аналізатором у тому випадку, якщо будуть дотримані загальні правила для коректного введення назв змінних у програмуванні:

1. Спісність (Camel Case, Snake Case):
   - Camel Case: Починайте кожне нове слово з великої літери, крім першого слова. Наприклад, myVariableName.
   - Snake Case: Використовуйте нижні підкреслення між словами. Наприклад, my_variable_name.

2. Зрозумілі та описові назви:
   - Назви змінних повинні бути описовими, щоб інші програмісти (або ви самі через певний час) могли зрозуміти призначення змінних.

3. Уникайте використання некоректних скорочень або абревіатур, які можуть бути неперевідні для інших розробників. Наприклад, використовуйте userCount замість usrCnt.

4. Не використовуйте ключові слова мови програмування:
   - Не використовуйте ключові слова мови програмування як назви змінних.

5. Використовуйте англійську мову

Приклади роботи аналізатора:

Приклад роботи аналізатора коду в стилі snake_case

Було:

def calc_total_cost(item_names, item_quantities, item_prices): // Calc total cost of items

Стало:

definition calculate total cost (item names, item quantities, item prices): // Calc total cost of items

2. ТЕОРЕТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМУ

2.1 Огляд алгоритму аналізатора

Перша версія алгоритму буде підтримувати всього декілька мов програмування, але архітектура застосунку буде підтримувати потенційне розширення функціоналу для аналізу більшої кількості мов.

Отже однією з цілей моєї роботи є реалізація даного алгоритму аналізатора:

1. Завантаження вхідного тексту програми(коду)
2. Визначення мови програмування
3. Заміна логічних та математичних операторів словами з NL
4. Розшифрування всіх самодокументованих назв сутностей в коді: змінних, методів, функцій, класів тощо.

Альтернативою алгоритму є розробка та навчання нейромережі для аналізу коду програми. Але незважаючи на це, код програми повинен відповідати тим самим вимогам які висунуті для роботи алгоритму.

Мовою програмування для розробки було обрано Python через наявність великою кількості бібліотек, що спростять нам розробку

2.2 Дослідження рішень для визначення мови програмування

Для реалізації другого пункту нашого алгоритму буде використано бібліотеку Pygments через наявність в ній функції визначення мови програмування лише за її кодом.

Pygments - це бібліотека інструментів для підсвічування синтаксису вихідного коду у текстових файлах для покращення читабельності інформації. Вона написана мовою програмування Python і надає можливість виділяти ключові слова, коментарі, рядки, функції і т. д. в різних мовах програмування.

Основні особливості Pygments:

- Підтримка багатьох мов програмування: Pygments підтримує велику кількість мов програмування, таких як Python, JavaScript, Java, C++, HTML, CSS, Ruby, PHP і багато інших.
- Підтримка різних форматів вихідного коду: Pygments може генерувати підсвічений синтаксис у різних форматах, таких як HTML, RTF, LaTeX, ANSI escape sequences та інші.
- Розширюваність: Ви можете легко розширити Pygments для підтримки нових мов програмування або форматів вихідного коду, визначивши власні лексери і стилі підсвічування.
- Підтримка інших мовних розміток: Поза підсвічуванням синтаксису вихідного коду, Pygments також підтримує підсвічування синтаксису для інших мовних розміток, таких як JSON, XML, YAML тощо.

2.3 План розробки алгоритму аналізатору даних мовою Python

Алгоритм роботи:

1. Language (Аналізатор мови) визначає мову програмування та чи підтримує наш алгоритм її
2. Spliter (Сплітер) розділяє код по рядках та редагує назви змінних (якщо декілька слів, то роздяліє їх)
3. Spliter (Сплітер) відправляє слово у клас Autocomplete
4. Autocomplete (Автозаповнювач) доповнює всі скорочення і повертає розшифрований рядок у Spliter(Сплітер)
5. Spliter (Сплітер) поєднує всі рядки та повертає розшифрований код
6. Клас Vocabulary (Словник) відповідає за заповнення словника новими словами.

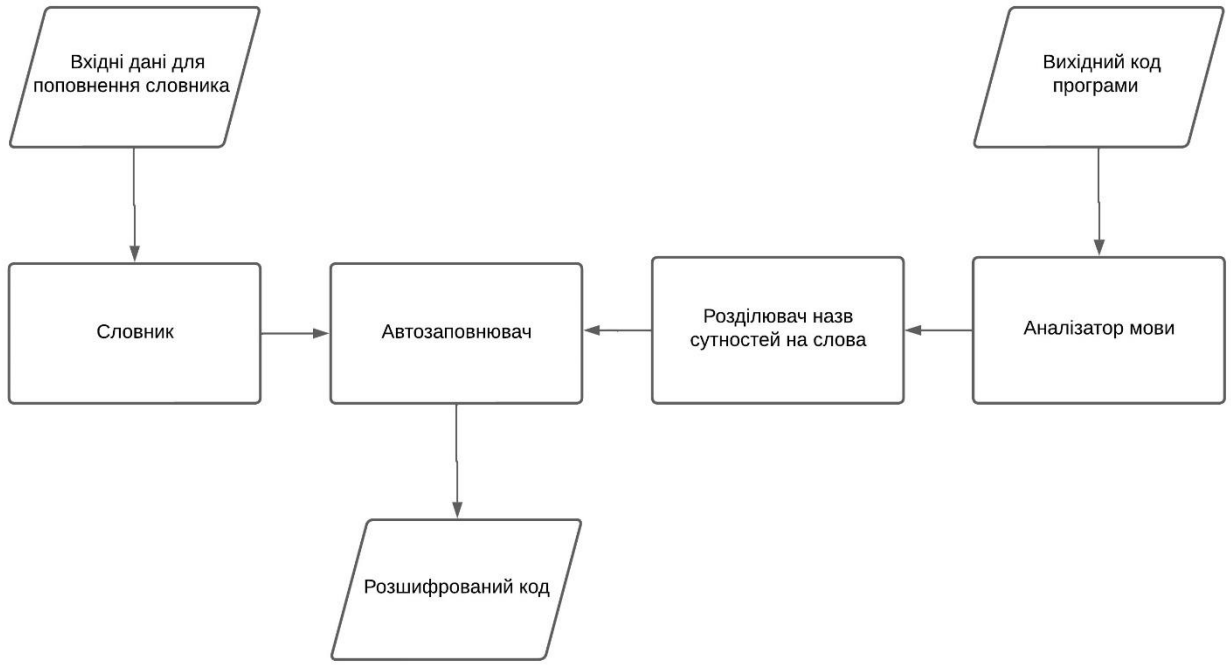


Рисунок 1.1 – Блок-схема алгоритму

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Програмна реалізація алгоритму

Програмна реалізація передбачає написання скрипту, в основі якого лежить блок-схема алгоритму (Рис. 1.1). Алгоритм займає 96 рядків (включно з пустими рядками між методами) та реалізований у вигляді сценарію в одному файлі. Для коректної роботи коду потрібно додатково встановити та імпортувати бібліотеки: redis для роботи з БД, re для роботи з регулярними виразами та pygments для визначення мови програмування. Код всієї програми знаходиться у Додатку 1 дипломної роботи, а приклад роботи – у Додатку 2.

3.2 Огляд бази даних(БД) для алгоритму

Для функціонування алгоритму достатньо зберігати тільки слова та частоту їх використання, тому базу даних можна реалізувати у вигляді простої таблиці в Redis. Для збільшення продуктивності таблицю було розбито на секції, за які відповідають ключі. Назви ключів являють собою букви англійського алфавіту, під кожним ключем зберігаються слова, які починаються на відповідну букву алфавіту.

Архітектура БД зі словами(Словника)

- Keys: ключ, за яким програма знаходить секцію з відповідним словом
    - Key(String): саме слово, що записано в словнику
    - Value(Int): кількість використання слова

3.3 Реалізація алгоритму з застосуванням технологій ШІ

Програмна реалізація процесу навчання моделі для автозаповненя тексту за першими літерами

Навчання мережі відбувалося за допомогою бібліотеки TensorFlow, яка призначена для машинного навчання. Код використаний для навчання знаходиться у Додатку 3 дипломної роботи, а код для перевірки роботи у Додатку 4.

Принцип роботи коду з Додатку 3:

1. Імпорт бібліотек:
    - Відбувається імпорт необхідних бібліотек і класів: numpy для роботи з масивами, Tokenizer для обробки тексту, pad_sequences для доповнення послідовностей, і класи Sequential, Embedding, LSTM, Dense, Bidirectional для побудови та навчання нейронної мережі з використанням бідирекційного LSTM.

2. Підготовка даних:
    - corpus: список кортежів, де кожний кортеж містить два рядки: функцію та її перетворення.
    - tokenizer: ініціалізується об'єкт Tokenizer та навчається на тексті corpus.
    - totalWords: обчислюється загальна кількість унікальних слів після токенізації тексту.
    - inputSequences та labels: створюються порожні списки, які будуть містити вхідні послідовності та відповідні мітки.

3. Формування послідовностей та міток:
    - Для кожного кортежу (function, transformation) в corpus:
    - Функція токенізується за допомогою texts_to_sequences.

- Формується кілька грамових послідовностей (nGramSequence) з префіксом зі збільшенням довжини.
- Кожна грамова послідовність і її відповідна мітка додаються до відповідних списків inputSequences та labels.

4. Побудова моделі:
    - Створюється об'єкт Sequential для побудови моделі.
    - Додається вбудований шар (Embedding), бідирекційний LSTM шар, LSTM шар та повнозв'язаний шар з активацією softmax.
    - Модель компілюється з використанням функції втрат sparse_categorical_crossentropy та оптимізатором adam.

5. Навчання моделі:
    - Модель навчається на inputSequences та labels протягом 100 епох.

6. Збереження моделі:
    - Навчена модель зберігається у файл з назвою 'functionTransformationModel.h5' для подальшого використання.

4. Емпіричне дослідження

4.1 Вибір методів дослідження

Для досягнення об'єктивних результатів у виборі найкращого підходу до розв'язання конкретної задачі ми вирішили використати емпіричний метод дослідження.

Параметри дослідження

1. Швидкість роботи:
    - Для оцінки швидкості роботи кожного методу будемо вимірювати час, необхідний для виконання конкретних завдань або обробки великих обсягів даних.
2. Точність:
    - Оцінюватимемо точність роботи кожного методу на основі порівняння результатів звичайних алгоритмів та штучного інтелекту з відомими правильними відповідями або еталонними даними.
3. Ресурсоємність:
    - Вивчатимемо ресурсоємність кожного методу, включаючи використання пам'яті, обчислювальну потужність та інші обмеження, які можуть впливати на ефективність застосування.

Проведення експериментів

Для збору даних та проведення емпіричного дослідження ми виконаємо наступні кроки:

1. Вибір тестових наборів даних:
    - Визначимо набори даних, які найкраще відображають характеристики та складність задач, що вивчаються. Важливо обрати репрезентативні дані, які дозволять адекватно порівняти роботу алгоритмів та штучного інтелекту.
2. Виконання експериментів:

- Виконаємо експерименти з використанням обох методів на визначених завданнях та зберемо відповідні дані.

3. Аналіз результатів:
    - Проведемо аналіз отриманих даних та порівняємо роботу звичайних алгоритмів та штучного інтелекту за обраними параметрами.
4. Формулювання висновків:
    - На основі аналізу результатів сформулюємо висновки щодо ефективності кожного методу та його придатності для вирішення визначеної задачі.

4.2 Збір та аналіз даних

Для початку нам треба зібрати тестові дані. Перевірку роботи алгоритму було вирішено випробувати з використанням коду на 3 різних мовах, а саме Phyton, JavaScript та C#. Через те, що поставлена у роботі задача передбачає просто перетворення коду і її реалізація передбачає оброку вхідного тексту програми по рядкам, то для перевірки було прийняте рішення використати тексти розміром по 100 рядків. Обидва рішення приймають код і не перевіряють його працездатність, що дає нам змогу використовувати не весь текст для її перевірки. Отже для дослідження роботи рішень з різним об’ємом коду нам не потрібно аналізувати різні тексти, а просто використовувати той самий текст, але різні по довжині його частини.

Аналіз швидкість роботи алгоритму та рішення на основі ШІ

Для аналізу швидкості роботи було здійснено виміри часу роботи програми для тексту об’ємом від 1 до 100 рядків. Враховуючи особливості роботи компілятора та навантаження системи час виконання коду з однаковими вхідними даними може відрізнятися, тому було вирахувано похибку показника часу роботи програми $\varepsilon=0{,}0174$ с.

За результатами вимірів швидкості роботи з кодами різних мов програмування було побудовано відповідні графіки часу роботи з вхідним текстом мовою C#, мовою JavaScript та мовою Python.

Рисунок 4.1 – Графік залежності часу виконання рішення на основі алгоритму з вхідним текстом мовою C#

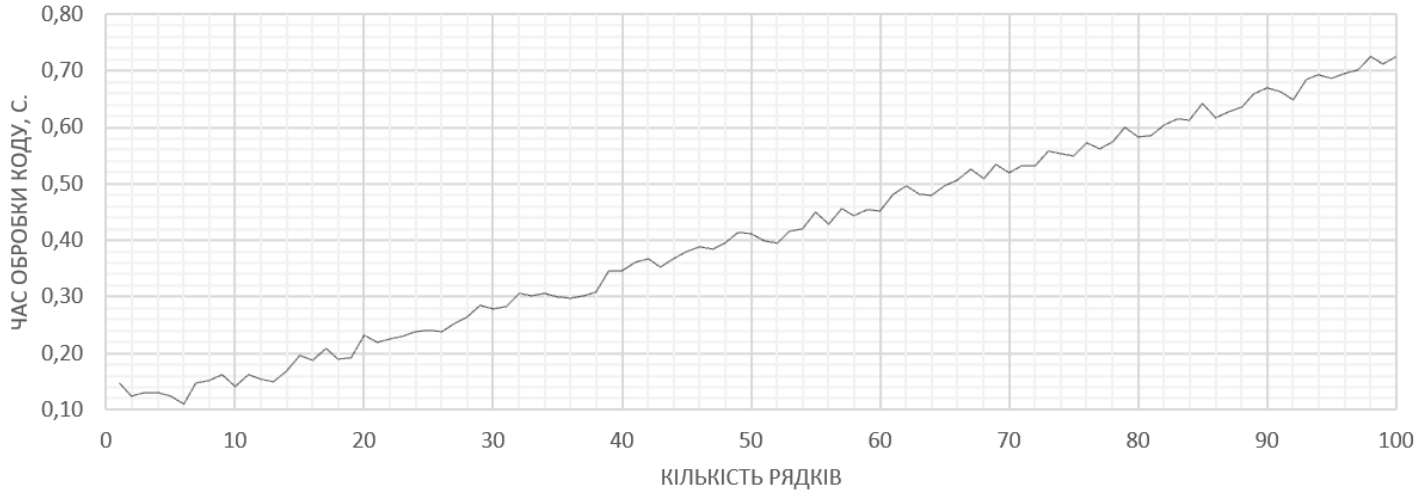


Рисунок 4.2 – Графік залежності часу виконання рішення на основі алгоритму з вхідним текстом мовою JavaScript

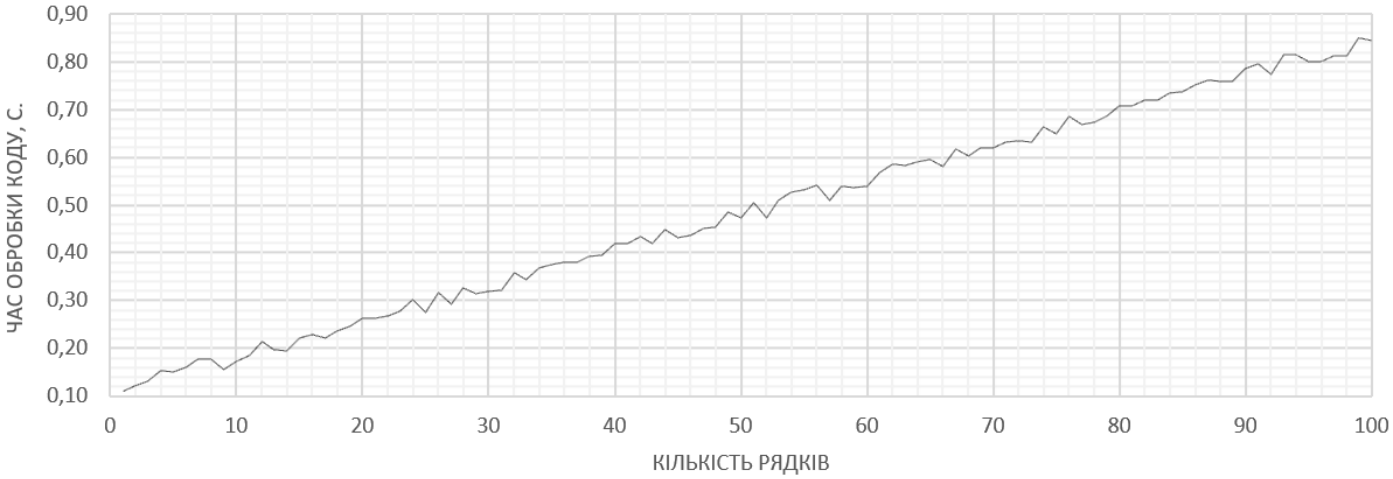


Рисунок 4.3 – Графік залежності часу виконання рішення на основі алгоритму з вхідним текстом мовою Python

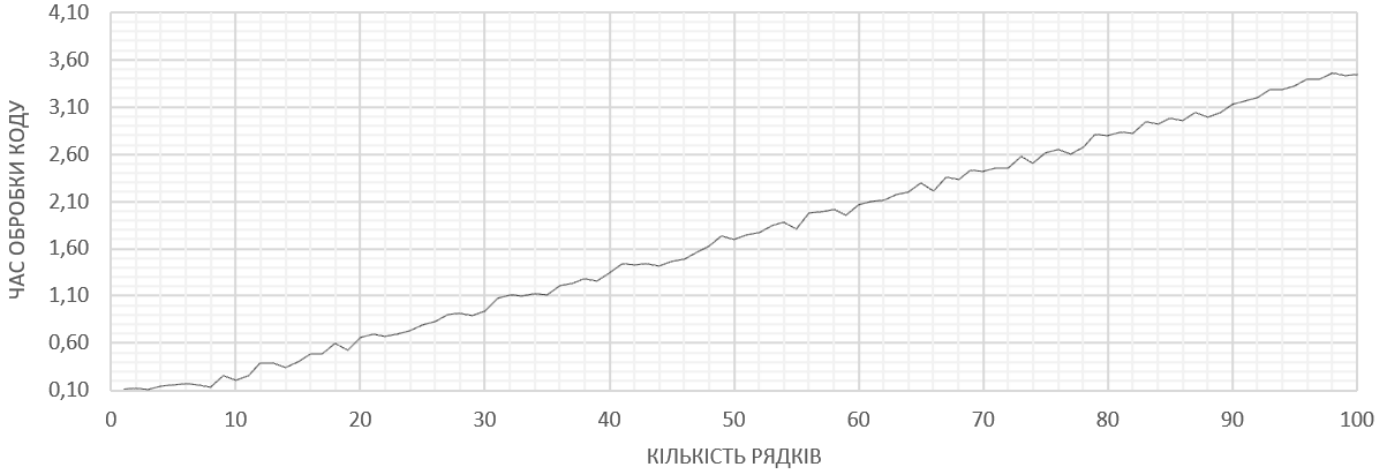


Рисунок 4.4 – Графік залежності часу виконання рішення на основі штучного інтелекту з вхідним текстом мовою C#

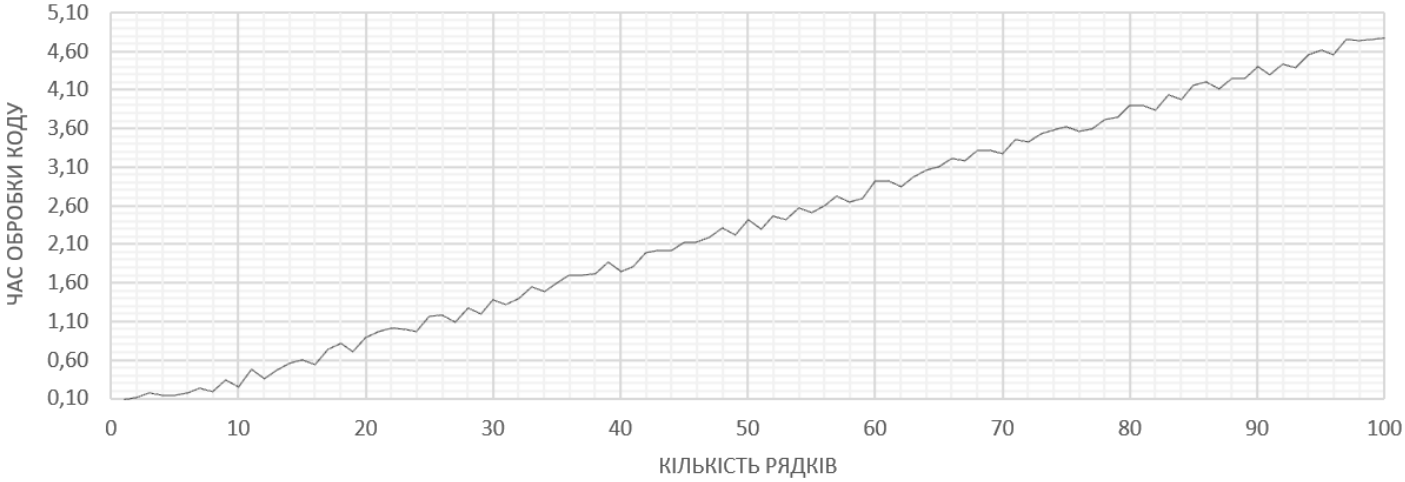


Рисунок 4.5 – Графік залежності часу виконання рішення на основі штучного інтелекту з вхідним текстом мовою JavaScript

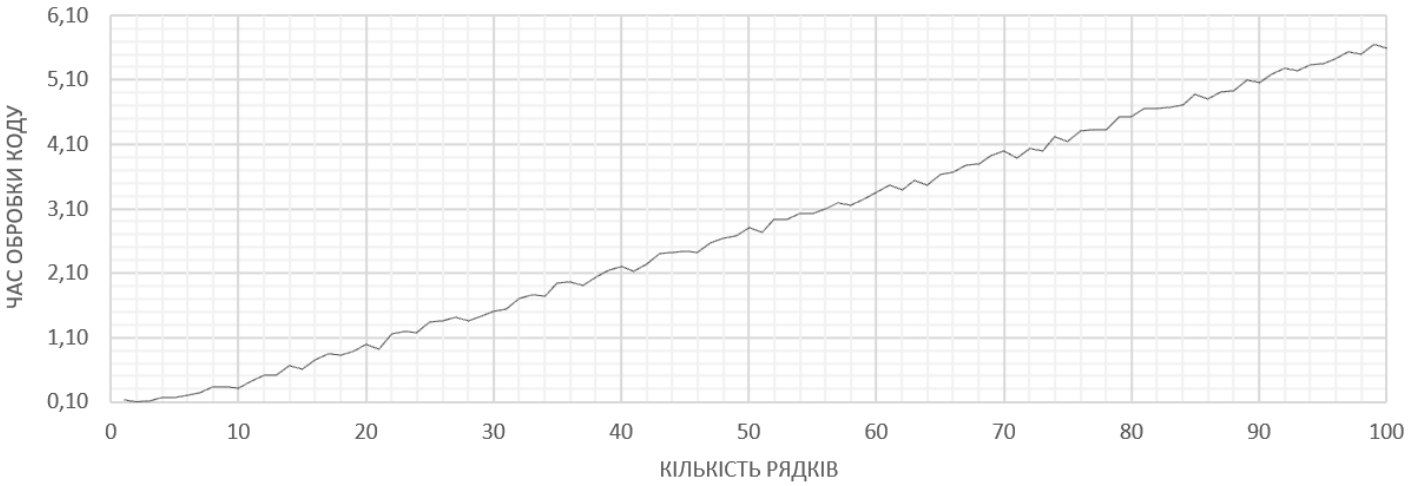


Рисунок 4.6 – Графік залежності часу виконання рішення на основі штучного інтелекту з вхідним текстом мовою Python

Проаналізувавши отримані дані ми можемо побачити, що рішення на основі ШІ працюють швидше з малою кількістю коду (до 10 рядків), але більшу кількість коду ШІ обробляє повільніше, ніж алгоритм. Це зумовлено тим, що алгоритм під час роботи для кожного слова робить запит у БД для його пошуку та відповідно для конвертації його у NL. З цього ми можемо зробити висновок, що ШІ на даній стадії працює повільніше, ніж алгоритм.

Аналіз точності алгоритму та рішення на основі ШІ

Для аналізу точності роботи було виміряно кількість помилок на кожні 10 рядків текстових кодів, написаних трьома мовами програмування (C#, JavaScript, Python).

За результатами аналізів також було побудовано відповідні графіки та пораховано середню кількість помилок на 10 рядків коду.

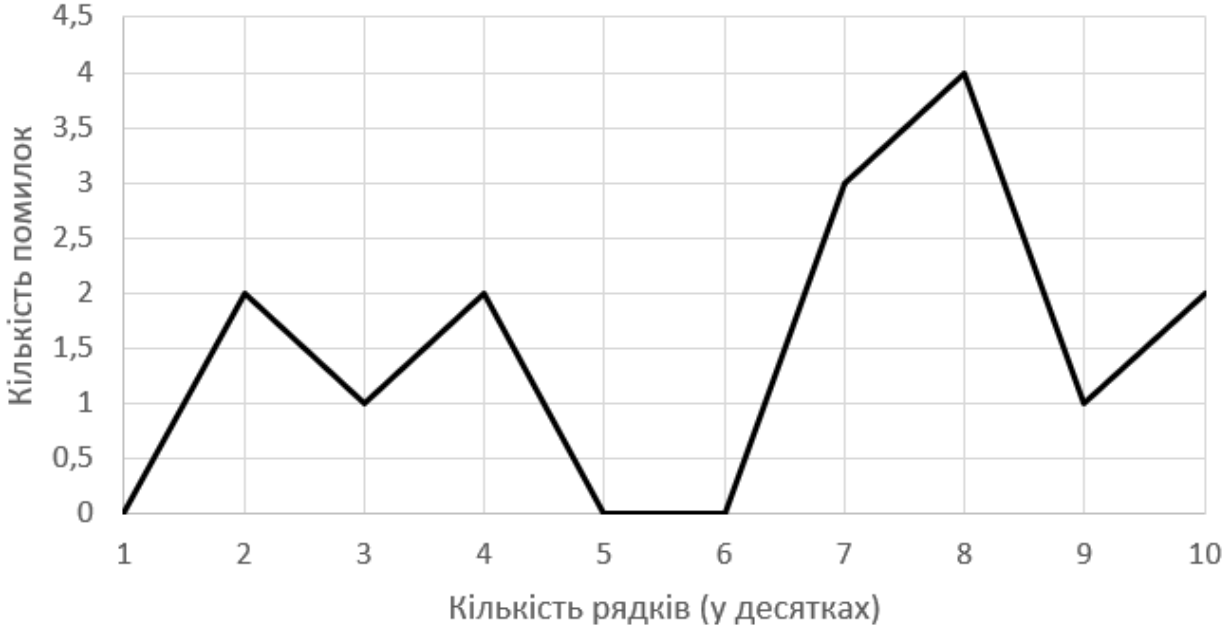


Рисунок 4.7 – Графік залежності кількості помилок алгоритму від кількості рядків з вхідним текстом мовою C#

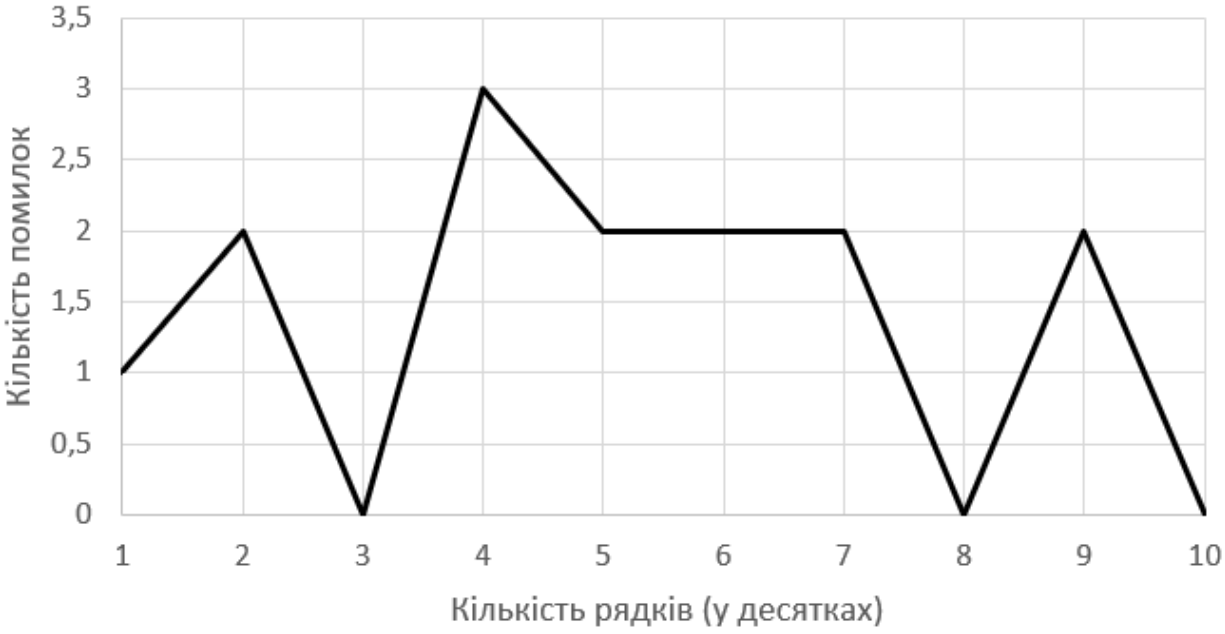


Рисунок 4.8 – Графік залежності кількості помилок алгоритму від кількості рядків з вхідним текстом мовою JavaScript

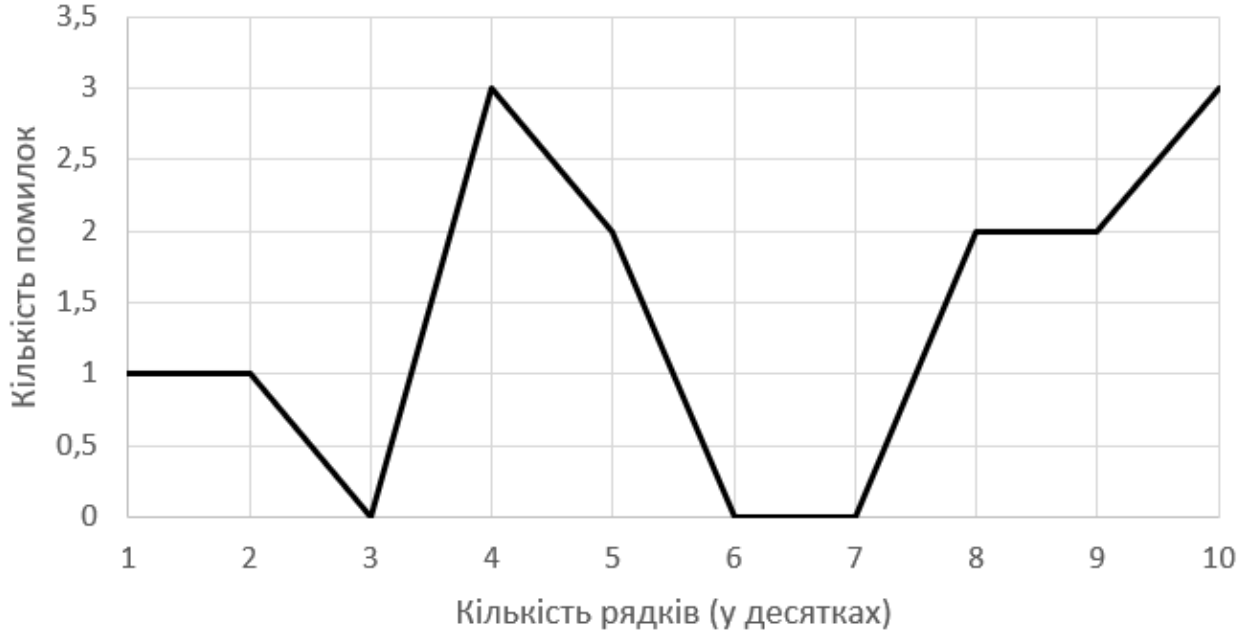


Рисунок 4.9 – Графік залежності кількості помилок алгоритму від кількості рядків з вхідним текстом мовою Python

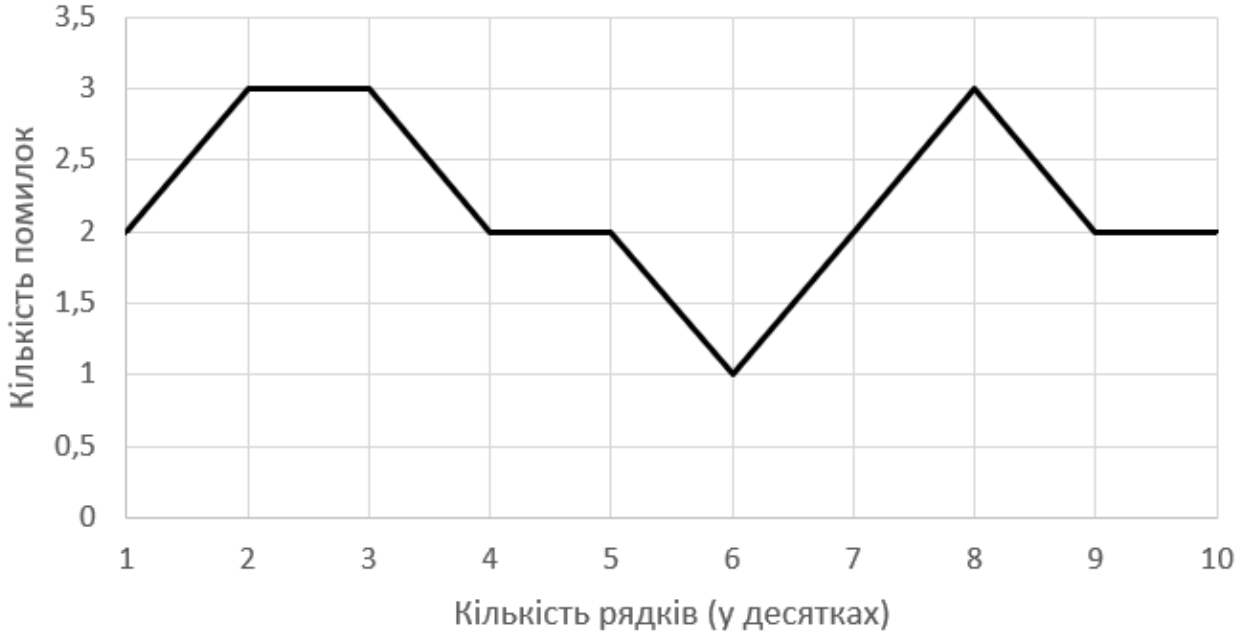


Рисунок 4.10 – Графік залежності кількості помилок рішення на основі ШІ від кількості рядків з вхідним текстом мовою C#

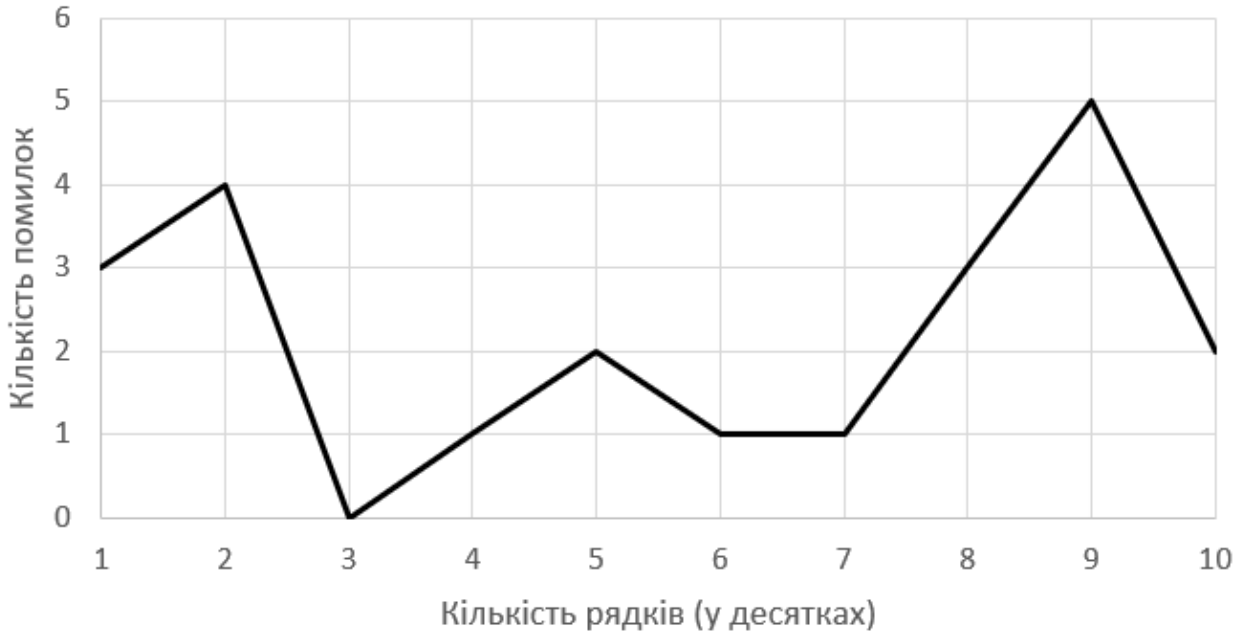


Рисунок 4.11 – Графік залежності кількості помилок рішення на основі ШІ від кількості рядків з вхідним текстом мовою JavaScript

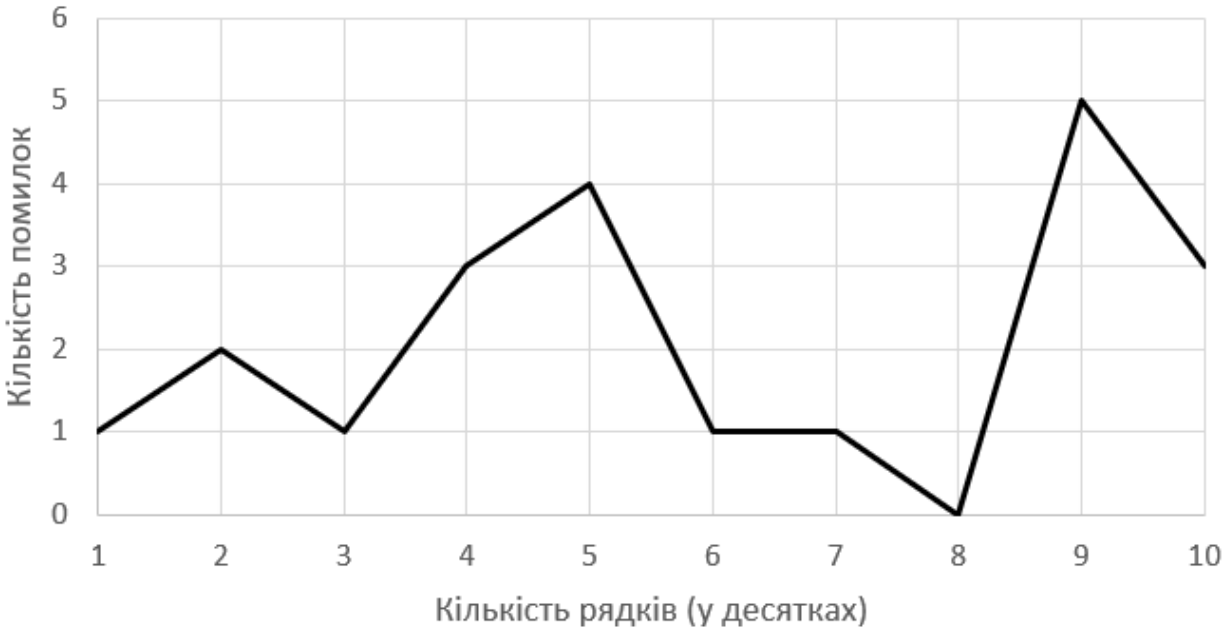


Рисунок 4.12 – Графік залежності кількості помилок рішення на основі ШІ від кількості рядків з вхідним текстом мовою Python

Для алгоритму середня кількість помилок становить 1.43 помилки на 10 рядків, а для рішення на основі ШІ середня кількість помилок становить 2.16 помилок на 10 рядків коду. Помилки в ШІ зумовлені недостатнім рівнем його розвитку, особливо при роботі з небуквеними символами. Помилки у роботі алгоритму зумовлені недостатньою кількості даних у БД, тобто недостатньою кількістю слів в словнику для покриття потреб у розшифровці даних.

Аналіз навантаження на систему алгоритму та рішення на основі ШІ

Для порівняння було обрано лише перевірка навантаження на процесор. Це зумовлено тим, що алгоритм використовує Redis, як середовище для зберігання даних, яке на пряму взаємодіє з ОЗУ. Максимальне навантаження середовища Redis регулюється на ОЗУ регулюється та залежить від кількості запитів та повернених даних, отже порівнювати навантаження на оперативну пам’ять алгоритму та рішенні на основі ШІ не є доцільним, бо у першому випадку воно у великій мірі залежить від наявності даних та від налаштувань користувача.

Перевірка навантаження проводилась на процесорі Intel Core i5-10300H з базовою частотою 2.5 GHz. Рішення на основі алгоритму показало середній відсоток навантаження 16%, а рішення на основі роботи ШІ – 21%.

ВИСНОВОК

У даній роботі мною було розроблено аналізатор вихідного тексту прогам. Дана робота дає змогу використовувати його для дешифрування звичайного коду у Natural Language(NL) для подальшого його використання наприклад у взаємодії з ШІ, який знається на NL. Аналізатор має реалізацію у вигляді алгоритму та у вигляді ШІ. У цій роботі було досліджено роботу цих двох рішень та проведено їх порівняння у трьох головних критеріях: швидкість роботи, точність та навантаження на систему. Алгоритм показав себе з кращої сторони, але слід також зважати на рівень розвитку даного ШІ.

Перелік посилань

1. Russell S., Norvig P. Artificial Intelligence: A Modern Approach / S. Russell, P. Norvig // Prentice Hall. – 2020.
2. Brynjolfsson E., McAfee A. The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies / E. Brynjolfsson, A. McAfee // W.W. Norton & Company. – 2014.
3. Goodfellow I., Bengio Y., Courville A. Deep Learning / I. Goodfellow, Y. Bengio, A. Courville // MIT Press. – 2016.
4. Lutz M. Learning Python / M. Lutz // O'Reilly Media. – 2013.
5. Beazley D., Jones B. K. Python Cookbook / D. Beazley, B. K. Jones // O'Reilly Media. – 2013.
6. VanderPlas J. Python Data Science Handbook: Essential Tools for Working with Data / J. VanderPlas // O'Reilly Media. – 2016.

ДОДАТКИ

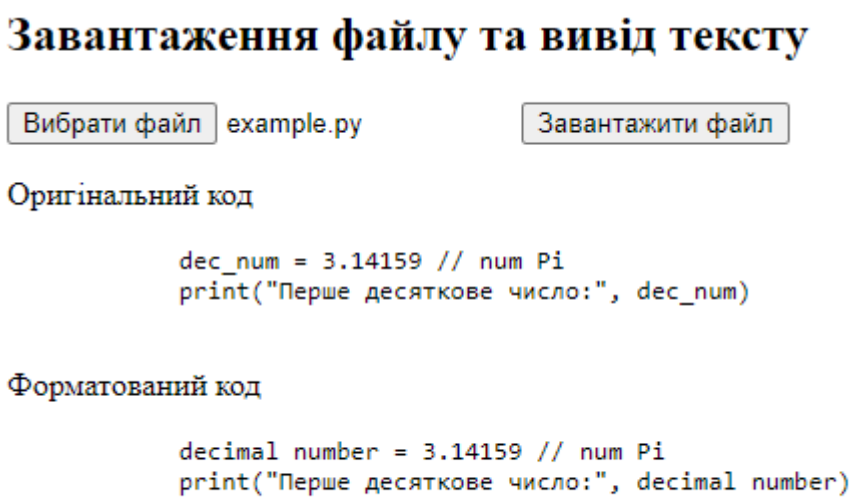
Додаток 1 (лістинг коду алгоритму мовою Python)

```
import redis
import re
from pygments import highlight
from pygments.lexers import get_lexer_for_filename
from pygments.formatters import HtmlFormatter

class Autocompleter:
    def addWordCounter(word):
        wordKey = word[0]
        if word.isalpha():
            r = redis.StrictRedis(host='localhost', port=6379, db=0)
            if (r.hget(wordKey, word)):
                r.hincrby(wordKey, word, 1)

    def addWordToRedis(word):
        word = word.lower()
        wordKey=word[0]
        if word.isalpha():
            r = redis.StrictRedis(host='localhost', port=6379, db=0)
            r.hset(wordKey, word, 0)

    def checkWordInRedis(word):
        r = redis.StrictRedis(host='localhost', port=6379, db=0)
        wordKey = word[0]
        vocWords = r.hkeys(wordKey)
        matchedWords=[]
        for vocWord in vocWords:
            if vocWord.decode().startswith(word):
                currentCounter = r.hget(wordKey, vocWord)
                matchedWords.append([vocWord, currentCounter])
        word = max(matchedWords, key=lambda x: x[1])[0].decode()
        return word

    def convertToEnglish(words):
        newWords = []
        for word in words:
            if word.isalpha():
                newWord = word.lower()
                redisWord=Autocompleter.checkWordInRedis(newWord)

                if(redisWord):
                    Autocompleter.addWordCounter(redisWord)
                    newWords.append(redisWord)
```

```
            else:
                newWords.append(word)
        result = ''.join(newWords)
        return result

class Spliter:
    def splitWordsCamelCase(input_string):
        pattern = re.compile(r'([a-zA-Z0-9(),[\]:\+\-\*/=|&%\\/.?,${}]+|[\s]+|[A-Z(),][a-z(),]*)')
        words = pattern.findall(input_string)
        return words
    def splitWordsSnakeCase(input_string):
        pattern = re.compile(r'([a-z0-9(),[\]:\+\-\*/=|&%\\/.?,${}]+|[\s]+|[A-Z(),][a-z(),]*)')
        words = pattern.findall(input_string)
        return words
    def checkCase(input_string):
        if "_" in input_string:
            return Spliter.splitWordsSnakeCase(input_string)
        elif any(x.isupper() for x in input_string):
            return Spliter.splitWordsCamelCase(input_string)
        else:
            return Spliter.splitWordsSnakeCase(input_string)

class Lexer:
    def openFile(fileName):
        with open(fileName, 'r') as file:
            code = file.read()
        return code
    def getLexer(fileName):
        lexer = get_lexer_for_filename(filename)
        return lexer

filename = './main_test.py'

lexer = Lexer.getLexer(filename)
code = Lexer.openFile(filename)
newCode = ''
splitedCode = code.splitlines()

for line in splitedCode:
    inputText = re.sub(r"([()\[\]:,.])", r" \1 ", line)
    words = Spliter.checkCase(inputText)
    outputText = Autocompleter.convertToEnglish(words)
    newCode+=outputText+"\n"

highlighted_code = highlight(newCode, lexer, HtmlFormatter())

outputFilename = './highlighted_code.html'
```

```
with open(outputFilename, 'w') as outputFile:
    outputFile.write(highlighted_code)

print(f"Виділений код збережено в {outputFilename}")
```

Додаток 2 (приклад роботи алгоритму)



Додаток 3 (код використаний для навчання ШІ)

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Bidirectional

corpus = [
    ("def detectLanguage(code):", "definition detect language ( code ):"),
    ("def sum(a, b):", "definition sum ( a , b ):"),
]

tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
totalWords = len(tokenizer.word_index) + 1

inputSequences = []
labels = []
for function, transformation in corpus:
    tokenList = tokenizer.texts_to_sequences([function])[0]
    for i in range(1, len(tokenList)):
        nGramSequence = tokenList[:i+1]
```

```
        inputSequences.append(nGramSequence)
        transformation_lower = transformation.lower()
        labels.append(tokenizer.texts_to_sequences([transformation_lower])[0][-1])

maxSequenceLen = max([len(x) for x in inputSequences])
inputSequences = np.array(pad_sequences(inputSequences, maxlen=maxSequenceLen, padding='pre'))
labels = np.array(labels)

model = Sequential()
model.add(Embedding(totalWords, 100, input_length=maxSequenceLen-1))
model.add(Bidirectional(LSTM(150, return_sequences=True)))
model.add(LSTM(100))
model.add(Dense(totalWords, activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(inputSequences, labels, epochs=100, verbose=1)

model.save('functionTransformationModel.h5')
```

Додаток 4 (код використаний для тестування ШІ)

```
import re
from sre_parse import Tokenizer
from pygments.lexers import get_lexer_for_filename
from tensorflow.keras.models import load_model

model = load_model('functionTransformationModel.h5')

def transformText(text, tokenizer, maxSequenceLen):
    for _ in range(maxSequenceLen - 1):
        tokenList = tokenizer.texts_to_sequences([text])[0]
        tokenList = pad_sequences([tokenList], maxlen=maxSequenceLen - 1, padding='pre')
        predicted = model.predict_classes(tokenList, verbose=0)
        outputWord = ""
        for word, index in tokenizer.word_index.items():
            if index == predicted:
                outputWord = word
                break
        text += " " + outputWord
        if outputWord == '<eos>':
            break
    return text

inputText = "def test_function(x, y):"
```

```
class Lexer:
   def openFile(fileName):
      with open(fileName, 'r') as file:
         code = file.read()
      return code
   def getLexer(fileName):
      lexer = get_lexer_for_filename(fileName)
      return lexer

filename = './lang.py'

lexer = Lexer.getLexer(filename)
code = Lexer.openFile(filename)
newCode = ''
tokenizer = Tokenizer()

splitedCode = code.splitlines()

for line in splitedCode:
   inputText = re.sub(r"([()\[\]:,.])", r" \1 ", line)
   outputText = transformedText = transformText(inputText.lower(), tokenizer, maxSequenceLen)
   newCode+=outputText+"\n"

print("Transformed Text:", transformedText)
```