ОТЧЁТ

о выполнении практической работы по теме:

«Анализ возможностей по обращению мутации слов»

Выполнил:

Лепёшкин К.С._____

(подпись)

Москва

2020 г.

Оглавление

Введение

Основная цель данной работы заключается в определении возможностей по обращению мутаций слов с последующим написанием программного средства, способного обращать данные мутации.

В чём же состоит проблема и насколько она актуальна? Почти каждый продвинутый пользователь осведомлен о существовании программного обеспечения, способного мутировать слова и текст для определенных целей. Одной из основных целей является вполне обычное желание каждого пользователя защитить собственную информацию. Как раз для таких целей создано множество утилит, как например Hashcat, способный мутировать слова по любому шаблону, а также открытое программное обеспечение, созданное пользователями Сети и располагающееся в открытом доступе на многих git-ресурсах, к примеру, Github.

Но если слово можно зашифровать, то вполне вероятно, его можно и расшифровать, то есть получить определенный набор слов, к которому мы можем применить те же самые методы мутации и получить наше зашифрованное слово. Основная проблема в том, что проверить данное утверждение в нашем случае затруднительно, так как доступного рядовому пользователю программного обеспечения найти не представляется возможным. Что это значит? Единственная технология, которая удовлетворила бы наши задачи, имеется у коммерческих организаций, как например, ElcomSoft, создающие приложения, способные восстанавливать пароли, просматривать скрытую информацию операционной системы и выполнять множество других задач, связанные с обеспечением безопасности данных. Но, как можно понять, данное программное обеспечение является коммерческим.

Имеется ли возможность найти открытую технологию, которая удовлетворила бы нашим требованиям? С большой вероятностью, ответ отрицательный.

Таким образом, цель данной работы заключается в написании программного обеспечения, способного получить определенный набор слов, которые являются базовыми словами, то есть теми, которые были мутированы. Или же доказать, что получить такой набор слов не представляется возможным.

1 Задачи проекта

Мы определили главную проблему в том, что доступного программного обеспечения, удовлетворяющее нашим требованиям, для выполнения цели не существует.

Таким образом, чтобы реализовать данное приложение, способное осуществить поставленную цель, следует рассмотреть ряд задач, которые будут выполнены в ходе данной практической работы.

Всего мы можем выделить 6 основных задач:

1) проанализировать существующие методы выделения лексем из текста без пробелов, чтобы иметь возможность корректно выделять базовые части слова и исправлять ошибки по мере необходимости;
2) реализовать и протестировать методы выделения лексем с удалением некорректных символов, чтобы иметь возможность очищать слова от некорректных символов;
3) проанализировать существующие методы выделения базовых частей из слова, полученного путем объединения несвязных слов, чтобы иметь возможность составить минимальный набор слов, удовлетворяющий нашей цели;
4) реализовать и протестировать метод выделения базовых частей из слова, чтобы составить набор базовых слов;
5) измерить время работы реализованных методов на большой выборке данных, чтобы получить представление о производительности программы и иметь возможность увеличения данной производительности;
6) реализация и тестирование программного средства сравнения совпадений в заданных файлах, чтобы иметь возможность сопоставить полученный набор слов с исходными задуманными словами.

В ходе выполнения поставленных задач мы сможем понять насколько поставленные цели осуществимы.

2 Используемые инструменты

2.1 Выбор языка программирования

Для реализации задач, поставленных в предыдущем пункте, мной были проанализированы существующие программные средства, являющиеся наиболее целесообразными для конкретной задачи. Целесообразность программного средства заключается в простоте его использования, чётком выполнении задачи, которую мы ставим перед этой программой, а также в актуальности на данный момент времени.

Все выбранные инструменты будут использоваться для языка программирования Python 3.7, поскольку данный язык обеспечивает высокую скорость разработки, совместимость со многими высокоуровневыми языками, что позволяет реализовывать программы на более производительном языке и внедрять их в свои модули, а также множество встроенных инструментов для огромного спектра задач.

Выбранные программные средства можно разделить на 4 группы в соответствие с выполнением определённой задачи.

2.2 Выбор алгоритма для выделения лексем из текста без пробелов

Данный алгоритм был найден на сервисе GitHub и является открытым программным средством. Поскольку он не удовлетворял задаче выделения лексем с достаточным быстродействием, он был изменён и приведен к окончательному виду.

Данный алгоритм реализован с применением метода динамического программирования и использует так называемую «стоимость» слова, то есть математически вычисляемую величину, прямо пропорциональную рангу слова в списке слов, упорядоченных по частоте употребления.

2.3 Выбор алгоритмов для исправления некорректных слов

Для начала уточним, что подразумевается под некорректным словом. Слово является некорректным, если при написании этого слова используются цифры, служебные символы, пробельные символы и тому подобное (например, 123password), если в написанном слове содержится орфографическая ошибка (например, pasword,

password) А также «Leet» символы, то есть символы, полученные заменой латинских букв на похожие цифры и символы (например, p4s5w0rd).

Поскольку, невозможно найти готовый алгоритм, который способен был бы решить решения проблемы с «Leet» символами, а также другими некорректными символами, мной был написан собственный алгоритм, реализованный на основе подсчёта «стоимости» слова.

Для решения задачи по исправлению орфографических ошибок я воспользовался известным алгоритмом БК-дерева. Данный алгоритм выстраивает дерево, основываясь на определенном расстоянии между узлами. Узлами данного дерева выступают слова из частотного списка. А в качестве расстояния между узлами мной было выбрано расстояние Дамерау-Левенштейна, поскольку оно идеально подходит для задач нахождения близких слов (то есть слов, которые минимально отличаются друг от друг посимвольно). Расстояние Дамерау-Левенштейна подсчитывает расстояние, учитывая не только взаимное расположение букв в словах, но и их перестановки (это улучшенное расстояние Левенштейна).

Для реализации БК-дерева мной был создан собственный класс, являющийся обёрткой для уже реализованного алгоритма дерева, находящегося в открытом доступе на ресурсе GitHub. Пакет, который я выбрал называется «pybktree». Поскольку этот пакет является актуальным и поддерживается разработчиком на данный момент, а также предоставляет понятный интерфейс для использования, было принято решение использовать данный вариант. Сам алгоритм мной изменён не был.

Расстояние Дамерау-Левенштейна реализовано также сторонним разработчиком и является открытым программным обеспечением. Пакет, который мной будет использоваться, называется «pyxdameraulevenshtein». Мой выбор пал на данный пакет потому, что он является лучшей реализацией данного алгоритма на данный момент. Алгоритм реализован с применением языка программирования Cython, упрощающий написание модулей C/C++ кода для Python. Код Cython преобразуется в C/C++ код для последующей компиляции и впоследствии может использоваться как расширение стандартного Python, что делает данный модуль производительнее любых других модулей, реализующих данный алгоритм, написанный на чистом Python.

Практический опыт использования пакета «pyxdameraulevenshtein», доказывает, что производительность увеличивается в более, чем 10 раз в сравнении с любым другим аналогом, написанным на Python.

2.4 Выбор алгоритма для выделения базовых частей слова

Для выделения базовых частей слова мной будет применяться собственный алгоритм, реализованный на основе подсчёта «стоимости» слова.

После выделения базовых частей, я буду искать корень каждой части, поскольку нам необходимо получить минимальный словарь слов. Для того, чтобы получить корень слова, я буду применять технологию стемминга, реализованную в библиотеке «Natural Language Toolkit» (NLTK). Данная библиотека содержит пакет библиотек и программ для символьной и статистической обработки естественного языка, написанных на языке программирования Python. Поскольку она является быстрой для выполнения задачи стемминга, а также имеет простой в использовании интерфейс, мной была выбрана данная библиотека.

2.5 Выбор библиотеки для тестирования

Для тестирования всех реализованных методов мной будет использоваться встроенный в язык Python модуль «unittest». Данный модуль предоставляет абсолютно весь необходимый для тестирования функционал, что позволяет тестировать модули быстро и эффективно.

Для реализации так называемых mock-текстов, мной был использован модуль «mock», входящий в состав «unittest».

3 Разработка программного средства

Описание разработки данного программного средства будет включать в себя последовательное описание всех модулей и компонентов программы. В процессе описания работы модулей, будут соответственно выполняться задачи, описанные раннее, а также рассмотрены реализации всех алгоритмов, необходимые для достижения поставленной цели.

Основной идеей построения всего программного средство было в том, чтобы предоставить пользователю максимально возможный функционал для получения удовлетворяющего его результата. Поэтому, реализована идея выбора 4 режимов работы, способных работать одновременно: посчитать «стоимость» слова, очистить слово от некорректных символов, исправить слово и выделить базовые части слова.

Для предоставления более гибкого функционала, реализована возможность изменения 4 дополнительных параметров, которые напрямую влияют на работу методов обработки и исправления слов, что позволяет получить оптимальный результат.

Пользователю предоставляется возможность вывода результата в файл для каждого режима работы, а также в стандартный поток вывода. Пользователь может запросить более понятный для человека вывод с помощью определенной опции (опция –v). Однако, если пользователю необходимо получить только список слов, и ничего кроме, ему достаточно не указывать дополнительной опции.

3.1 Структура проекта

Структура всего проекта состоит из пакетов «analyzer», в котором содержится все основные модули, «tests», в котором содержатся тесты для модулей, а также директории «data», содержащая списки слов, и сторонние конфигурационные файлы.

Структура проекта приведена рисунке 3.1.1.

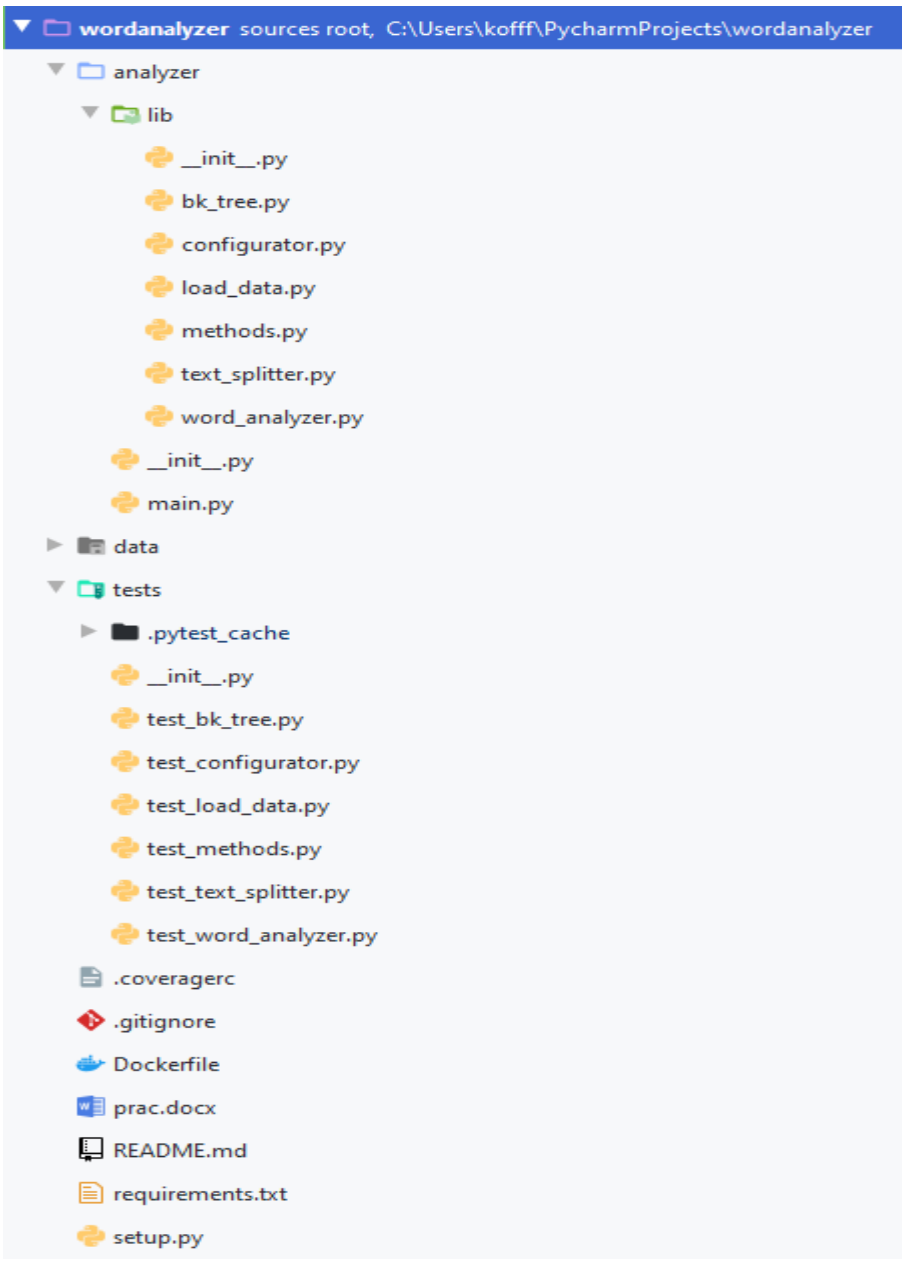


Рисунок 3.1.1 – Демонстрация структуры проекта

3.2 Основной модуль

Основным модулем программы, с которого начинается старт является модуль main.py (См. Листинг 3.2.1).

Основная идея данного модуля состоит в том, чтобы, получив аргументы от пользователя из командной строки, передать их в функцию, которая обработает аргументы и выдаст определенный результат.

Для достижения данной задачи было принято решение создать класс Configurator, который обработает все аргументы командной строки и предоставит интерфейс для получения данных.

Классом, который будет выполнять все вычисления, является класс WordAnalyzer, принимающий на вход статической функции build объект класса Configurator. Благодаря внутреннему интерфейсу, класс WordAnalyzer получит необходимые данные и запустит функцию analyze(), выполняющая вычисления.

Листинг 3.2.1

```
import sys
from analyzer.lib.configurator import Configurator
from analyzer.lib.word_analyzer import WordAnalyzer


def main():
    configurator = Configurator(sys.argv[1:])
    analyzer = WordAnalyzer.build(configurator)
    analyzer.analyze()


if __name__ == '__main__':
    main()
```

3.3 Модуль конфигуратора

Данный модуль реализует интерфейс между получением аргументов из командной строки от пользователя и передачей этих данных другому объекту.

Чтобы правильно получить и обработать аргументы командной строки используется модуль Argparse, позволяющий задать аргументы, которые необходимо получить на вход программе.

Основная идея состоит в следующем: конструктор класса Configurator получает на вход аргументы командной строки. Данные аргументы, а также параметры описания, названия и другие параметры программы передаются в статический метод создания объекта класса ArgumentParser (класс из модуля Argparse). Метод возвращает созданный объект, с помощью которого в последствие будет осуществляться обработка аргументов (См. Рисунок 3.3.1).

Объект класса ArgumentParser обрабатывается методом _get_parameters (См. Рисунок 3.3.2), возвращающий объект, содержащий все переданные пользователем параметры. Важно отметить, что в данном методе производится проверка на наличие обязательных аргументов: исходного файла со списком слов, которые необходимо обработать или список слов, переданный напрямую; имя файла, содержащий частотный словарь; хотя бы один из режимов работы программы.

Используя полученный из метода _get_parameters объект, класс предоставляет свой API для предоставления данных. Каждый метод обрабатывает данные необходимым образом и предоставляет вывод.

Для загрузки списка слов в классе присутствует метод _load_words (См. Рисунок 3.3.2), являющимся обёрткой над внешней функцией load_words из метода load_data. Данный классовый метод обрабатывает возможные ошибки и возвращает загруженный список слов. Важно отметить, что если пользователь передает список слов напрямую из командной строки (опция –w), то даже если опция получения списка слов из файла (опция –s) присутствует, она будет проигнорирована, так как первая опция является приоритетной.

```
@staticmethod
def _get_parser(program_name: str = None, description: str = None, epilog: str = None) -> ArgumentParser:
    """
    Method creates the instance of the ArgumentParser class, adds arguments in here and returns that instance.

    :param program_name: name of the program
    :param description: description of the program
    :param epilog: epilog of the program
    :return: an instance of the ArgumentParser class
    """

    parser = ArgumentParser(prog=program_name, description=description, epilog=epilog)

    parser.add_argument('-s', '--source',
                        help='The source file containing the set of words you need to analyze.',
                        type=str)

    parser.add_argument('-f', '--frequency',
                        help="The dictionary ordered by frequency of word usage, which will be used to perform the "
                             "splitting text and correcting incorrect words (if the dictionary parameter -w isn't "
                             "override).",
                        type=str)
```

Рисунок 3.3.1 – Демонстрация метода создания объекта класса ArgumentParser

```python
def _get_parameters(self, args):
    """
    This method gets all parameters from the args of the command line.

    :param args: list of the arguments of the command line
    :return: parsed arguments
    """
    parameters = self._parser.parse_args(args)

    # At least one of the parameters must be
    if not (parameters.source or parameters.words):
        self._parser.error('No action requested, add -s/--source or -w/--words')
    if not parameters.frequency:
        self._parser.error('To process the words, you must specify a file containing a list of frequency words')
    if not (parameters.correct or parameters.base_words or parameters.clear_word or parameters.total_cost):
        self._parser.error('No mode specified, add -correct/--correct or(and) -cost/--total-cost '
                           'or(and) -clr/--clear-word or(and) -basic/--base-words')

    return parameters
```

```python
@staticmethod
def _load_words(filename: str, count: int = None, encoding: str = None, prefix: str = None, postfix: str = None):
    if prefix:
        print(prefix)

    try:
        words = load_words(filename, count, encoding)
    except (TypeError, FileNotFoundError, EmptyFileError, ValueError) as e:
        print(str(e), file=sys.stderr)
        exit(-1)

    if postfix:
        print(postfix)

    return words
```

Рисунок 3.3.2 – Демонстрация методов получения параметров и загрузки слов

3.4 Модуль загрузки списка слов

Предположим, что нам необходимо получить список слов из файла. Но что, если файл чрезмерно большой, а нам необходимо получить лишь случайный срез? А если кодировка файла нестандартная и файл не может быть просто обработан? Для этих ситуаций создан модуль load_data, содержащий функцию load_words.

Пользователь может передать функции определенное количество слов, которое ему необходимо обработать (опция –c). А если пользователю необходимо указать кодировку файла, он может передать её с помощью опции –e.

В данной функции обрабатываются стандартные ошибки пользователя, связанные с неверным вводом имя файла или случая, когда указан пустой файл.

Листинг данного модуля приведен на рисунке 3.4.1.

```python
def load_words(words_filename: str, count: int = None, encoding: str = 'utf-8') -> list:
    """
    :param encoding: encoding of the file
    :param words_filename: str - filename
    :param count: int - count of words which will be processed
    :return: list(str) - list of loaded words
    """

    if not isinstance(words_filename, str):
        raise TypeError("That type isn't string!")

    if not os.path.isfile(words_filename):
        raise FileNotFoundError(f"Error: this file: \"{words_filename}\" doesn't exist!")

    if stat(words_filename).st_size == 0:
        raise EmptyFileError(f"Error: The specified file: \"{words_filename}\" is empty!")

    if count is not None and count <= 0:
        raise ValueError("Error: The number of words that you want to load can't be 0 and less")

    with codecs.open(words_filename, 'r', encoding) as f:
        data = f.read().splitlines()

    return sample(data, count) if count else data
```

Рисунок 3.4.1 – Демонстрация основной функции модуля load_data

3.5 Модуль создания БК-дерева

Данный модуль необходим для того, чтобы сохранять, создавать или загружать объект класса BKTree из модуля pybktree, рассмотренный ранее.

В модуле bk_tree реализован класс BuildBKTree (См. Рисунок 3.5.1), являющийся потомком класса BKTree. В этом классе реализованы 3 статических метода: метод для построения дерева из списка слов, используя расстояние Дамерау-Левенштейна из модуля pyxdameraulevenshtein, метод для сохранения построенного дерева в файл, используя встроенный в язык Python модуль pickle, и метод для загрузки дерева из файла.

Реализованные методы обеспечивают производительность программы путем того, что пользователю не придётся строить объект класса дерева каждый раз при запуске программы. Он может сохранить построенную структуру в файл и в последующие разы выгружать дерево из файла.

Все взаимодействия структуры BuildBKTree с файлами осуществляются самостоятельно программой с помощью опции –t.

```python
class BuildBKTree(BKTree):
    @staticmethod
    def build_tree(words):
        if not words:
            raise WordsBKTreeError("The passed list of words is empty")

        tree = BuildBKTree(distance, words)
        return tree

    @staticmethod
    def save_tree(filename, tree):
        if not filename:
            raise FileBKTreeError("The file where you're going to save the bk-tree has the incorrect name")
        if not isinstance(tree, BKTree):
            raise WrongTreeError("You're trying to save into the file not a BK-tree object")
        if os.path.isfile(filename) and os.stat(filename).st_size != 0:
            raise FileBKTreeError("The file what you specified is not empty. It's unable to write into that file")

        with open(filename, 'wb') as file:
            pickle.dump(tree, file)
        return tree

    @staticmethod
    def load_tree(filename):
        if not filename or not os.path.isfile(filename):
            raise FileBKTreeError("This file of the bk-tree structure doesn't exist")

        try:
            with open(filename, 'rb') as file:
                tree = pickle.load(file)
                if not isinstance(tree, BKTree):
                    raise WrongTreeError("You're trying to load from the file not a BK-tree object")

        except pickle.UnpicklingError:
            raise FileBKTreeError("This file of the bk-tree structure doesn't contain any bk-tree structure actually")
        except EOFError:
            raise FileBKTreeError("The bk-tree file you specified is empty and can't be loaded")

        return tree
```

Рисунок 3.5.1 – Демонстрация модуля для работы с БК-деревом

3.6 Модуль выделения лексем из текста без пробелов

Данный модуль содержит класс TextSplitter, способный выполнять разделение текста на слова, используя вычисляемую им же «стоимость» слова.

«Стоимость» слова – это математически вычисляемая величина, зависящая от ранга слова в частотном словаре, то есть его позиция в списке слов в файле.

Основная идея состоит в том, что конструктор класса получает на вход список слов, упорядоченный по частоте употребления, а затем рассчитывает стоимость каждого слова как логарифм ранга слова, умноженного на логарифм от количества всех слов (См. Рисунок 3.6.1). Таким образом, имея словарь из 125 тысяч слов, мы получим величины, находящие в диапазоне от 1 до 14.

```python
class TextSplitter:
    def __init__(self, words: list):
        """
        Builds words cost dictionary using the list of words passed to constructor
        :param words: the list of words is ordered by frequency usage
        """

        # Check that list is not empty or None
        if words == [] or words is None:
            raise ValueError("The list of words is empty or None")

        # It's necessary the each element is str type in the list
        if not all(isinstance(word, str) for word in words):
            raise TypeError("The list of words has a non string type element")

        # Build a cost dictionary, assuming Zipf's law
        self.word_cost = dict((word, log((number + 1) * log(len(words)))) for number, word in enumerate(words))
        self.max_len = max(len(x) for x in words)
```

Рисунок 3.6.1 – Демонстрация конструктора класса TextSplitter

Класс TextSplitter выполняет разделение текста на слова с помощью метода split, принимающий текст на вход. Используя метод динамического программирования, данный метод по очереди вычисляет лучшее совпадение для i-го символа в тексте, проходя по каждому символу. Для того, чтобы найти наилучшее совпадение, как раз и используется ранее вычисленные стоимости слов. Когда наилучшие совпадения для каждого символа найдены, при помощи цикла из этого текста выделяются наименьшие по стоимости слова (См. Рисунок 3.6.2).

```python
def split(self, text: str):
    """
    Split the text without spaces. Returns array of split words
    :param text: text you need to split
    :return: list of split words
    """

    # It was decided that it's necessary to deliberately refuse to check for incorrect characters in the
    # text to increase performance. Also, this check must be performed before passing the text to the
    # function as intended by the author.

    if not isinstance(text, str):
        raise TypeError("Text must be str type")

    # Find the best match for the i first characters, assuming cost has
    # been built for the i-1 first characters.
    # Returns a pair (match_cost, match_length).
    def best_match(index):
        candidates = enumerate(reversed(cost[max(0, index - self.max_len):index]))
        costs = list(((match_cost + self.word_cost.get(text[index - match_length - 1:index].lower(), 9e999),
                       match_length + 1) for match_length, match_cost in candidates))

        return min(costs)

    # Build the cost array.
    cost = [0]
    for i in range(1, len(text) + 1):
        match_cost, match_length = best_match(i)
        cost.append(match_cost)

    # Backtrack to recover the minimal-cost string.
    out = []
    i = len(text)
    while i > 0:
        match_cost, match_length = best_match(i)
        assert match_cost == cost[i]

        out.append(text[i - match_length:i])

        i -= match_length
    return list(reversed(out))
```

Рисунок 3.6.2 – Демонстрация метода выделения слова из текста

3.7 Модуль дополнительных функций

Модуль methods является одним из основных модулей программы, поскольку в нем реализованы методы, выполняющие необходимые вычисления, но которые не могут быть включены в основной класс обработки слов ввиду общности их функционала.

В данном модуле реализованы 6 важный функций, самые основные из которых будут рассмотрены далее.

3.7.1 Функция очистки слова

Функция leet_transform реализует процесс очистки слова от некорректных символов, а также leet-символов. На вход данная функция получает индексы некорректных символов в строке и саму строку.

Данная функция проходит по каждому индексу в списке и, если этот символ является leet-символом, заменяет его на корректный символ, иначе удаляет его.

Данная функция нужна для того, чтобы для любого некорректного слова можно было найти минимальную его стоимость, путем перебора индексов некорректных символов для нахождения оптимальной их комбинации. Каждая такая комбинация индексов вместе с словом посылается на вход данной функции, она преобразовывает слово и возвращает результат (См. Рисунок 3.7.1.1)

```
def leet_transform(word: str, indices: tuple) -> str:
    """
    This functions clears the passed word from digits, service symbols or any incorrect characters.
    Also it corrects "leet" characters if their indices were passed
    :param word: the word you need to correctTak
    :param indices: indices of incorrect characters that will be corrected
    :return: corrected word
    """

    # "Leet" characters that need to be replaced
    comparison = {
        '$': 's',
        '1': 'i',
        '0': 'o',
        '3': 'e',
        '@': 'a',
        '4': 'a',
        '7': 't',
        '6': 'g',
        '5': 's'
    }

    for index in indices:
        # Get corrected symbol
        symbol = comparison.get(word[index])

        if not symbol:
            continue

        # Build corrected word
        word = word[:index] + symbol + word[index + 1:]

    # Remove all incorrect symbols
    return re.sub(r'[\W\d\s_]+', r'', word)
```

Рисунок 3.7.1.1 – Демонстрация функции удаления некорректных символов

3.7.2 Функция вывода результатов

Функция print_results получает на вход паттерн вывода и аргументы для этого вывода. Суть данной функции заключается в том, чтобы вывести результаты, переданные на вход в файл или в стандартный поток вывода (См. Рисунок 3.7.2.1).

```python
def print_results(word: str, pattern: str, args: list, file, verbose: bool = False):
    """
    This function applies the passed function to the list of words, then processes these words and pr
    in the command line or/and saves to the file if necessary. All preferences are set by the passed
    """
    count_brackets = len(re.findall(r'{\d*}', pattern))
    count_args = len(args) + 1 if verbose else len(args)

    if not isinstance(file, _io.TextIOWrapper):
        raise TypeError(f"Error: The file \"{file}\" is not a file")

    word = word if word else "NONE"

    if count_args < count_brackets:
        args = args[:] + [None for i in range(count_brackets - count_args)]
    elif count_args > count_brackets:
        raise ValueError("Error: The number of the brackets is less than the number of the arguments"

    args = [arg if arg else 'NONE' for arg in args]

    if verbose:
        if len(args) == 1:
            for arg in args:
                if isinstance(arg, Iterable) and not isinstance(arg, str):
                    for arg_word in arg:
                        file.write(pattern.format(word, arg_word))
                else:
                    file.write(pattern.format(word, arg))
        else:
            file.write(pattern.format(word, *args))
    else:
        if len(args) == 1:
            for arg in args:
                if not arg:
                    break

                if isinstance(arg, Iterable) and not isinstance(arg, str):
                    for arg_word in arg:
                        file.write(pattern.format(arg_word))
```

Рисунок 3.7.2.1 – Демонстрация функции вывода информации

3.7.3 Функция построения шаблона вывода

Функция get_patterns строит шаблоны для вывода, которые в последствие буду переданы в функцию print_results.

Шаблоны сроятся на основе режимов, которые выбрал пользователь. Это реализовано для того, чтобы пользователь получил наиболее понятный для человека вывод результатов (См. Рисунок 3.7.3.1)

```python
def get_patterns(mode: dict, verbose: bool = False):
    patterns = {filename: ["word: {}"] if verbose else [] for filename in set(mode.values())}

    filename_modes = collections.defaultdict(list)
    for _mode, filename in mode.items():
        filename_modes[filename].append(_mode)

    for filename, _mode in filename_modes.items():
        if cfg.MODE_COST in _mode:
            patterns[filename].append("cost: {}" if verbose else "{}")
        if cfg.MODE_CLEAR in _mode:
            patterns[filename].append("cleared: {}" if verbose else "{}")
        if cfg.MODE_CORRECT in _mode:
            patterns[filename].append("corrected: {}" if verbose else "{}")
        if cfg.MODE_BASIC in _mode:
            patterns[filename].append("base words: {}" if verbose else "{}")

    if patterns:
        patterns = {filename: ", ".join(pattern) + '\n' for filename, pattern in patterns.items()}
    else:
        patterns = ""
    return patterns
```

Рисунок 3.7.3.1 – Демонстрация функции построения шаблонов вывода

3.8 Модуль анализа слов

Модуль word_analyzer является основным модулем программы, в котором реализованы методы для обработки слов.

Как отмечалось ранее, статический метод класса WordAnalyzer принимает объект класса Configurator для того, чтобы получить все данные, переданные пользователем через командную строку (См. Рисунок 3.8.1).

Старт обработки слов начинается с метода analyze, который обрабатывает список слов на основе режимов, выбранные пользователем. В зависимости от режимов работы, вызываются определенные методы для обработки списка слов (См. Рисунок 3.8.2)

Чтобы вывести результат пользователю используется функция print_results, описанная ранее.

```python
@staticmethod
def build(configurator: cfg.Configurator):
    analyzer = WordAnalyzer()
    analyzer.words = configurator.get_words(verbose=True)
    analyzer.frequency_words = configurator.get_frequency_words(verbose=True)
    analyzer.splitter = TextSplitter(analyzer.frequency_words)
    analyzer.tree = configurator.get_tree()
    analyzer.mode = configurator.get_mode()
    analyzer.verbose = configurator.get_verbose()

    # This flag specifies that's need add extra information with a word when analyzing words
    analyzer.verbose_file = False

    # If the word isn't in the dictionary, then its cost is default_cost
    analyzer.default_cost = 1000

    # How many similar words will be returned by get_similar_words method
    analyzer.number_similar_words = configurator.get_configuration_values()['similar_words']
    # What distance will be used to search for similar words
    analyzer.distance = configurator.get_configuration_values()['distance']
    # How many parts will be spliced in the get_correct_words method
    analyzer.threshold = configurator.get_configuration_values()['threshold']
    # How many words will be returned by get_correct_words method
    analyzer.number_of_corrected_words = configurator.get_configuration_values()['number_corrected']

    # Define dictionary with values like (number: list_of_divisors).
    # There are defined all numbers from 1 to max length of all words.
    # It's necessary to improve efficiency, because divisors won't calculated again for same number
    analyzer.divisors = dict((number, factorize(number)) for number in range(1, analyzer.splitter.max_len + 1))

    return analyzer
```

Рисунок 3.8.1 – Демонстрация статического метода создания объекта класса WordAnalyzer

```python
for word in words:
    clear = corrects = None
    arguments = {filename: [] for filename in set(mode.values())}

    if cfg.MODE_COST in mode:
        cost = self._get_total_cost(word)
        filename = mode[cfg.MODE_COST]
        arguments[filename].append(cost)

    if cfg.MODE_CLEAR in mode:
        clear = self._get_clear_word(word)
        filename = mode[cfg.MODE_CLEAR]
        arguments[filename].append(clear)

    if cfg.MODE_CORRECT in mode:
        if clear:
            corrects = self._get_correct_words(word, clear_word=clear)
        else:
            corrects = self._get_correct_words(word)

        filename = mode[cfg.MODE_CORRECT]
        arguments[filename].append(corrects)

    if cfg.MODE_BASIC in mode:
        basics = set()
        if not corrects:
            corrects = self._get_correct_words(word, clear_word=clear)
        for correct in corrects:
            basics |= self._get_base_parts(correct)

        filename = mode[cfg.MODE_BASIC]
        arguments[filename].append(basics)

    for filename in set(mode.values()):
        pattern = patterns[filename]
        args = arguments[filename]
        file = files[filename]

        print_results(word, pattern, args, file, verbose)
```

Рисунок 3.8.2 – Демонстрация части метода analyze, вызывающий основные методы

4 Тестирование

Как утверждалось раннее, тестирование программы реализовано с использованием модуля unittest. Тестирование произведено для каждого модуля. Общее покрытие тестами можно увидеть на рисунке 4.1.

Coverage

Name	Stmts	Miss	Cover
analyzer/__init__.py	1	0	100%
analyzer/bk_tree.py	42	0	100%
analyzer/configurator.py	129	19	85%
analyzer/load_data.py	20	0	100%
analyzer/methods.py	85	4	95%
analyzer/text_splitter.py	28	0	100%
analyzer/word_analyzer.py	166	62	63%
TOTAL	471	85	82%

Рисунок 4.1 – Общее покрытие тестами всех модулей программы

5 Развёртывание программного средства

Для развёртывания данного программного средства существует 3 основных способа:

1) Использовать модуль main.py;
2) Установить программу с помощью модуль setup.py;
3) Использовать Docker-образ.

Чтобы воспользоваться первым способом, пользователю достаточно использовать утилиту python с аргументом main.py. Данный способ позволяет использовать программу, не устанавливая и при этом совершенствовать её и изменять. Однако, это самый неудобный в случае использования приложения способ.

Второй способ подразумевает непосредственную установку программного средства на машину пользователя. Данный способ является самым непредпочтительным, однако заметно упрощает использование программы.

Использование Docker образа является наиболее безопасным, и удобным вариантом развёртывания данного программного средства. Для построения докер-образа достаточно использовать Dockerfile, расположенный в корневой директории проекта. Данный файл соберет образ, и пользователь сможет запустить докер-контейнер с установленным на него программным средством (См. Рисунок 5.1).

```
FROM ubuntu
MAINTAINER lpshkn

RUN apt-get update && apt-get install -y python3 && apt-get install -y python3-setuptools \
&& apt-get install -y python3-pip

COPY . /wordanalyzer
WORKDIR /wordanalyzer

RUN pip3 install numpy
RUN python3 setup.py install && python3 setup.py test
```

Рисунок 5.1 – Демонстрация скрипта построения Docker-образа

6 Измерение времени работы программного средства

У разработанного программного средства существует 4 режима работа. Все режима работы пользователь может запустить одновременно, и как становится понятно, это изменяет скорость работы приложения.

Самый быстрый режим – режим подсчёта стоимости слова. Поскольку там не возникает никаких сложных вычислений, то, выбрав только этот режим, мы можем получить результат, продемонстрированный на рисунке 6.1.

```
Loading 300 words from ./data/short_rockyou.txt...
Loading all frequency words from ./data/frequency_words.txt...
The bk-tree is loading from ./data/tree.picle...
The bk-tree loaded successfully
word: march5th, cost: 3007.8927297778014
word: mikes17, cost: 2014.1834863716094
word: RowNathan0, cost: 3000
word: arasangel26, cost: 2022.66329323724
word: hoters02, cost: 2021.1654228613281
word: barmee, cost: 23.052525049681996
      . . . . .
word: scar11, cost: 2011.3958607573088
word: 51096, cost: 5000
word: mrs.kevin1, cost: 6017.191581438246
word: up1234, cost: 4006.845410783521
word: 029618505, cost: 9000
Working time:  0:00:00.019968
```

Рисунок 6.1 – Демонстрация работы программы в режиме подсчёта стоимости

В данном случае, в выборке было 300 слов. Однако, если мы попробуем обработать слова на выборке из 300 слов в самом долгом режим – режиме выделения базовых частей из слова, время работы программы значительно возрастет, что объясняется более сложным вычислительным процессом. Данный факт продемонстрирован на рисунке 6.2.

Однако, при включении одновременно всех 4 режимов, общее время работы программы возрастет незначительно. Причиной этому является тот факт, что вычисляемые на одном режиме работы программы данные передаются в другие режимы работы. Иными словами, если мы очищаем слово, то результат работы данного режима будет передан в метод исправления слова. А от метода результат

будет передан в режим выделения базовых частей слова. Таким образом, приблизительное время работы всех режимов программы на небольшой выборке данных не будет значительно отличаться от времени работы режима выделения базовых частей (См. Рисунок 6.3).

```
Loading 300 words from ./data/short_rockyou.txt...
Loading all frequency words from ./data/frequency_words.txt...
The bk-tree is loading from ./data/tree.picle...
The bk-tree loaded successfully
word: bubbahganosh, base words: a
word: bubbahganosh, base words: bubba
word: bubbahganosh, base words: hano
word: bubbahganosh, base words: nosh
word: bubbahganosh, base words: vga
word: bubbahganosh, base words: hg
word: bubbahganosh, base words: sh
word: 02038359, base words: NONE
word: prostiaumana, base words: a
word: prostiaumana, base words: ianman
    .   .   .   .   .
word: 110818esdafe, base words: sda
word: 42376111575, base words: NONE
word: bodhib, base words: bodh
word: bodhib, base words: bodhi
word: bodhib, base words: ib
Working time:  0:00:05.482524
```

Рисунок 6.2 – Демонстрация работы программы в режиме выделения базовых частей слова

```
Loading 300 words from ./data/short_rockyou.txt...
Loading all frequency words from ./data/frequency_words.txt...
The bk-tree is loading from ./data/tree.picle...
The bk-tree loaded successfully
word: jogologo, cost: 24.031681150787193, cleared: jogologo, corrected: {'jogologo'}, base words: {'logo', 'jogo'}
word: 505abcdefg, cost: 9003.849678509967, cleared: abcdefg, corrected: {'abcdefg'}, base words: {'abcd', 'fg'}
word: 35337679, cost: 8000, cleared: NONE, corrected: NONE, base words: NONE
word: 1986_child, cost: 9006.557728711068, cleared: child, corrected: {'child'}, base words: {'child'}
word: norween, cost: 32.895484121234276, cleared: norween, corrected: {'noween', 'norween', 'noreen'}, base words: {'now', 'no', 'rw', 'een', 'noreen'}
word: jhamar05, cost: 6007.699357019934, cleared: jhamar, corrected: {'jhamar', 'jamar', 'shamar'}, base words: {'mar', 'ja', 'a', 'shamar'}
word: claudiagar, cost: 24.48748361822267, cleared: claudiagar, corrected: {'claudiagar'}, base words: {'gar', 'claudia'}
          .   .   .   .   .
word: simonedawonderfuldancer, cost: 46.01468768421543, cleared: simonedawonderfuldancer, corrected: {'simonedawonderfuldancer'}, base words: {'wonder', 'dancer', 'eda', 'simon'}
word: loislover3, cost: 1023.5912356826556, cleared: loislover, corrected: {'loislover'}, base words: {'lover', 'loi'}
word: 058219079, cost: 9000, cleared: NONE, corrected: NONE, base words: NONE
word: fghj, cost: 25.755370292301656, cleared: fghj, corrected: {'fghj'}, base words: {'hj', 'fg'}
Working time:  0:00:07.950278
```

Рисунок 6.3 – Демонстрация работы программы в 4 режимах одновременно

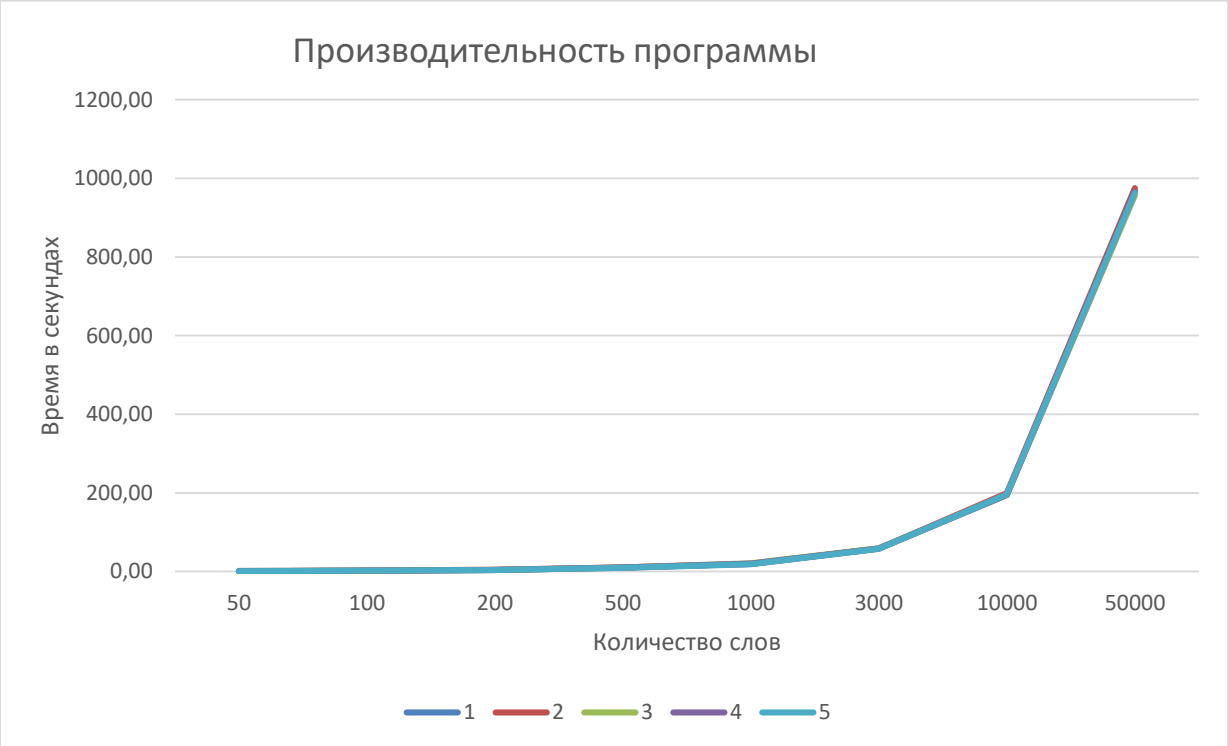
Чтобы иметь полное представление о времени, которое потребуется программе для обработки больших данных, мной было проверено время работы программы на количестве слов от 50 до 50000. Все данные приведены в таблице 6.1. Время в таблице указано в секундах.

Таблица 6.1 – Сравнение времени работы программы к количеству слов

Количество слов	50	100	200	500	1000	3000	10000	50000
1 тест	0,92	1,98	3,99	9,72	19,15	58,22	194,32	969,93
2 тест	0,93	1,93	4,26	10,01	19,18	58,70	199,24	975,03
3 тест	1,10	1,82	4,08	10,50	20,53	59,12	195,51	956,19
4 тест	1,04	1,96	4,03	10,00	20,07	58,51	194,94	965,73
5 тест	0,96	2,03	4,00	9,45	18,86	57,75	196,01	963,11
Ср. значение	0,99	1,94	4,07	9,94	19,56	58,46	196,00	966,00

На основе приведенных данных можно построить диаграмму, демонстрирующую зависимость времени от количества слов наглядно. Диаграмма приведена на диаграмме 6.1.

Диаграмма 6.1 – Зависимость времени работы программы от количества слов



Как мы можем заметить, то данная зависимость линейна и в среднем за 1 секунду обрабатывается 50 слов. Учитывая данное предположения, мы можем полагать, что имея словарь на 15 миллионов слов, программа обработает их за 83 часа 20 минут или за 300000 секунд.

Также стоит учитывать, что тестирование проводилось на конкретном компьютере и время обработки на других системах может значительно отличаться.

Заключение

В ходе выполненной практической работы мной было разработано программное средство, позволяющее обратить мутации слов с ощутимой точностью. Мной был разработан удобный пользователю интерфейс работы с программой, предоставляющий максимальную гибкость в настройке конфигурационных параметров, позволяющие получить оптимальный результат.

Данное программное средство было протестировано на нескольких наборах слов и была получена приблизительное количество времени необходимое для обработки больших массивов данных.