**ОТЧЁТ**

о выполнении практической работы по теме:

«Анализ возможностей по обращению мутации слов»

Выполнил:

Лепёшкин К.С.\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Москва

2020 г.

Оглавление

[Введение 3](#_Toc40831247)

[1 Задачи проекта 4](#_Toc40831248)

[2 Используемые инструменты 5](#_Toc40831249)

[2.1 Выбор языка программирования 5](#_Toc40831250)

[2.2 Выбор алгоритма для выделения лексем из текста без пробелов 5](#_Toc40831251)

[2.3 Выбор алгоритмов для исправления некорректных слов 5](#_Toc40831252)

[2.4 Выбор алгоритма для выделения базовых частей слова 7](#_Toc40831253)

[2.5 Выбор библиотеки для тестирования 7](#_Toc40831254)

[3 Разработка программного средства 8](#_Toc40831255)

[3.1 Структура проекта 8](#_Toc40831256)

[3.2 Основной модуль 9](#_Toc40831257)

[3.3 Модуль конфигуратора 10](#_Toc40831258)

[3.4 Модуль загрузки списка слов 12](#_Toc40831259)

[3.5 Модуль создания БК-дерева 13](#_Toc40831260)

[3.6 Модуль выделения лексем из текста без пробелов 14](#_Toc40831261)

[3.7 Модуль дополнительных функций 16](#_Toc40831262)

[3.7.1 Функция очистки слова 16](#_Toc40831263)

[3.7.2 Функция вывода результатов 17](#_Toc40831264)

[3.7.3 Функция построения шаблона вывода 18](#_Toc40831265)

[3.8 Модуль анализа слов 19](#_Toc40831266)

[4 Тестирование 21](#_Toc40831267)

[5 Развёртывание программного средства 22](#_Toc40831268)

[6 Измерение времени работы программного средства 23](#_Toc40831269)

[Заключение 27](#_Toc40831270)

# 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.

# 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.

# Используемые инструменты

## 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.

## 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.

## 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 С/С++ кода для Python. Код Cython преобразуется в С/С++ 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.

## 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.

## 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».

# 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.

## 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.

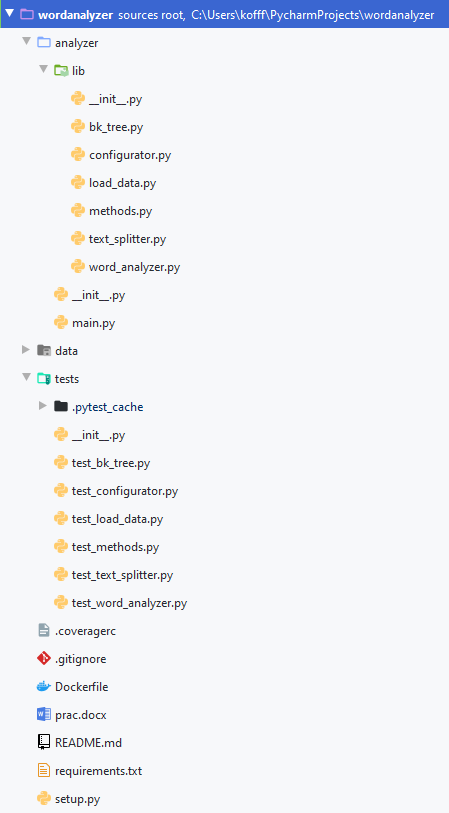


Рисунок 3.1.1 – Демонстрация структуры проекта

## 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()

## 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.

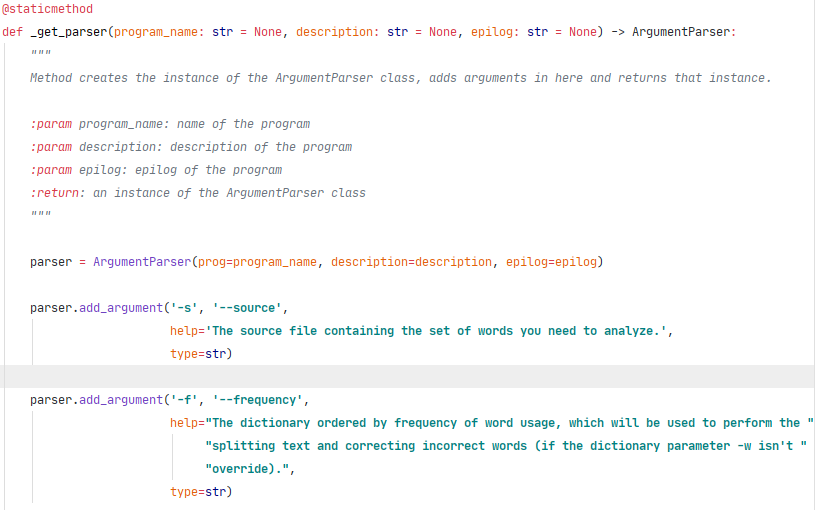


Рисунок 3.3.1 – Демонстрация метода создания объекта класса ArgumentParser



Рисунок 3.3.2 – Демонстрация методов получения параметров и загрузки слов

## 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.

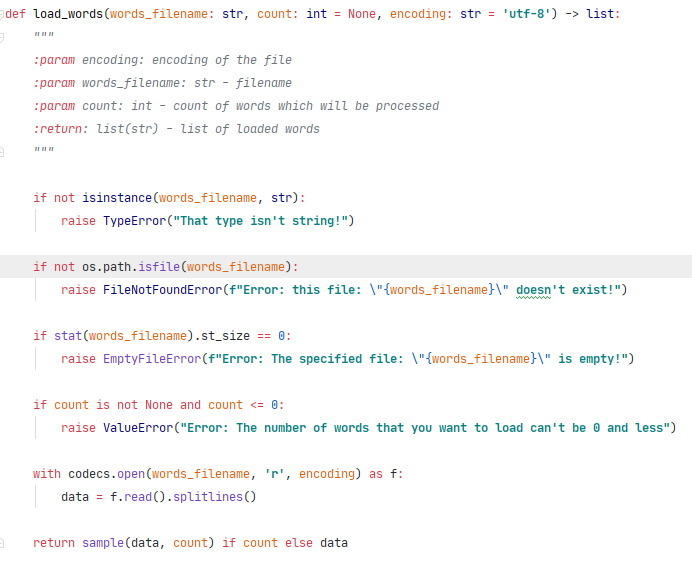


Рисунок 3.4.1 – Демонстрация основной функции модуля load\_data

## 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.



Рисунок 3.5.1 – Демонстрация модуля для работы с БК-деревом

## 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.

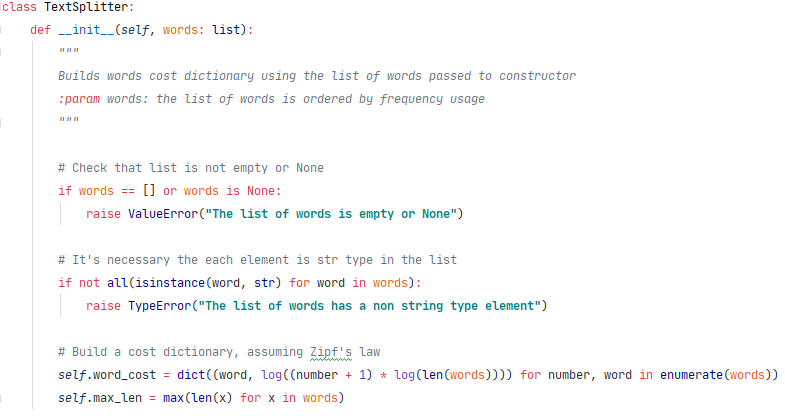


Рисунок 3.6.1 – Демонстрация конструктора класса TextSplitter

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).



Рисунок 3.6.2 – Демонстрация метода выделения слова из текста

## 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.

### 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)

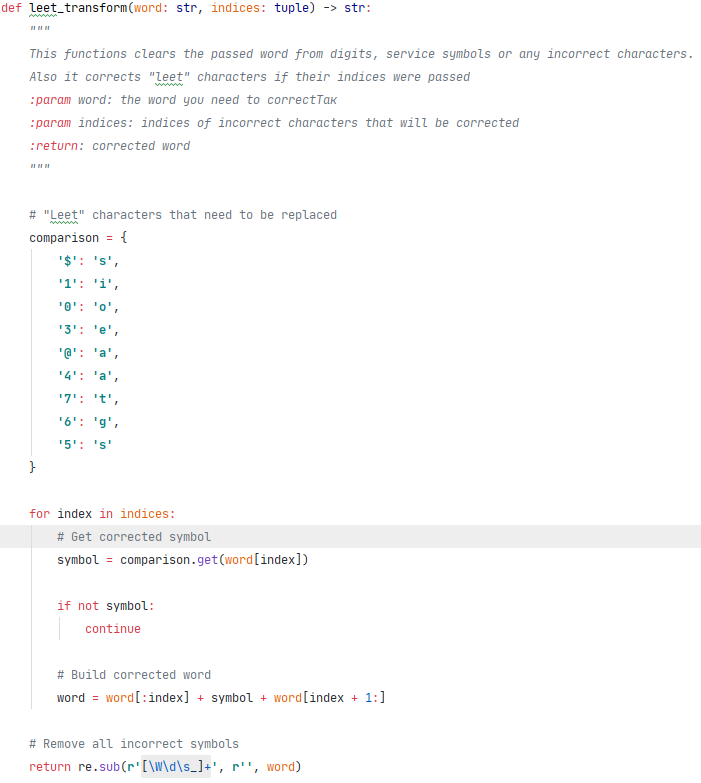


Рисунок 3.7.1.1 – Демонстрация функции удаления некорректных символов

### 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).

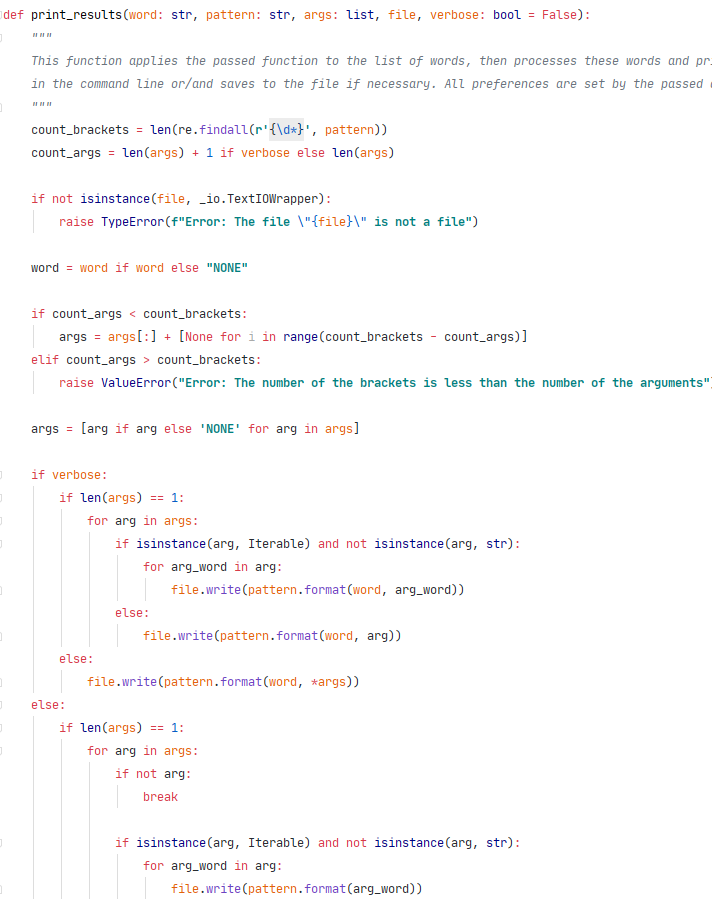


Рисунок 3.7.2.1 – Демонстрация функции вывода информации

### 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)

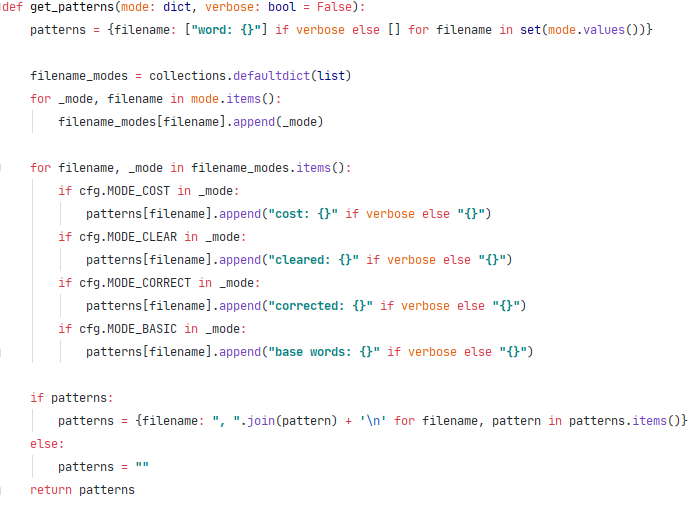


Рисунок 3.7.3.1 – Демонстрация функции построения шаблонов вывода

## 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.



Рисунок 3.8.1 – Демонстрация статического метода создания объекта класса WordAnalyzer

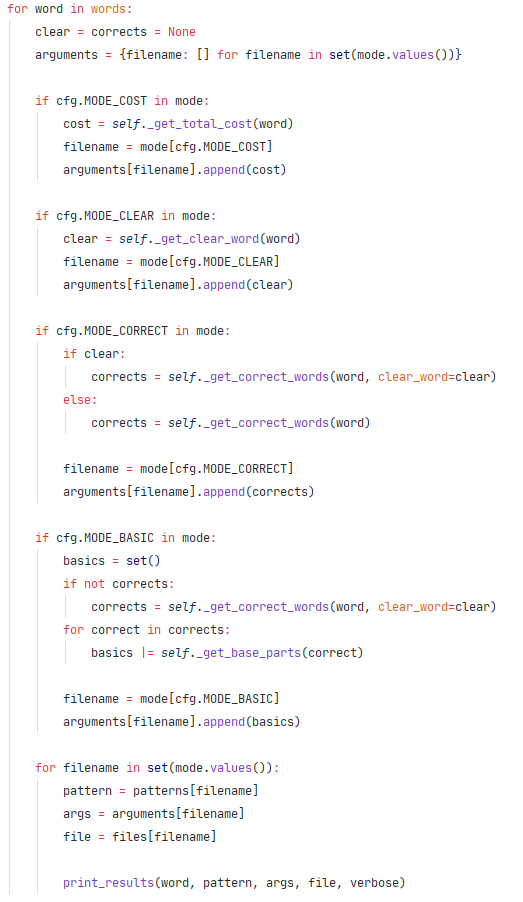


Рисунок 3.8.2 – Демонстрация части метода analyze, вызывающий основные методы

# 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.

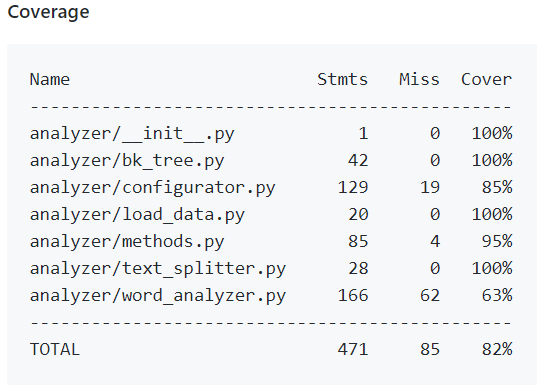


Рисунок 4.1 – Общее покрытие тестами всех модулей программы

# 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).

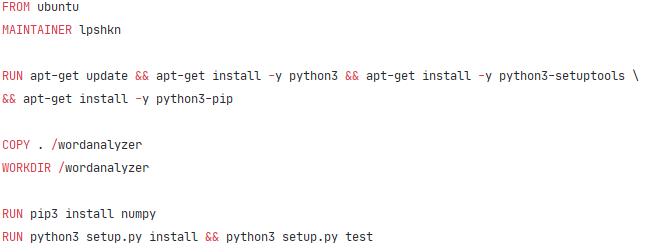


Рисунок 5.1 – Демонстрация скрипта построения Docker-образа

# 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.

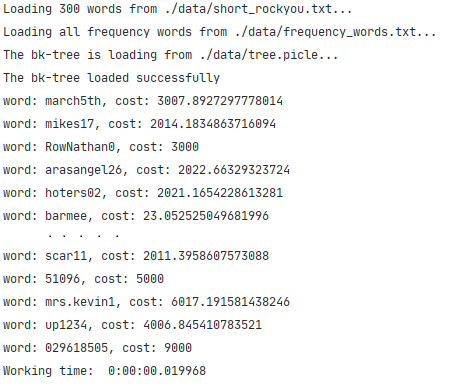


Рисунок 6.1 – Демонстрация работы программы в режиме подсчёта стоимости

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).

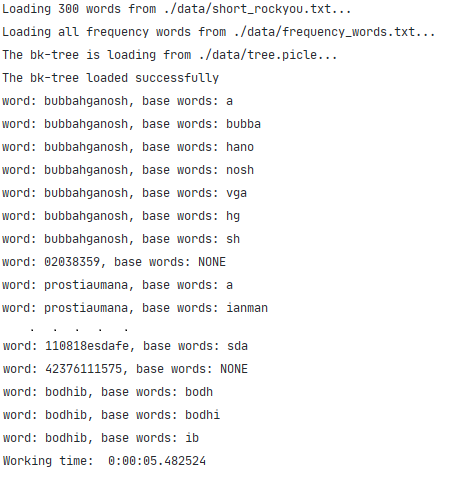


Рисунок 6.2 – Демонстрация работы программы в режиме выделения базовых частей слова

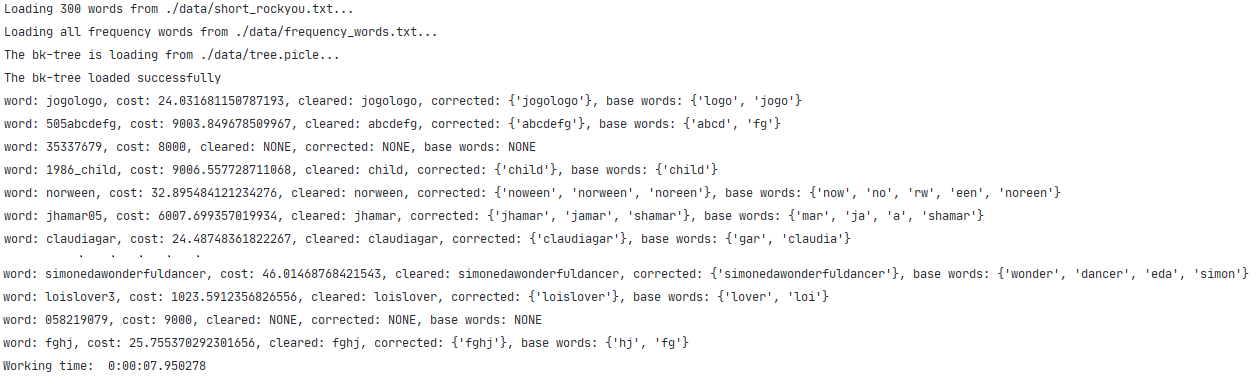


Рисунок 6.3 – Демонстрация работы программы в 4 режимах одновременно

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.