МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
СЕВЕРО-КАВКАЗСКИЙ ГОРНО-МЕТАЛЛУРГИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ)

Факультет	Информационных технологий и электронной техники
Кафедра	Информатики и вычислительной техники
Направление подготовки	09.03.01 Информатика и вычислительная техника
Профиль	Автоматизированные системы обработки информации и управления

«ДОПУСКАЕТСЯ К ЗАЩИТЕ»
Заведующий кафедрой ИВТ,
д.т.н., профессор

_____ Гроппен В.О.

«____»_____20__ г.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

на тему: «Разработка системы кластеризации документов на базе методов машинного обучения».

Студент	Кастуев Хетаг Асланович
Руководитель проекта	к.т.н., Будаева А.А.

г. Владикавказ 2021 г.

Задание

на выполнение выпускной квалификационной работы
студента группы ИВб-17-1 Кастуева Хетага Аслановича…,
обучающегося по направлению подготовки 09.03.01
«Информатика и вычислительная техника»
Профиль «Автоматизированные системы обработки информации и управления»

1. **Тема работы:**

Разработка системы кластеризации документов на базе методов машинного обучения.

(утверждена приказом по СКГМИ (ГТУ) №806/Об от 30.12.2020)

2. **Срок сдачи студентом законченной работы:** 21.06.2021
3. **Исходные данные к работе:**
   3.1. Изображения, разбитые на фрагменты
4. **Содержание расчетно-пояснительной записки:**
   4.1. Введение.
   4.2. Глава 1. Аналитический обзор.
   4.3. Глава 2. Выбор и обоснование алгоритма.
   4.4. Глава 3. Выбор и обоснование программной реализации.
   4.5. Глава 4. Экспериментальная часть.
   4.6. Заключение.
   4.7. Список литературы
   4.8. Приложение А. Листинг программы.
5. **Перечень графического материала:**
   5.1. Цель и задачи
   5.2. Условные обозначения алгоритма кластеризации;
   5.3. Математические модели кластеризации;
   5.4. Алгоритм работы программы кластеризации;

5.5. Условные обозначения алгоритма классификации;

5.6. Математические модели классификации;

5.7. Алгоритм работы программы классификации;

5.8. Интерфейс программы;

5.9. Результаты экспериментального тестирования.

6. **Дата выдачи задания:** 30.12.2020

Научный руководитель: _____ доц., к.т.н. Будаева А.А.

Задание принял к исполнению: 30.12.2020

Студент _____ Кастуев Х. А.

РЕФЕРАТ

Пояснительная записка 84 с., 21 рис., 11 табл., 12 ист., 1 прил. 4 графика.
Кластеризация, классификация, документы.

Объект разработки: системы кластеризации документов на базе методов машинного обучения.

Цель выпускной работы: создание ПП, способного кластеризовывать документы.

Использованное прикладное и системное программное обеспечение:
Операционная система Windows 10 PRO – 64 bit.
Среда разработки IntelliJ IDEA.

Характеристики вычислительной системы:
AMD Ryzen 5 3600 6-Core Processor 3.6 (4,2) ГГц
NVIDIA GeForce RTX 2060 6GB
ОЗУ 16Гб

Оглавление

ВВЕДЕНИЕ

Трудно представить себе жизнь современного человека без машинного обучения и искусственного интеллекта. Данные технологии создают новые возможности. Например, системы безопасности в метро используют машинное обучение для распознавания и поиска лиц, которые находятся в розыске. Компьютерное зрение используется в беспилотных автомобилях и летательных аппаратах для коммерческих и военных целей.

Машинный интеллект - это направление технологических исследований. Оно связано с созданием системы, которая способна обучаться самостоятельно.

Искусственный интеллект широко применяется в самых разных областях, как:

1. lT-сфера: разработка приложений;
2. Рекламные компании;
3. Маркетинговые исследования;
4. Медицинская диагностика;
5. Техническая диагностика;
6. Автопилотирование;
7. Биоинформатика и в другие сферы;
8. Обработка текста (в том числи кластеризация).

Все это и многое другое работает при помощи машинного интеллекта.

С одной стороны, такой технический прогресс приносит человеку огромную пользу, взять хотя бы медицинскую диагностику: многие исследования стали быстрыми и доступными, а с другой стороны такой технический прогресс бросает вызов каждому из нас: ведь совсем скоро не нужны будут машинисты в электропоездах, многий обслуживающий персонал заменят компьютеры, таксистов заменят беспилотные автомобили. Поэтому тему машинного обучения я считаю актуальной в настоящее время с научной

и практической точки зрения. Нужно не только знать, что существует так называемое машинное обучение, но и понимать, как это работает.

Машинное обучение избавляет программиста от необходимости писать большой код, т.е объяснять компьютеру, как нужно решить какую-нибудь проблему. В процессе машинного обучения компьютер учат самостоятельно находить правильное решение. То есть у нас есть какие-либо известные данные и на основе их мы собираем статистику и уже на новых данных мы учимся что-то понимать (находим новые, ещё не описанные закономерности). И задачи классификации и кластеризации документов так же являются частью машинного обучения.

В связи с наблюдаемым на протяжении последних десятилетий стремительным ростом накапливаемых объемов электронных документов особое значение приобретает разработка программных средств поиска информации.

Традиционными подходами к решению проблемы поиска информации в коллекциях полнотекстовых документов, являются поиск по ключевым словам. Однако, традиционные механизмы классификационного поиска не успевают изменяться вслед за темпом развития науки и техники или требуют высоких затрат как на адаптацию классификаторов.

Таким образом, в настоящее время существует потребность в разработке методов классификации, которые способны на основе анализа текстов и внутренних связей между ними автоматически строить рубрикаторы коллекций полнотекстовых документов. Среди известных методов автоматического анализа текстовых данных потенциально способных решить представленную проблему следует выделить методы кластеризации, которые автоматически разбивают документы на группы (кластеры) на основе анализа тематической близости между ними.

Таким образом, **объектом исследований** в дипломной работе являются способы кластеризации и классификации документов на базе методов машинного обучения.

Целью моей дипломной работы является создание программного средства, способного кластеризовать и классифицировать различные наборы документов.

Исходя из поставленной цели, были выделены следующие задачи:

1) Провести аналитический обзор предметной области.
2) Выбрать математическую модель и методов ее решения
3) Создать программный комплекс, реализующий выбранную математическую модель
4) Провести ряд экспериментов, показывающих эффективность разработанного программного комплекса.

ГЛАВА 1. АНАЛИТИЧЕСКИЙ ОБЗОР

Задача кластеризации — частный случай задачи машинного обучения, в частности, обучения без учителя, которая сводится к разбиению имеющегося множества объектов данных на подмножества таким образом, что элементы одного подмножества существенно отличались по некоторому набору свойств от элементов всех других подмножеств. Объект данных обычно рассматривается как точка в многомерном метрическом пространстве, каждому измерению которого соответствует некоторое свойство (атрибут) объекта, а метрика – есть функция от значений данных свойств. От типов измерений этого пространства, которые могут быть как числовыми, так и категориальными, зависит выбор алгоритма кластеризации данных и используемая метрика. Этот выбор продиктован различиями в природе разных типов атрибутов.

Кластеризация — задача группировки множества объектов на подмножества (кластеры) таким образом, чтобы объекты из одного кластера были более похожи друг на друга, чем на объекты из других кластеров по какому-либо критерию.

Классификация — это система распределения предметов, явлений или понятий какой-нибудь области на классы, разделы и разряды.

1.1. Цели кластеризации

- Понимание данных путём выявления кластерной структуры. Разбиение выборки на группы схожих объектов позволяет упростить дальнейшую обработку данных и принятия решений, применяя к каждому кластеру свой метод анализа (стратегия «разделяй и властвуй»).
- Сжатие данных. Если исходная выборка избыточно большая, то можно сократить её, оставив по одному наиболее типичному представителю от каждого кластера.

- Обнаружение новизны. Выделяются нетипичные объекты, которые не удаётся присоединить ни к одному из кластеров.

В первом случае число кластеров стараются сделать поменьше. Во втором случае важнее обеспечить высокую степень сходства объектов внутри каждого кластера, а кластеров может быть сколько угодно. В третьем случае наибольший интерес представляют отдельные объекты, не вписывающиеся ни в один из кластеров.

Во всех этих случаях может применяться иерархическая кластеризация, когда крупные кластеры дробятся на более мелкие, те в свою очередь дробятся ещё мельче, и т. д. Такие задачи называются задачами таксономии.

Результатом таксономии является древообразная иерархическая структура. При этом каждый объект характеризуется перечислением всех кластеров, которым он принадлежит, обычно от крупного к мелкому.

Классическим примером таксономии на основе сходства является биноминальная номенклатура живых существ, предложенная Карлом Линнеем в середине XVIII века. Аналогичные систематизации строятся во многих областях знания, чтобы упорядочить информацию о большом количестве объектов.

1.2. Применение

1.2.1. Биология и биоинформатика

- В области экологии кластеризация используется для выделения пространственных и временных сообществ организмов в однородных условиях;
- Кластерный анализ используется для группировки схожих геномных последовательностей в семейство генов, которые являются консервативными структурами для многих организмов и могут выполнять схожие функции;

- Кластеризация помогает автоматически определять генотипы по различным частям хромосом;
- Алгоритмы применяются для выделения небольшого числа групп генетических вариации человеческого генома.

1.2.2. Медицина

- Используется в позитронно-эмиссионной томографии для автоматического выделения различных типов тканей на трехмерном изображении;
- Определение заболеваний;
- Применяется для выявления шаблонов устойчивости к антибиотикам; для классификации антибиотиков по типу антибактериальной активности.

1.2.3. Маркетинг

- Кластеризация широко используется при изучении рынка для обработки данных, полученных из различных опросов. Может применяться для выделения типичных групп покупателей, разделения рынка для создания персонализированных предложений, разработки новых линий продукции.

1.2.4. Интернет

- Выделение групп людей на основе графа связей в социальных сетях;
- Повышение релевантности ответов на поисковые запросы путем группировки веб-сайтов по смысловым значениям поискового запроса.

1.2.5. Компьютерные науки

- Кластеризация используется в сегментации изображений для определения границ и распознавания объектов;
- Кластерный анализ применяется для определения образовавшихся популяционных ниш в ходе работы эволюционных алгоритмов для улучшения параметров эволюции;
- Подбор рекомендаций для пользователя на основе предпочтений других пользователей в данном кластере;
- Определение аномалий путем построения кластеров и выявления неклассифицированных объектов.

1.3. Виды алгоритмов

1.3.1. Алгоритмы квадратичной ошибки

Задачу кластеризации можно рассматривать как построение оптимального разбиения объектов на группы. При этом оптимальность может быть определена как требование минимизации среднеквадратической ошибки разбиения:

$$e^2(X, L) = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \left\| x_i^{(j)} - c_j \right\|^2 \quad (1.1)$$

где c_j — «центр масс» кластера j (точка со средними значениями характеристик для данного кластера).

Алгоритмы квадратичной ошибки относятся к типу плоских алгоритмов. Самым распространенным алгоритмом этой категории является метод k-средних. Этот алгоритм строит заданное число кластеров, расположенных как можно дальше друг от друга. Работа алгоритма делится на несколько этапов:

1. Случайно выбрать k точек, являющихся начальными «центрами масс» кластеров.
2. Отнести каждый объект к кластеру с ближайшим «центром масс».

3. Пересчитать «центры масс» кластеров согласно их текущему составу.
4. Если критерий остановки алгоритма не удовлетворен, вернуться к п. 2.

В качестве критерия остановки работы алгоритма обычно выбирают минимальное изменение среднеквадратической ошибки. Так же возможно останавливать работу алгоритма, если на шаге 2 не было объектов, переместившихся из кластера в кластер.

К недостаткам данного алгоритма можно отнести необходимость задавать количество кластеров для разбиения.

1.3.2. Алгоритмы иерархической кластеризации

Среди алгоритмов иерархической кластеризации выделяются два основных типа: восходящие и нисходящие алгоритмы. Нисходящие алгоритмы работают по принципу «сверху-вниз»: в начале все объекты помещаются в один кластер, который затем разбивается на все более мелкие кластеры. Более распространены восходящие алгоритмы, которые в начале работы помещают каждый объект в отдельный кластер, а затем объединяют кластеры во все более крупные, пока все объекты выборки не будут содержаться в одном кластере. Таким образом строится система вложенных разбиений. Результаты таких алгоритмов обычно представляют в виде дерева – дендрограммы. Классический пример такого дерева – классификация животных и растений.

Для вычисления расстояний между кластерами чаще все пользуются двумя расстояниями: одиночной связью или полной связью (см. обзор мер расстояний между кластерами).

К недостатку иерархических алгоритмов можно отнести систему полных разбиений, которая может являться излишней в контексте решаемой задачи.

1.3.3. Нечеткие алгоритмы

Наиболее популярным алгоритмом нечеткой кластеризации является алгоритм c-средних (c-means). Он представляет собой модификацию метода k-средних. Шаги работы алгоритма:

1. Выбрать начальное нечеткое разбиение n объектов на k кластеров путем выбора матрицы принадлежности U размера n x k. Используя матрицу U, найти значение критерия нечеткой ошибки:

$$E^2(X,U) = \sum_{i=1}^{N}\sum_{k=1}^{K} U_{ik} \left\| x_i^{(k)} - c_k \right\|^2 \quad (1.2)$$

   где c_k — «центр масс» нечеткого кластера k:

$$c_k = \sum_{i=1}^{N} U_k x_i. \quad (1.3)$$

2. Перегруппировать объекты с целью уменьшения этого значения критерия нечеткой ошибки.
3. Возвращаться в п. 2 до тех пор, пока изменения матрицы U не станут незначительными.

Этот алгоритм может не подойти, если заранее неизвестно число кластеров, либо необходимо однозначно отнести каждый объект к одному кластеру.

1.3.4. Алгоритмы, основанные на теории графов

Суть таких алгоритмов заключается в том, что выборка объектов представляется в виде графа G=(V, E), вершинам которого соответствуют объекты, а ребра имеют вес, равный «расстоянию» между объектами. Достоинством графовых алгоритмов кластеризации являются наглядность, относительная простота реализации и возможность вносения различных усовершенствований, основанные на геометрических соображениях. Основными алгоритмам являются алгоритм выделения связных компонент, алгоритм построения минимального покрывающего (остовного) дерева и алгоритм послойной кластеризации.

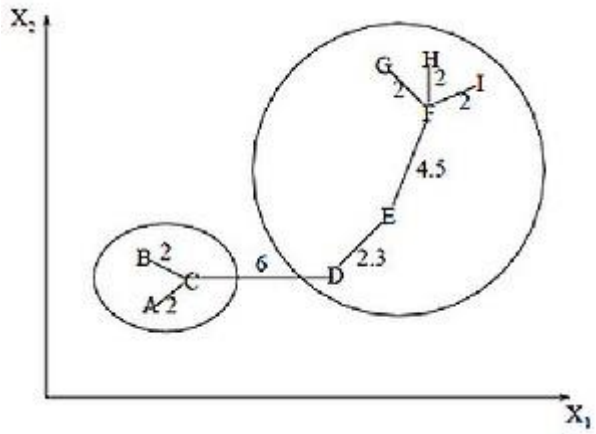
1.3.5. Алгоритм выделения связных компонент

В алгоритме выделения связных компонент задается входной параметр R и в графе удаляются все ребра, для которых «расстояния» больше R. Соединенными остаются только наиболее близкие пары объектов. Смысл алгоритма заключается в том, чтобы подобрать такое значение R, лежащее в диапазон всех «расстояний», при котором граф «развалится» на несколько связных компонент. Полученные компоненты и есть кластеры.

Для подбора параметра R обычно строится гистограмма распределений попарных расстояний. В задачах с хорошо выраженной кластерной структурой данных на гистограмме будет два пика – один соответствует внутрикластерным расстояниям, второй – межкластерным расстояния. Параметр R подбирается из зоны минимума между этими пиками. При этом управлять количеством кластеров при помощи порога расстояния довольно затруднительно.

1.3.6. Алгоритм минимального покрывающего дерева

Алгоритм минимального покрывающего дерева сначала строит на графе минимальное покрывающее дерево, а затем последовательно удаляет ребра с наибольшим весом. На рисунке изображено минимальное покрывающее дерево, полученное для девяти объектов.



Путём удаления связи, помеченной CD, с длиной равной 6 единицам (ребро с максимальным расстоянием), получаем два кластера: {A, B, C} и {D, E, F, G, H, I}. Второй кластер в дальнейшем может быть разделён ещё на два кластера путём удаления ребра EF, которое имеет длину, равную 4,5 единицам.

1.3.7. Послойная кластеризация

Алгоритм послойной кластеризации основан на выделении связных компонент графа на некотором уровне расстояний между объектами (вершинами). Уровень расстояния задается порогом расстояния *c*. Например, если расстояние между объектами $0 \le \rho(x, x') \le 1$, то $0 \le c \le 1$.

Алгоритм послойной кластеризации формирует последовательность подграфов графа *G*, которые отражают иерархические связи между кластерами:

$G^0 \subseteq G^1 \subseteq \ldots \subseteq G^m$, где $G^t = (V, E^t)$ — граф на уровне c^t, $E^t = \{e_{ij} \in E : \rho_{ij} \leq c_t\}$, c^t – t-ый порог расстояния, m – количество уровней иерархии, $G^0 = (V, o)$, o – пустое множество ребер графа, получаемое при $t^0 = 1$, $G^m = G$, то есть граф объектов без ограничений на расстояние (длину ребер графа), поскольку $t^m = 1$.

Посредством изменения порогов расстояния $\{c^0, \ldots, c^m\}$, где $0 = c^0 < c^1 < \ldots < c^m = 1$, возможно контролировать глубину иерархии получаемых кластеров. Таким образом, алгоритм послойной кластеризации способен создавать как плоское разбиение данных, так и иерархическое.

1.4. Сравнение алгоритмов

1.4.1. Вычислительная сложность алгоритмов

Алгоритм кластеризации	**Вычислительная сложность**
Иерархический	O(n2)
k-средних c-средних	O(nkl), где k – число кластеров, l – число итераций
Выделение связных компонент	зависит от алгоритма
O(n2 log n)	Минимальное покрывающее дерево
Послойная кластеризация	O(max(n, m)), где m

1.4.2. Сравнительная таблица алгоритмов

Алгоритм кластеризации	**Форма кластеров**	**Входные данные**	**Результаты**
Иерархический	Произвольная	Число кластеров или порог расстояния для усечения иерархии	Бинарное дерево кластеров
k-средних	Гиперсфера	Число кластеров	Центры кластеров
c-средних	Гиперсфера	Число кластеров, степень нечеткости	Центры кластеров, матрица принадлежности
Выделение связных компонент	Произвольная	Порог расстояния R	Древовидная структура кластеров
Минимальное покрывающее дерево	Произвольная	Число кластеров или порог расстояния для удаления ребер	Древовидная структура кластеров

Алгоритм кластеризации	Форма кластеров	Входные данные	Результаты
Послойная кластеризация	Произвольная	Последовательность порогов расстояния	Древовидная структура кластеров с разными уровнями иерархии

1.5. Мера расстояний

Итак, как же определять «похожесть» объектов? Для начала нужно составить вектор характеристик для каждого объекта — как правило, это набор числовых значений, например, рост-вес человека. Однако существуют также алгоритмы, работающие с качественными (т.н. категорийными) характеристиками.

После того, как мы определили вектор характеристик, можно провести нормализацию, чтобы все компоненты давали одинаковый вклад при расчете «расстояния». В процессе нормализации все значения приводятся к некоторому диапазону, например, [-1, -1] или [0, 1].

Наконец, для каждой пары объектов измеряется «расстояние» между ними — степень похожести.

1.5.1. Евклидово расстояние

Наиболее распространенная функция расстояния. Представляет собой геометрическим расстоянием в многомерном пространстве:

$$p(x, x') = \sqrt{\sum_i^n (x_i - x_i')^2} \quad (1.4)$$

1.5.2. Квадрат евклидова расстояния

Применяется для придания большего веса более отдаленным друг от друга объектам. Это расстояние вычисляется следующим образом:

$$p(x, x') = \sum_i^n (x_i - x_i')^2 \quad (1.5)$$

1.5.3. Расстояние городских кварталов

Это расстояние является средним разностей по координатам. В большинстве случаев эта мера расстояния приводит к таким же результатам, как и для обычного расстояния Евклида. Однако для этой меры влияние отдельных больших разностей (выбросов) уменьшается (т.к. они не возводятся в квадрат). Формула для расчета манхэттенского расстояния:

$$p(x, x') = \sum_i^n |x_i - x_i'| \quad (1.6)$$

1.5.4. Расстояние Чебышева

Это расстояние может оказаться полезным, когда нужно определить два объекта как «различные», если они различаются по какой-либо одной координате. Расстояние Чебышева вычисляется по формуле:

$$p(x, x') = \max(|x_i - x_i'|) \quad (1.7)$$

1.5.5. Степенное расстояние

Применяется в случае, когда необходимо увеличить или уменьшить вес, относящийся к размерности, для которой соответствующие объекты сильно отличаются. Степенное расстояние вычисляется по следующей формуле:

$$p(x, x') = r\sqrt{\sum_i^n (x_i - x_i')^p} \quad (1.8)$$

где r и p – параметры, определяемые пользователем. Параметр p ответственен за постепенное взвешивание разностей по отдельным координатам, параметр r ответственен за прогрессивное взвешивание больших расстояний между объектами. Если оба параметра – r и p — равны двум, то это расстояние совпадает с расстоянием Евклида.

1.5.6. TF-IDF

TF-IDF — статистическая мера, которая используется для оценки важности слова в контексте документа, являющегося частью набора документов. Вес слова пропорционален частоте употребления этого слова в документе и обратно пропорционален частоте употребления слова во всех документах коллекции.

TF (term frequency — частота слова) — отношение числа вхождений некоторого слова к общему числу слов документа. Таким образом, оценивается важность слова в пределах отдельного документа.

$$tf(t, d) = \frac{n_t}{\sum_k n_k} \quad (1.9)$$

где nt есть число вхождений слова t в документ, а в знаменателе — общее число слов в данном документе.

IDF (inverse document frequency — обратная частота документа) — инверсия частоты, с которой некоторое слово встречается в документах коллекции.

$$idf(t, D) = \log \frac{|D|}{|\{d_i \in D \mid t \in d_i\}|} \quad (1.10)$$

где |D| — число документов в коллекции, $\{d_i \in D \mid t \in d_i\}$ — число документов из коллекции D, в которых встречается t.

Мера TF-IDF часто используется в задачах анализа текстов и информационного поиска, например, как один из критериев релевантности документа поисковому запросу, при расчёте меры близости документов при кластеризации.

1.6. Существующие программные средства

На данный момент существует не одно программное средство, реализующее различные алгоритмы кластеризации. Примером такого проекта является Apache Mahout. Apache Mahout — это открытый проект Apache Software Foundation (ASF), основной целью которого является создание масштабируемых алгоритмов машинного обучения, которые предлагаются для бесплатного использования по лицензии Apache. Mahout содержит реализации кластеризации, категоризации, CF и эволюционного программирования. Более того, там, где это разумно, он использует библиотеку Apache Hadoop, что позволяет Mahout эффективно масштабироваться в облаке. Помимо Apache Mahout также есть ряд других программых комплексов, например, Weka. Weka – это продукт, который задумывался как современная среда для разработки методов машинного обучения.

1.7. Заключение

В данной главе были описаны алгоритмы кластеризации и показано их сравнение.

Из рассмотренных в первой главе алгоритмов был выбран алгоритм минимального покрывающего дерева, так как он имеет произвольную форму

кластера, а в качестве критерия схожести используется мера TF-IDF, так как этот алгоритм был предложен в задании к выпускной работе.

Во второй главе будет более подробно описан алгоритм кластеризации с помощью минимального покрывающего дерева.

ГЛАВА 2. ВЫБОР И ОБОСНОВАНИЕ АЛГОРИТМА

2.1. Содержательная постановка задачи

Содержательная постановка задачи

Имеется фиксированное множество документов. Требуется разбить заданное множество документов на группы в соответствии с их семантической схожестью.

Предусмотреть 2 режима работы:

1) автоматическое выявление групп документов
2) выбор одной группы документа (из заранее определенных групп) на основании его содержания.

В качестве критериев схожести документов при кластеризации используется мера TF-IDF.

2.2. Обозначения и формальная постановка

2.2.1. Кластеризация

Обозначения:

D – исходное множество документов коллекции,

$d_i \in D$ – i-й документ множества D,

$t_{d_i,j}$ – j-е слово документа d_i,

$n(t_{d_i,j})$ – число вхождений t_j-го слова в документ d_i,

$N(d_i)$ – общее число слов в документе d_i, $\forall i : N(d_i) = \sum_j n(t_{d_i,j})$

$\forall i,j : tf(t_{d_i,j}) = \frac{n(t_{d_i,j})}{N(d_i)}$ – частота t_j-го слова (term frequency) в документе d_i,

$\forall i,j : idf(t_{d_i,j}, D) = \log \frac{|D|}{|\{d_i \in D \mid t_{d_i,j}|}$ – обратная частота документа (inverse document frequency) – инверсия частоты, с которой слово $t_{d_i,j}$ встречается в документах коллекции

$\left|\{d_i \in D \mid t_{d_i,j}\right|$ – число документов D, в которых встречается слово $t_{d_i,j}$ (когда $n(t_{d_i,j}) \neq 0$)

$Z(d_i, p_k)$ – булева переменная, равная 1 – если d_i-й документ относится к таксону p_k и 0 – в противном случае,

$tf\_idf(d_i, j) = tf\left(t_{d_i,j}\right) \times idf\left(t_{d_i,j}, D\right)$ – оценка важности слова в контексте документа (мера TF-IDF),

$r\left(d_i, d_j\right) = \frac{\sum_{k=1}^{n} tf\_idf(d_i,k) \times tf\_idf(d_j,k)}{\sqrt{\sum_{k=1}^{n} tf\_idf(d_i,k))^2} \times \sqrt{\sum_{k=1}^{n} tf\_idf(d_j,k))^2}}$ - косинусное расстояние между d_i-м и d_l-м документами на основании меры TF-IDF (2.1),

r_{max} – максимально допустимое расстояние между объектами одной группы,

T – множество групп (таксонов)

$p_k \in T$ – k-й таксон из множества T;

$$\begin{cases} |T| \to \min \\ \forall i,j,k, d_i \in D, d_j \in D,\ p_k \in T,\ Z(d_i, p_k) = 1, Z\left(d_j, p_k\right) = 1, i \neq j: \\ \qquad\qquad r\left(d_i, d_j\right) \cdot Z(d_i, p_k) \cdot Z\left(d_j, p_k\right) \leq r_{max} \\ \forall i : \sum_k Z(d_i, p_k) = 1 \\ \forall i,k : Z\left(d_j, p_k\right) = \overline{1,0} \end{cases} \quad (2.2)$$

Цель – разбить документы множества на минимальное число групп, при ограничении на максимально допустимое расстояние между документами одной группы.

2.2.2. Классификация

Обозначения:

D – исходное множество документов коллекции,

$d_j \in D$ – j-й документ множества D,

T – множество таксонов

$p_f \in T$ – наиболее подходящий таксон для выбранного документа;

c_{p_k} – центр k-го таксона

$r(c_{p_k}, d_j)$ – расстояние до центра таксона p_k от документа d_j

$Z(d_i, p_k)$ – булева переменная, равная 1 – если d_i-й документ относится к таксону p_k и 0 – в противном случае

$$\begin{cases} \sum_{p_k \in T} \sum_{d_j \in D} Z(d_j, p_k) \to \max \\ \forall d_j \in D : \min\limits_k r(c_{p_k}, d_j) \cdot Z(d_j, p_k) \le r_{max} \\ \forall d_j \in D : \exists p_f \in T : \ r\left(c_{p_f}, d_j\right) = \min\limits_{p_k \in P} r(c_{p_k}, d_j) \\ \forall d_j \in D : \sum_{p_k \in T} Z(d_j, p_k) \le 1 \\ \forall d_j \in D, p_k \in T : Z(d_j, p_k) = \overline{1,0} \end{cases} \quad (2.3)$$

Цель – требуется классифицировать максимальное число документов так, что расстояние между документом и выбранным таксоном не превышало заданного ограничения.

2.2. Алгоритм

2.2.1. Кластеризация

Шаг 1. Готовим данные к обработке:

Шаг 1.1. Считать содержимое всех файлов;

Шаг 1.2. Объединить слова в единый список (словарь);

Шаг 1.3. Удалить слова, которые не несут смысловой нагрузки;

Шаг 1.4. Получить основу каждого слова

Шаг 2. Строим матрицу схожести документов, используя TF-IDF и косинусное расстояния

$$tf(t_{d_i,j}) = \frac{n(t_{d_i,j})}{N(d_i)}, \quad (2.4)$$

$$idf\left(t_{d_i,j}, D\right) = \log \frac{|D|}{\left|\{d_i \in D \mid t_{d_i,j}\right|}, \tag{2.5}$$

$$tf\_idf\left(t_{d_i,j}\right) = tf\left(t_{d_i,j}\right) \times idf\left(t_{d_i,j}, D\right). \tag{2.6}$$

$$r\left(d_i, d_j\right) = \frac{\sum_{k=1}^{n} tf\_idf(d_i,k) \times tf\_idf(d_j,k)}{\sqrt{\sum_{k=1}^{n} tf\_idf(d_i,k))^2} \times \sqrt{\sum_{k=1}^{n} tf\_idf(d_j,k))^2}} \tag{2.7}$$

Шаг 3. Нормализуем матрицу, полученную на предыдущем шаге

$$r\left(d_i, d_j\right)' = \frac{r\left(d_i,d_j\right) - \min\limits_{k}\min\limits_{l} r(d_k,d_l)}{\max\limits_{k}\max\limits_{l} r(d_k,d_l) - \min\limits_{k}\min\limits_{l} r(d_k,d_l)} \tag{2.7}$$

Шаг 4. Используя алгоритм Прима, строим минимальное остовное дерево;

Шаг 5. Удаляем из графа ребра, вес которых превышает r_{max}.

2.2.2. Классификация

Шаг 1. Готовим данные к обработке:

Шаг 1.1. Считать содержимое файла классов;

Шаг 1.2. Объединить ключевые слова классов в единый список (словарь);

Шаг 1.3. Считать содержимое файлов;

Шаг 1.4. Получить основу каждого слова, входящего в словарь

Шаг 2. Строим матрицу, у который по горизонтали классы и файлы, по вертикали количество повторений слов, входящих в словарь;

Шаг 3. Нормализуем матрицу по формуле

$$r\left(c_{p_k}, d_j\right)' = \frac{r\left(c_{p_k},d_j\right) - \min\limits_{i}\min\limits_{l} r\left(c_{p_i},d_l\right)}{\max\limits_{i}\max\limits_{l} r\left(c_{p_i},d_l\right) - \min\limits_{i}\min\limits_{l} r\left(c_{p_i},d_l\right)} \tag{2.8}$$

Шаг 4. Определяем группу выбранного документа по формуле (2.3). А все документы, вес которых меньше r_{max}, попадают в группу «Прочее».

2.3. Пример решения

2.3.1. Кластеризация

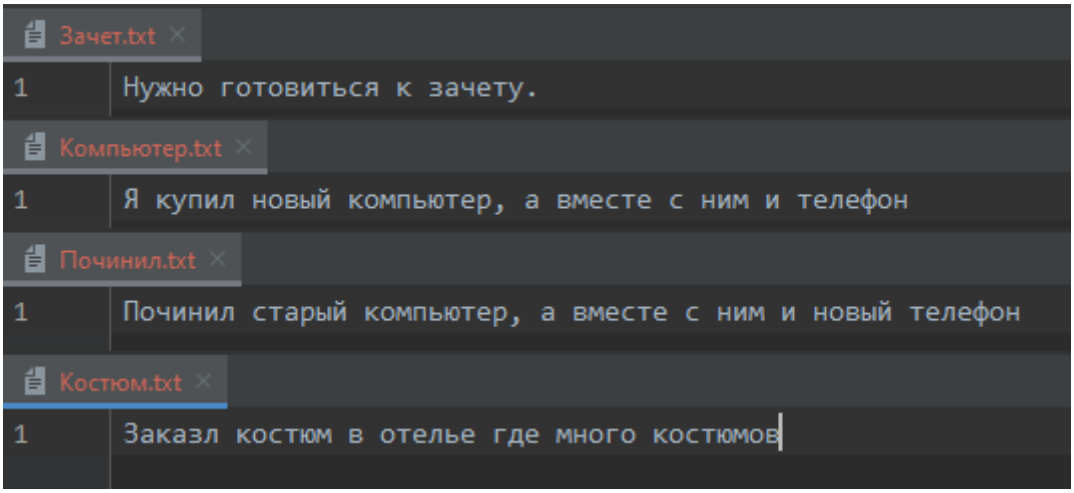


Рис. 2.1. Содержимое файлов

Шаг 1. Готовим данные к обработке:

Шаг 1.1. Считать содержимое всех файлов;

Шаг 1.2. Объединить слова в единый список (словарь);

Шаг 1.3. Удалить слова, которые не несут смысловой нагрузки;

Шаг 1.4. Получить основу каждого слова

Проделав эти действия получаем словарь, состоящий из 15 слов следующего вида: *вмест, готов, заказл, зачет, компьютер, костюм, куп, мног, нов, нужн, отел, перенесл, почин, стар, телефон*.

Для получения основы слов был использован стеммер Портера.

Шаг 2. Строим матрицу схожести документов;

Таблица 2.1.

Матрица схожести документов

	Зачет.txt	Компьютер.txt	Костюм.txt	Починил.txt
Зачет.txt	0	0.86	1	0.98
Компьютер.txt	0.86	2.22E-16	1	0.76

Костюм.txt	1	1	0	1
Починил.txt	0.98	0.76	1	0

Шаг 3. Нормализуем полученную на предыдущем шаге матрицу:

Таблица 2.2.

Нормализованная матрица схожести документов

	Зачет.txt	Компьютер.txt	Костюм.txt	Починил.txt
Зачет.txt	0	0.86	1	0.98
Компьютер.txt	0.86	0	1	0.76
Костюм.txt	1	1	0	1
Починил.txt	0.98	0.76	1	0

Шаг 4. Используя алгоритм Прима, строим минимальное остовное дерево:

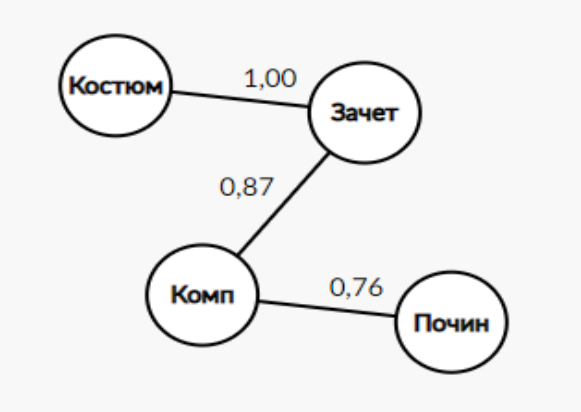


Рис. 2.2. Минимальное остовное дерево.

Шаг 5. Удаляем из графа ребра, вес которых превышает r_{max}:

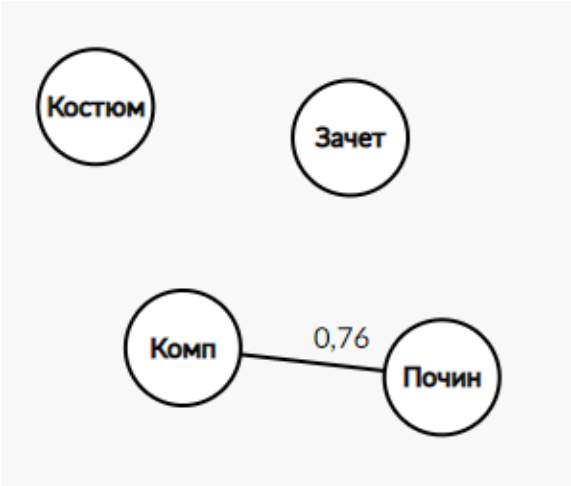


Рис. 2.3. Полученные группы.

В итоге получили следующий результат:

Группа №1: Компьютер.txt, Починил.txt

Группа №2: Зачет.txt

Группа №3: Костюм.txt

2.3.2. Классификация

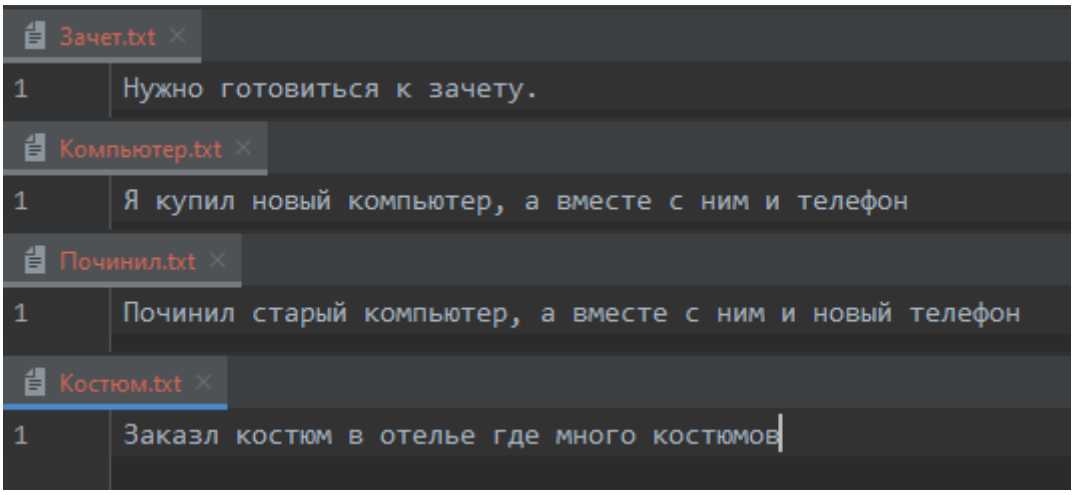


Рис. 2.4. Содержимое файлов

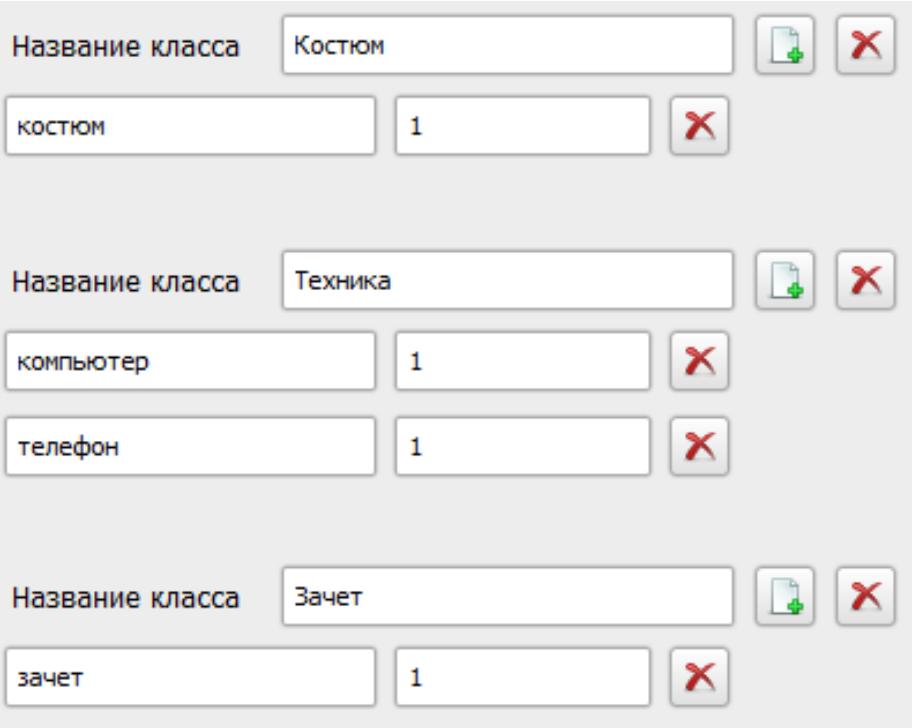


Рис. 2.5. Классы

Шаг 1. Готовим данные к обработке:

Шаг 1.1. Считать содержимое файла классов;

Шаг 1.2. Объединить ключевые слова классов в единый список (словарь);

Шаг 1.3. Считать содержимое файлов;

Шаг 1.4. Получить основу каждого слова, которое входит в словарь;

Проделав эти действия получаем словарь, состоящий из 4 слов следующего вида: *зачет, компьютер, костюм, телефон*;

Шаг 2. Строим матрицу, у который по горизонтали классы и файлы, по вертикали количество повторений слов, входящих в словарь;

Таблица 2.3.

Частоты терминов

	зачет	компьютер	костюм	телефон
Группа «Костюм»	0	0	1	0
Группа «Техника»	0	1	0	1
Группа «Зачет»	1	0	0	0
Зачет.txt	2	1	0	0
Компьютер.txt	1	1	0	1
Костюм.txt	0	0	2	0
Починил.txt	0	1	0	1

Шаг 3. Нормализуем матрицу;

Таблица 2.4.

Нормализованная таблица частот

	зачет	компьютер	костюм	телефон
Группа «Костюм»	0	0	1	0
Группа «Техника»	0	1	0	1
Группа «Зачет»	1	0	0	0
Зачет.txt	1	0,5	0	0
Компьютер.txt	1	1	0	1
Костюм.txt	0	0	1	0
Починил.txt	0	1	0	1

Шаг 4. Определяем группу выбранного документа по формуле (2.3). А все документы, вес которых меньше r_{max}, попадают в группу «Прочее»:

Таблица 2.5.

Результат классификации

	Группа «Костюм»	Группа «Техника»	Группа «Зачет»
Зачет.txt	1	0,68	0,1
Компьютер.txt	1	0,18	0,42
Костюм.txt	0	1	1
Починил.txt	1	0	1

Для примера пусть $r_{max} = 1$.

Из таблицы (2.3) наглядно виден класс, соответствуюий каждому документу:

Зачет.txt – Группа «Зачет»

Компьютер.txt - Группа «Техника»

Починил.txt - Группа «Техника»

Костюм.txt - Группа «Костюм»

Так как все не нулевые значения больше r_{max}, то в группу «Прочее» ничего не попадает.

2.4. Заключение

В главе предложена содержательная и формальная постановки задачи, в качестве меры сравнения документов используется косинусная мера; приведены алгоритмы для решения, и пример решения поставленной задачи.

ГЛАВА 3. ВЫБОР И ОБОСНОВАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ

3.1. Архитектура программного обеспечения

Разработанное программное средство реализовано на языке Java, что в свою очередь позволяет запускать данный ПП под управлением любой ОС, на которой заранее установлена соответствующая виртуальная машина.

В качестве среды для разработки программного продукта выбрана интегрированная среда разработки IntelliJ IDEA. Данная среда зарекомендовала себя, как надежный инструмент разработки, и имеет следующие преимущества:

1. Поддержка множества языков программирования;
2. Кроссплатформенность;
3. Наличие бесплатной версии (Community);
4. Удобный инструмент отладки;
5. Поддержка системы контроля версиями;

Из минусов можно выделить разве что высокое потребление ресурсов, но при текущих мощностях современных ЭВМ данный пункт нивелируется.

3.2. Язык программирования Java

Язык программирования в значительной степени влияет на качество и надежность программного обеспечения. Для реализации поставленной задачи был выбран язык программирования Java по следующим причинам:

1. Объектно-ориентированный подход к разработке. Это означает, что у разработчика появляется возможность описывать абстрактные конструкции на основе предметной области, а потом реализовывать между ними взаимодействие;
2. Большое количество готовых библиотек, ускоряющих разработку;
3. Удобная среда разработки (IntelliJ IDEA);

4. Безопасный код;
5. Кроссплатформенность.

Java — объектно-ориентированный язык программирования, разрабатываемый компанией Sun Microsystems и официально выпущенный 23 мая 1995 года.

3.3. Работа программного комплекса

В программе предусмотрено 2 режима работы:

1. Кластеризация
2. Классификация

Оба режима поддерживают обработку файлов с расширениями:

- .txt
- .docx

Далее будут продемонстрированы оба режима работы.

3.3.1. Режим «Кластеризация»

По умолчанию, при запуске программы, бывает выбран режим работы кластеризации, как это показано на рис. 3.1.

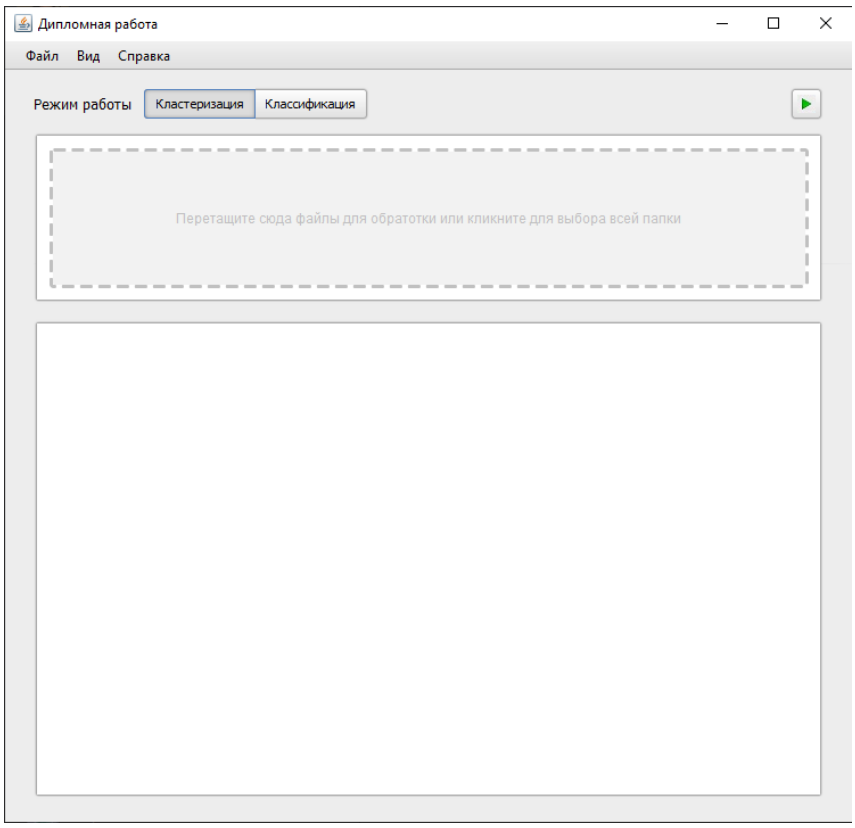


Рис. 3.1. Режим работы кластеризации.

Далее необходимо перетащить нужные файлы в соответствующую область или один раз кликнуть по той же области, для того чтобы в диалоговом окне выбрать папку, в которой мы хотим обработать файлы. Количество файлов должно быть больше двух.

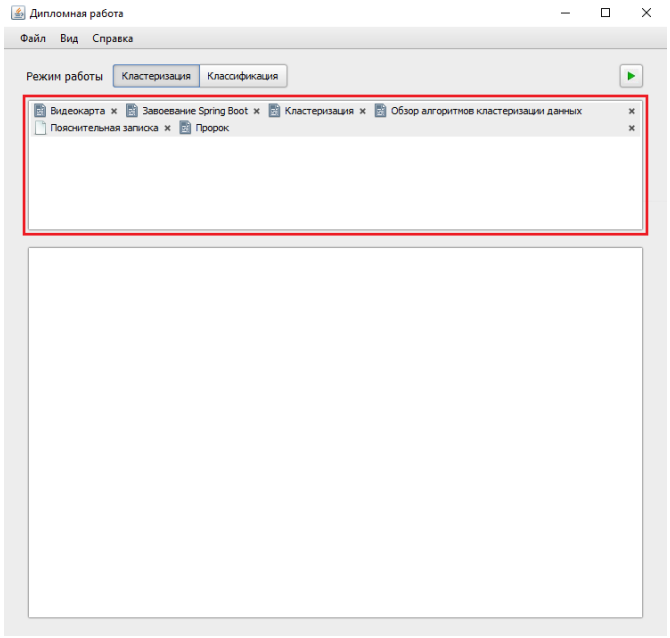


Рис. 3.2. Выбор файлов.

Далее нужно запустить программу.

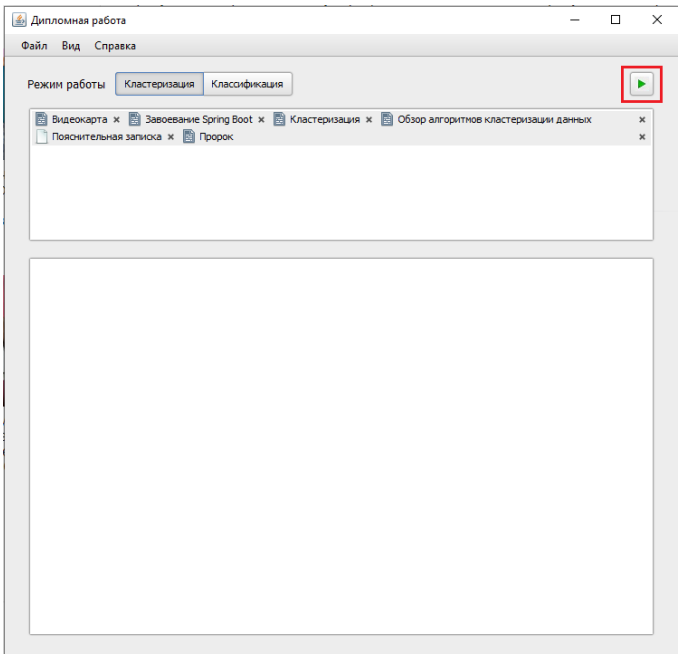


Рис. 3.3. Запуск.

И в текстовом поле появится результат работы

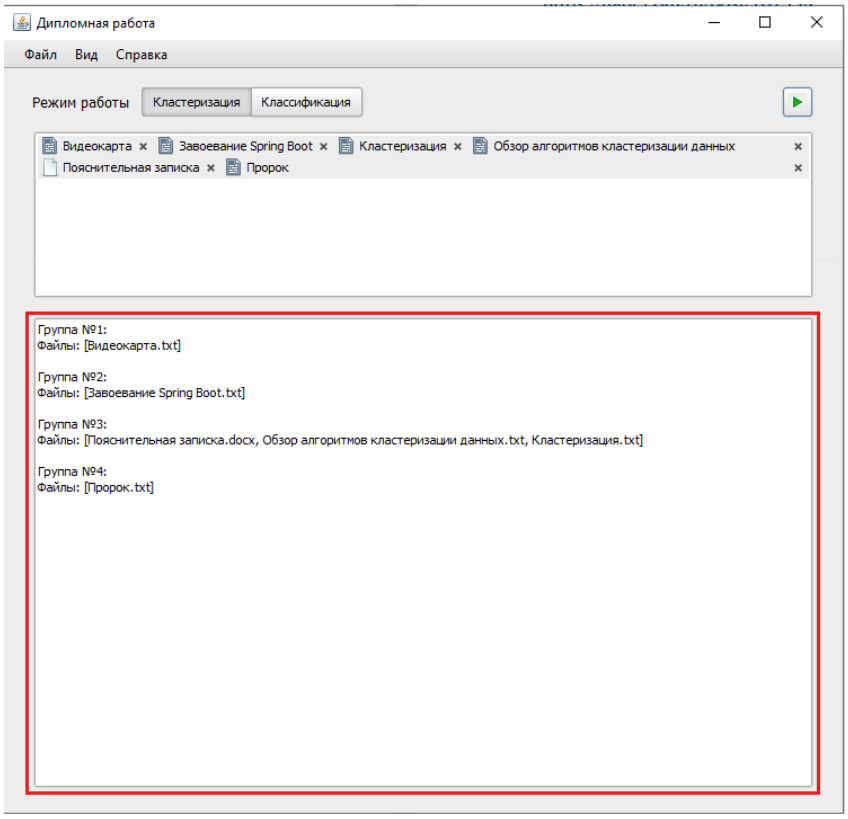


Рис. 3.4. Результат кластеризации.

3.3.2. Режим «Классификация»

Для того, чтобы программа начала работать в режиме классификации необходимо выбрать соответствующий режим:

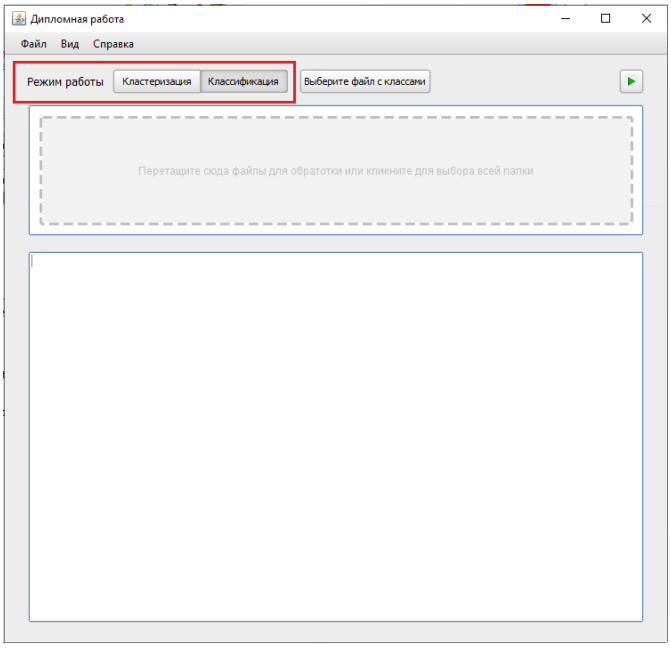


Рис. 3.5. Режим работы классификации.

Далее нужно выбрать файл с заранее заготовленными классами

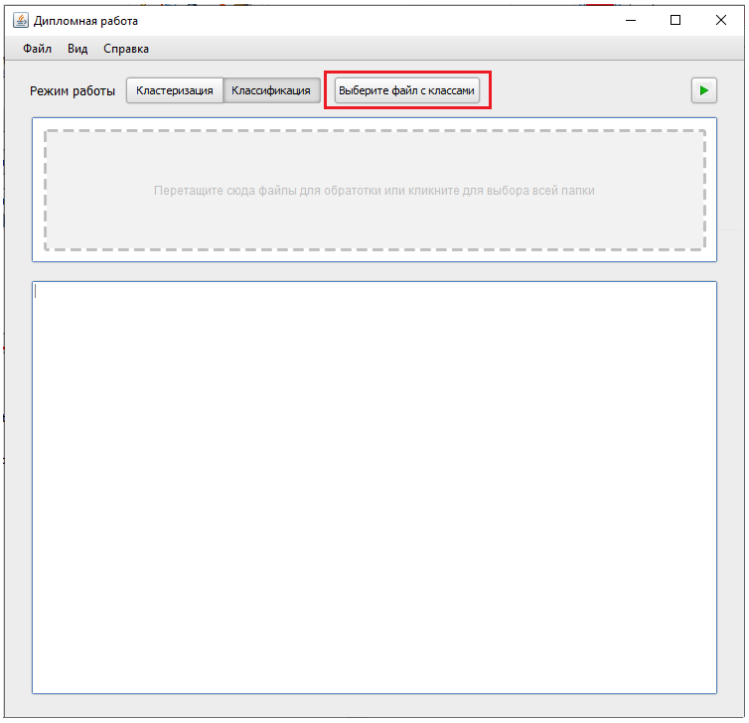


Рис. 3.6. Выбор файла классов.

Далее необходимо перетащить нужные файлы в соответствующую область или один раз кликнуть по той же области, для того чтобы в диалоговом окне выбрать папку, в которой мы хотим обработать файлы.

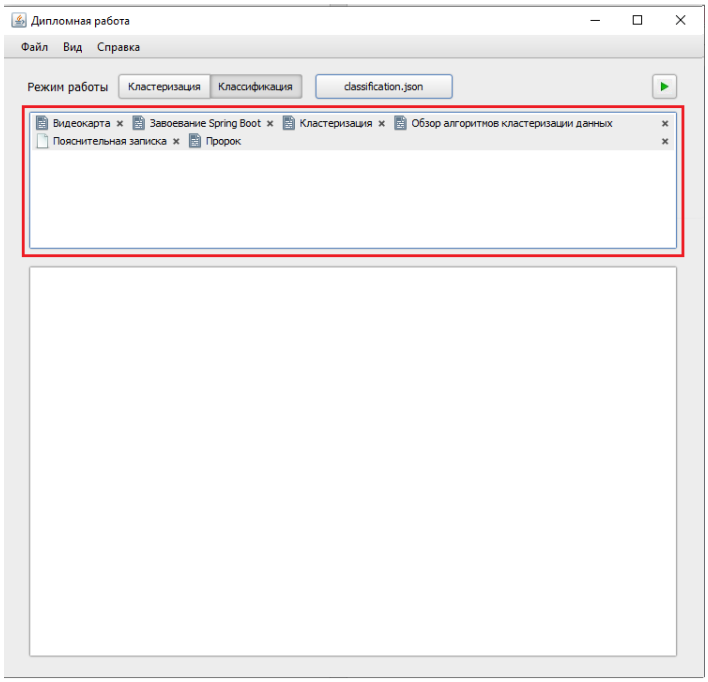


Рис. 3.7. Выбор файлов.

И, наконец, запустить программу

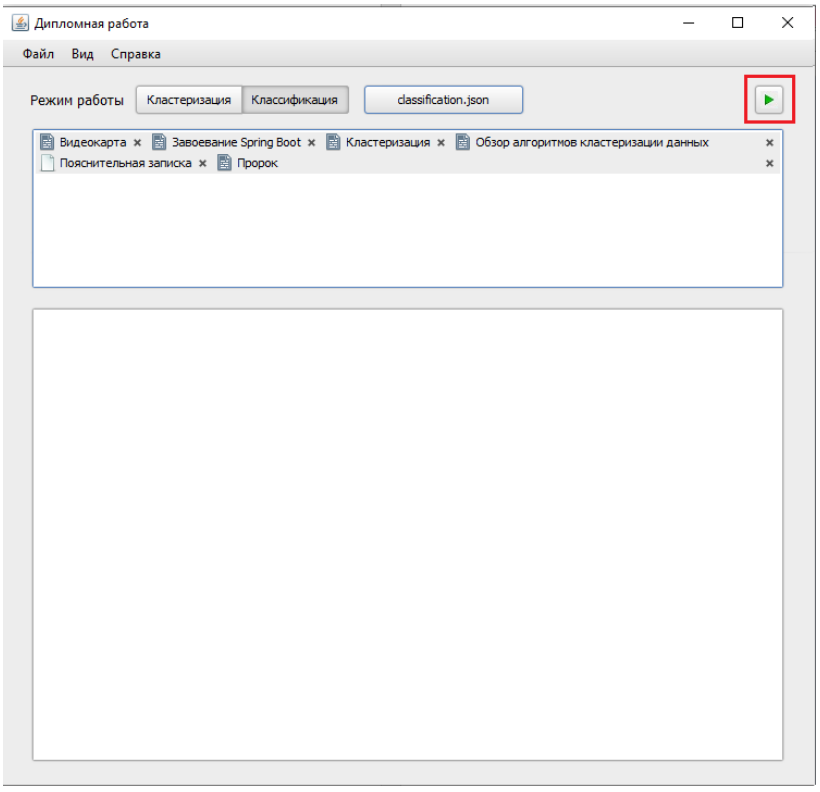


Рис. 3.8. Запуск.

И в текстовое поле выведется результат работы программы

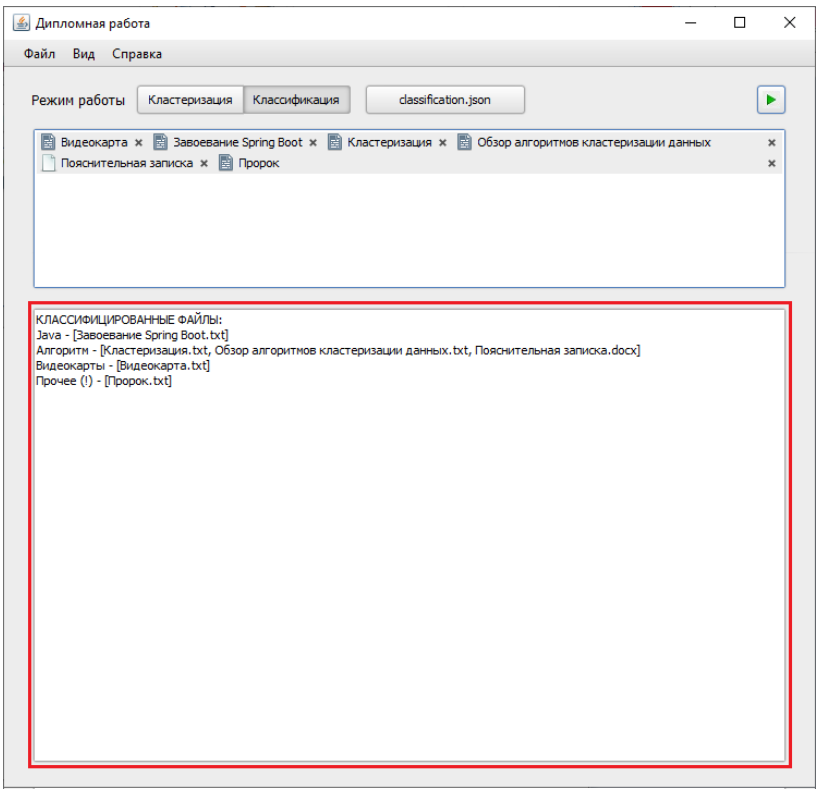


Рис. 3.9. Просмотр результатов.

3.3.3. Создание и редактирование файла классов

Также разработанной продукт продоставляет возможность создания и редактирования файлов с классами.

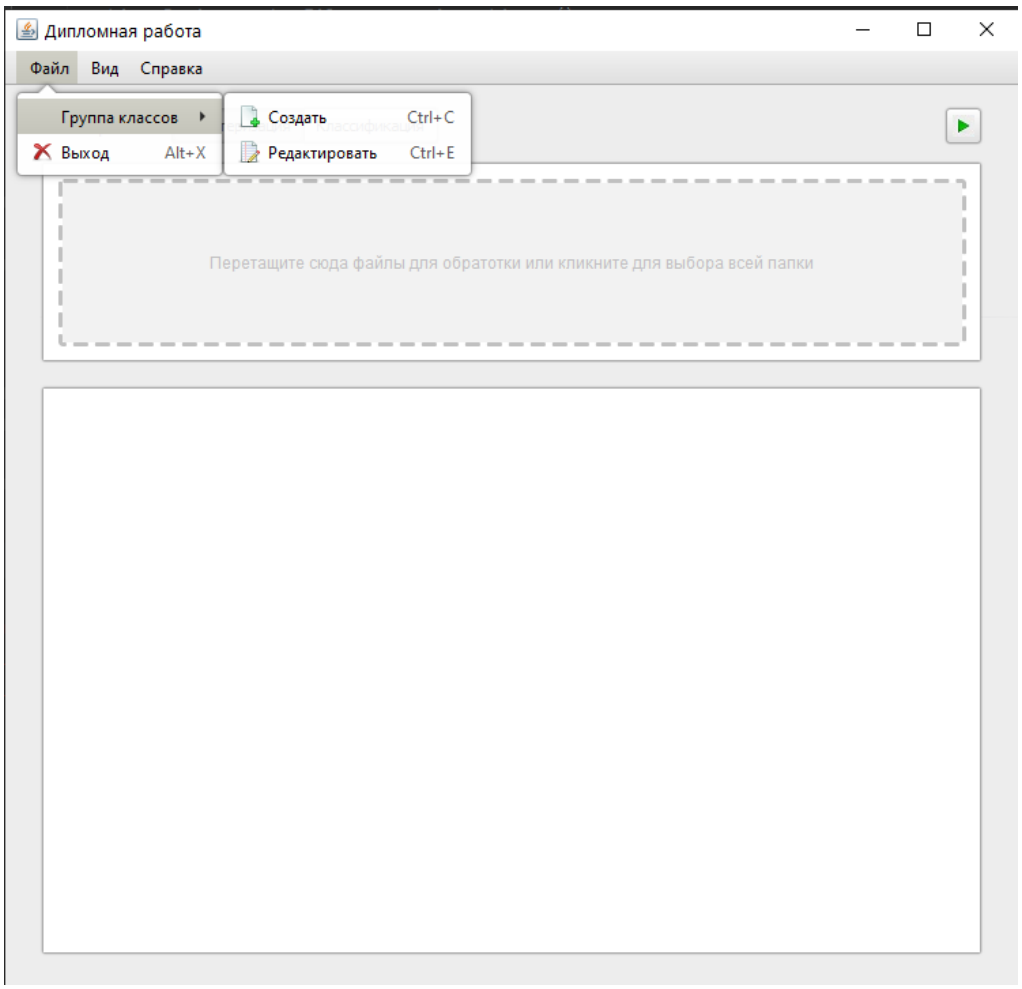


Рис. 3.10. Группы классов.

Ддя создания нужно выбрать соответствующий пункт меню, при нажатии на который откроется диалоговое окно следующего вида:

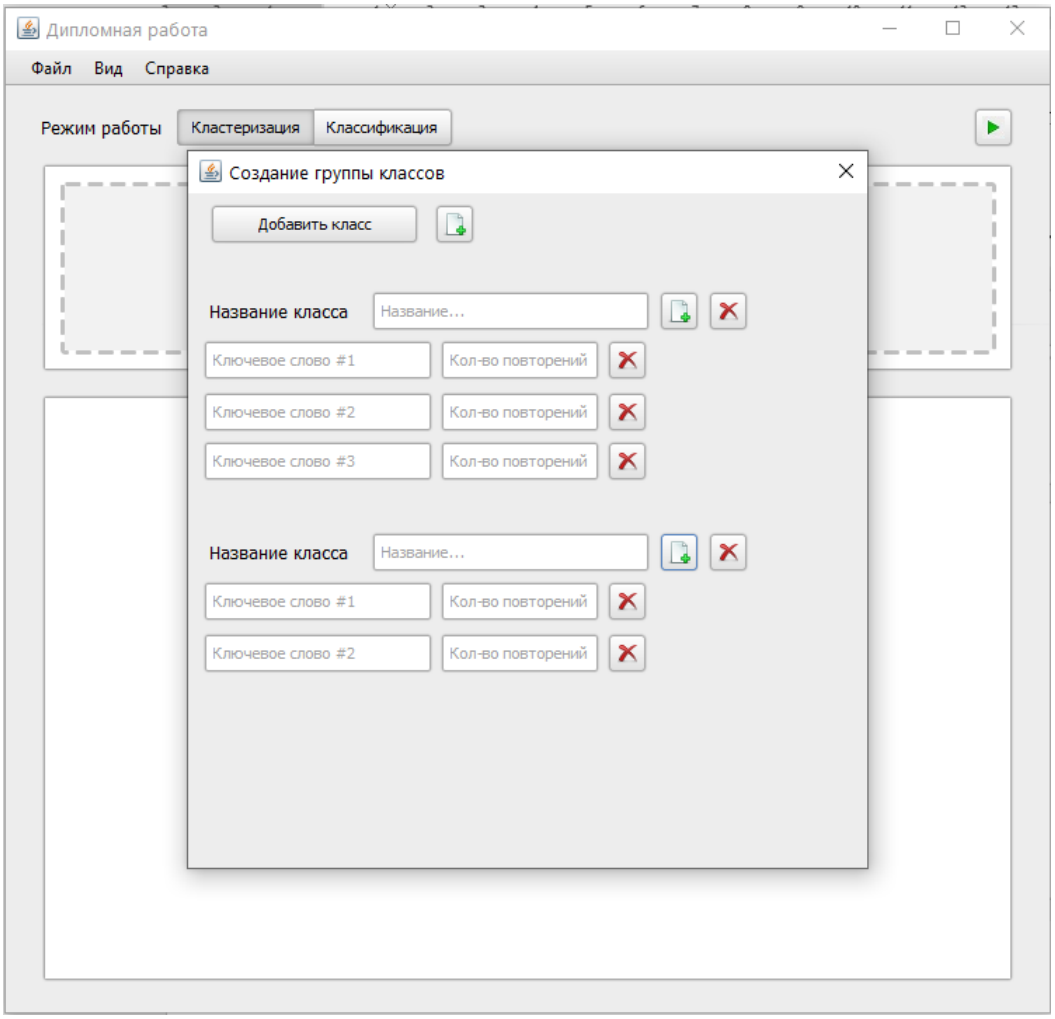


Рис. 3.11. Окно создания классов.

В нем пользователь может создать любое количество групп и указать любое количество ключевых слов данной группы. Для каждого ключевого слова нужно указать частоту повторений. Для каждой группы должно быть указано хотя бы одно ключевое слово. Также, при желании, пользователь может удалять группы и ключевые слова.

Для редактирования нужно выбрать соотвествующий пункт меню, при выборе которого, программа попросит выбрать файл, который требуется отредактировать. И затем откроет соответствующее деалоговое окно.

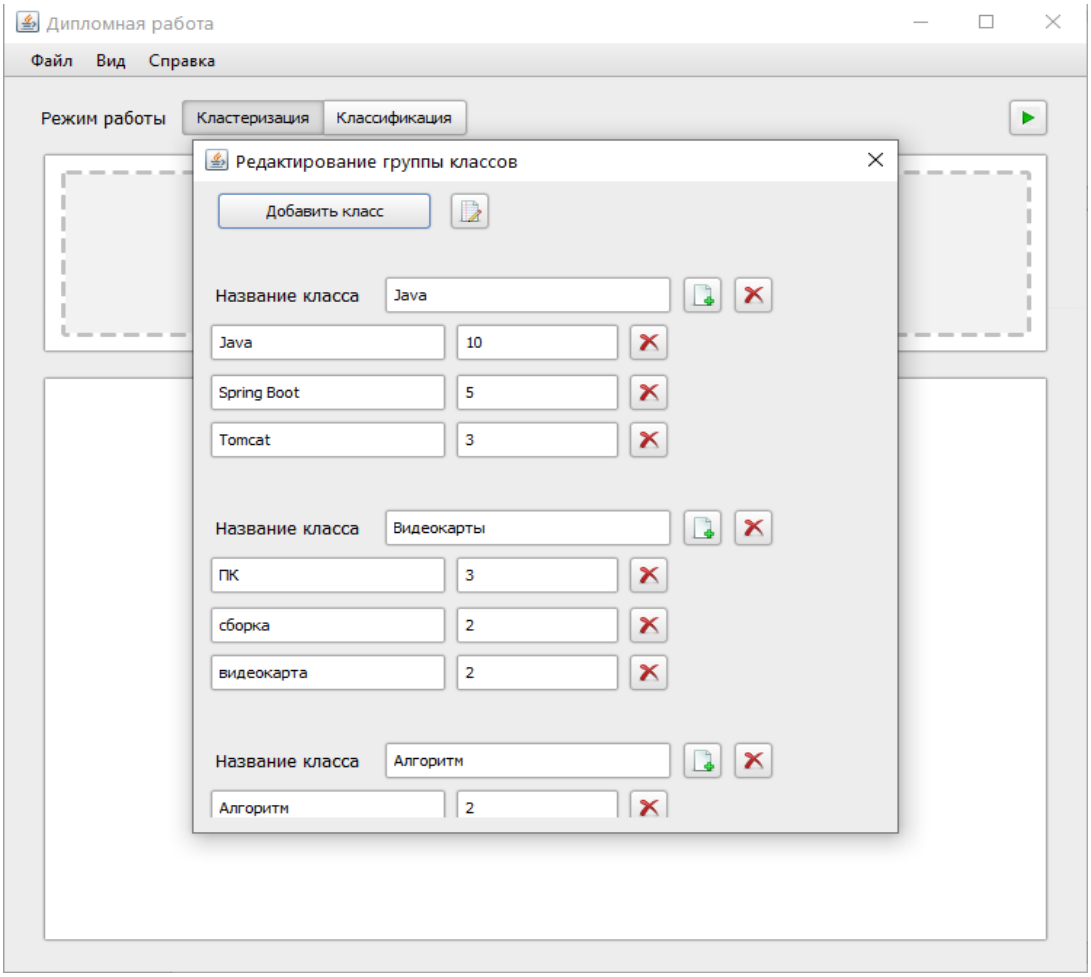


Рис. 3.12. Окно редактирования классов.

Тут так же присутвует возможность создания/удаления групп, добавления/удаления улючевых слов. В конце работы нужно сохранить изменения.

3.4. Заключение

В данной главе были рассмотрены среды разработки, выбран язык программирования, на котором создавался программный комплекс и была рассмотрена работа программного комплекса.

ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

Для проведения анализа была проведена серия экспериментов.

Для проведения экспериментов использовался ПК со следующими характеристиками:

Таблица 4.1.

Характеристики ПК

Производитель процессора	AMD
Тип процессора	Ryzen 5 3600
Количество ядер процессора	6
Баз. такт. частота	3.59 GHz
Кэш память	32 МБ
Оперативная память (RAM)	16 Гб
Тип памяти	DDR4
Производитель видеокарты	NVIDIA
Графический контроллер	GeForce RTX 2060
Жесткий диск (SSD)	256 ГБ
ОС	Windows 10 PRO (64 bit)

Главной исследуемой характеристикой работы программного комплекса является время его работы.

4.1. Оценка производительности алгоритма кластеризации

4.1.1. Зависимость времени от количества слов в файле

Цели эксперимента:

- Построить графические зависимости, отражающие среднее время кластеризации в зависимости от количества слов в файле;

– Определить аналитическую зависимость, отражающие среднее время кластеризации в зависимости от количества слов в файле.

Ход эксперимента:

Разместить N файлов, с количеством слов в диапазоне $5000 \leq N \leq 35000$ с шагом 5000. При каждом фиксированном N алгоритм отрабатывает 100 раз.

Таблица 4.2.

Зависимость времени кластеризации от количества слов в файле

Кол-во слов	Время работы (мс)
5000	2195,72
10000	2735,15
15000	3108,07
20000	3283,41
25000	3509,29
30000	3813,6

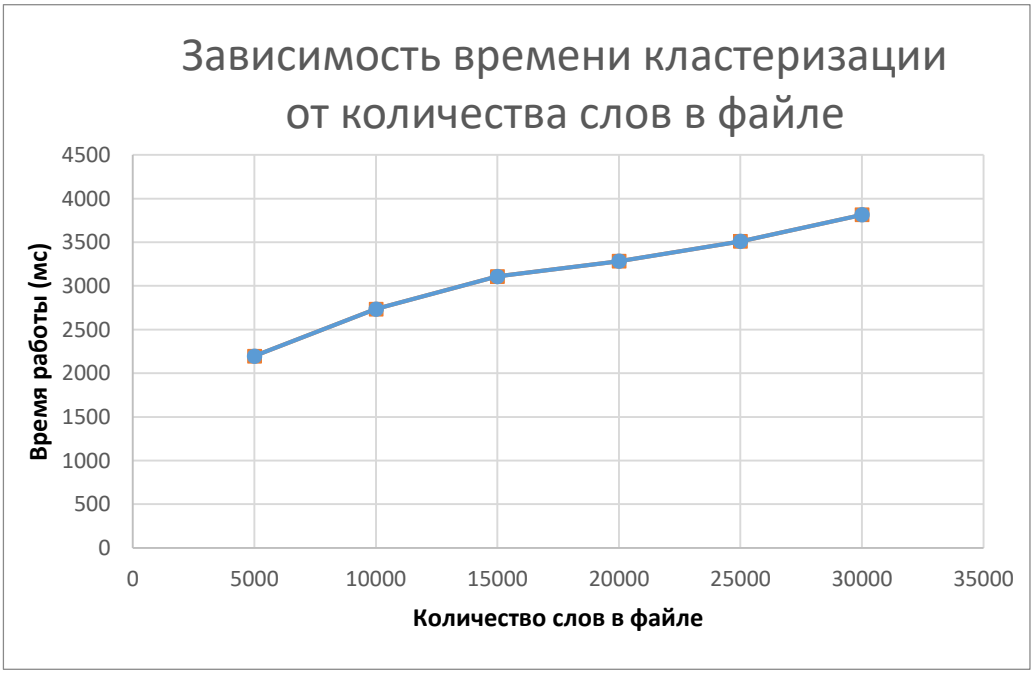


График 4.1. Зависимость времени кластеризации от количества слов в файле

Для оценки вида зависимости проведем аппроксимацию следующими видами функций:

а. Полиномиальная 2-й степени
б. Полиномиальная 3-й ступени
в. Степенная
г. Линейнай

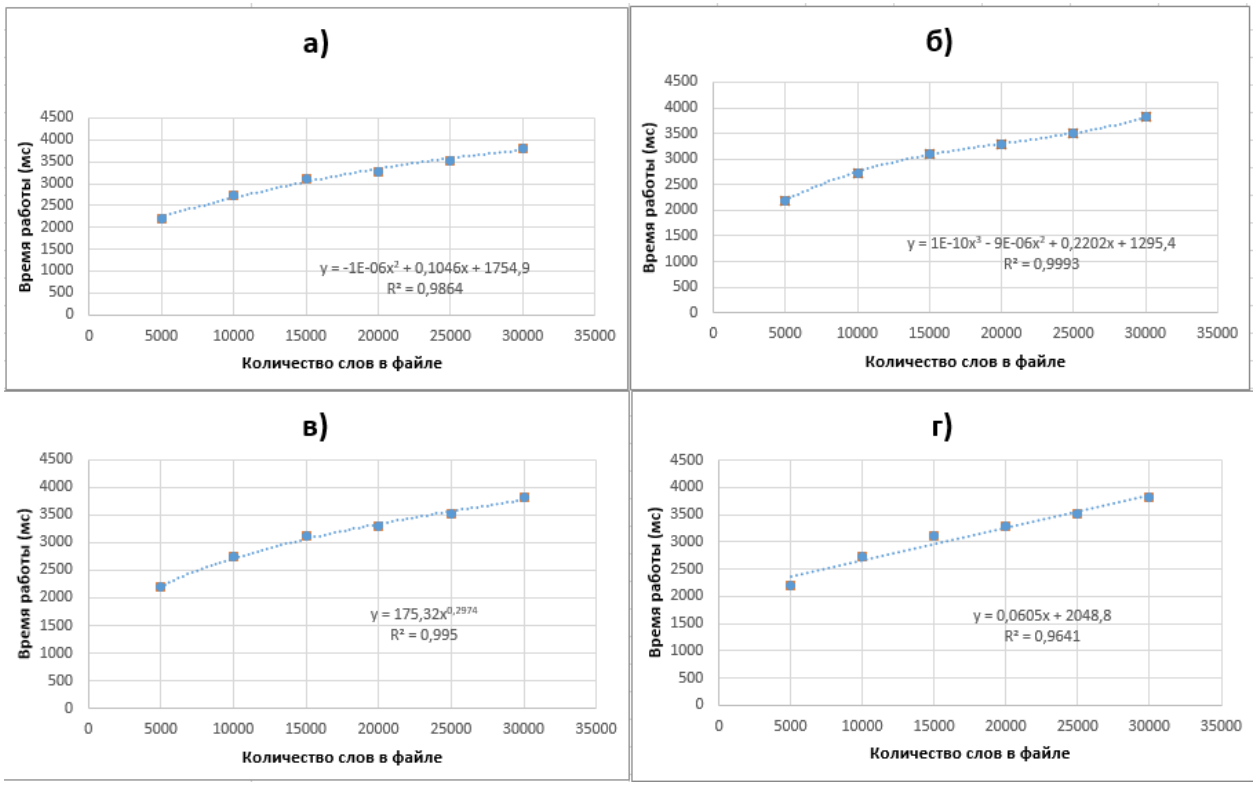


Рис. 4.1. Аппроксимация функции, представленной на графике 4.1. различными видами функциональных зависимостей: а) полиномиальная 2-й степени, б) полиномиальная 3-й степени, в) степенная, г) линейная.

.

Анализируя величину R^2 для каждой линии тренда, можно сделать вывод, что наиболее подходящей аппроксимирующей функцией является полином степени 3.

4.1.2. Зависимость от количества файлов

Цели эксперимента:

- Построить графические зависимости, отражающие среднее время кластеризации в зависимости от количества файлов;
- Определить аналитическую зависимость, отражающие среднее время кластеризации в зависимости от количества файлов.

Ход эксперимента:

Разместить N файлов в диапазоне $5 \leq N \leq 20$ с шагом 5. При каждом фиксированном N алгоритм отрабатывает 100 раз.

Таблица 4.3.

Зависимость времени кластеризации от количества файлов.

Кол-во файлов	Время работы (мс)
5	426,88
10	1309,14
15	3031,75
20	5926,5

График 4.2. Зависимость времени кластеризации от количества файлов.

Для оценки вида зависимости проведем аппроксимацию следующими видами функций:

а. Полиномиальная 2-й степени
б. Полиномиальная 3-й стпени
в. Степенная
г. Экспоненциальная

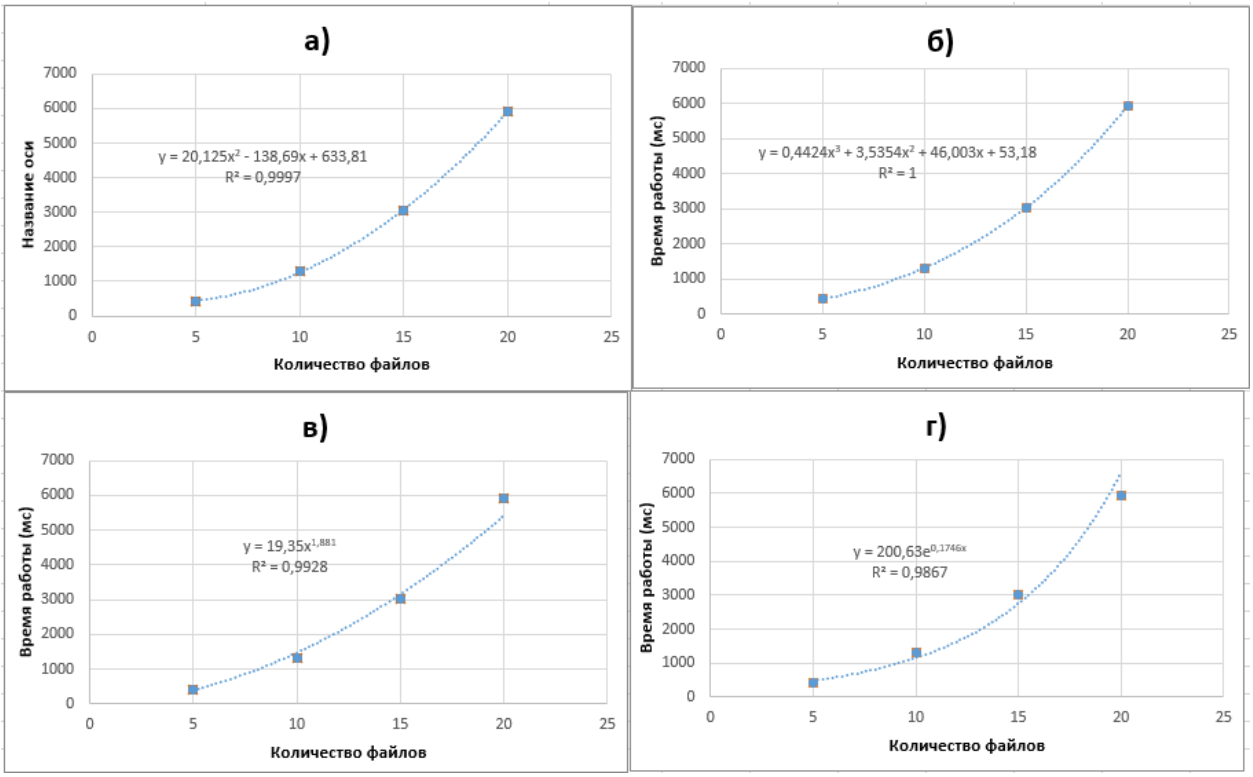


Рис. 4.2. Аппроксимация функции, представленной на графике 4.2. различными видами функциональных зависимостей: а) полиномиальная 2-й степени, б) полиномиальная 3-й степени, в) линейная, г) степенная.

Анализируя величину R^2 для каждой линии тренда, можно сделать вывод, что наиболее подходящей аппроксимирующей функцией является полином степени 3.

4.2. Оценка производительности алгоритма классификации

4.2.1. Зависимость от количества слов в файле

Цели эксперимента:

- Построить графические зависимости, отражающие среднее время классификации в зависимости от количества слов в файле;
- Определить аналитическую зависимость, отражающие среднее время классификации в зависимости от количества слов в файле.

Ход эксперимента:

Разместить N файлов, с количеством слов в диапазоне $5000 \leq N \leq 35000$ с шагом 5000. При каждом фиксированном N алгоритм отрабатывает 100 раз.

Таблица 4.4.

Зависимость времени классификации от количества слов в файле

Кол-во слов	Время работы (мс)
5000	298,92
10000	576,34
15000	854,73
20000	1166,2
25000	1422,53
30000	1690,57

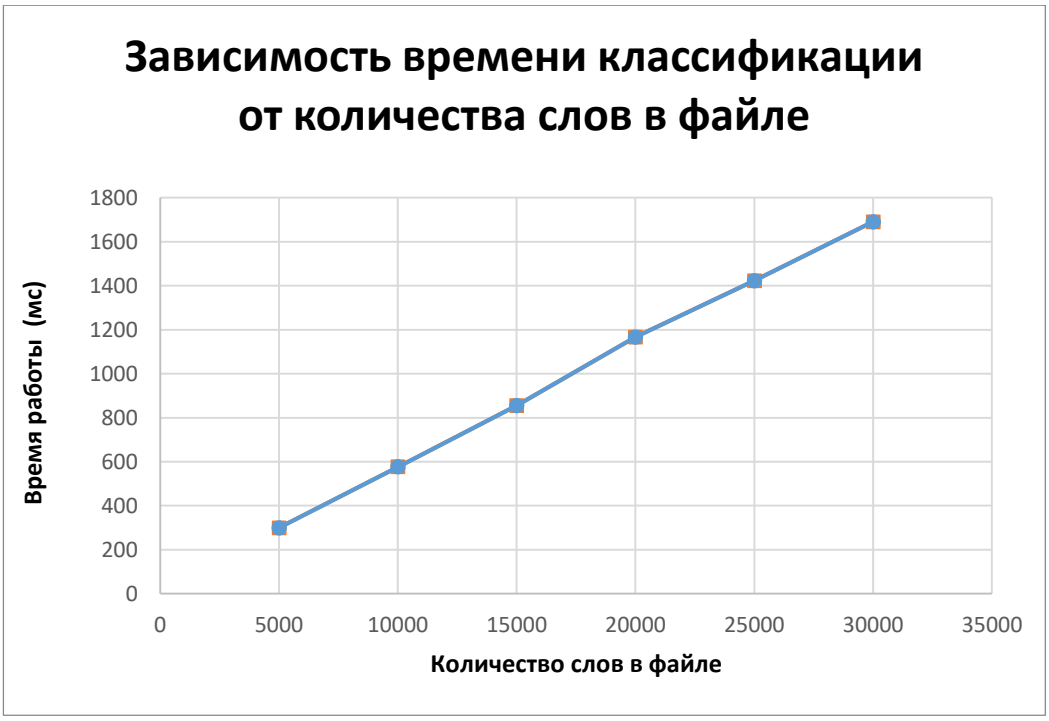


График 4.3. Зависимость времени кластеризации от количества файлов.

Для оценки вида зависимости проведем аппроксимацию следующими видами функций:

а. Полиномиальная 2-й степени
б. Полиномиальная 3-й стпени
в. Линейная
г. Степенная

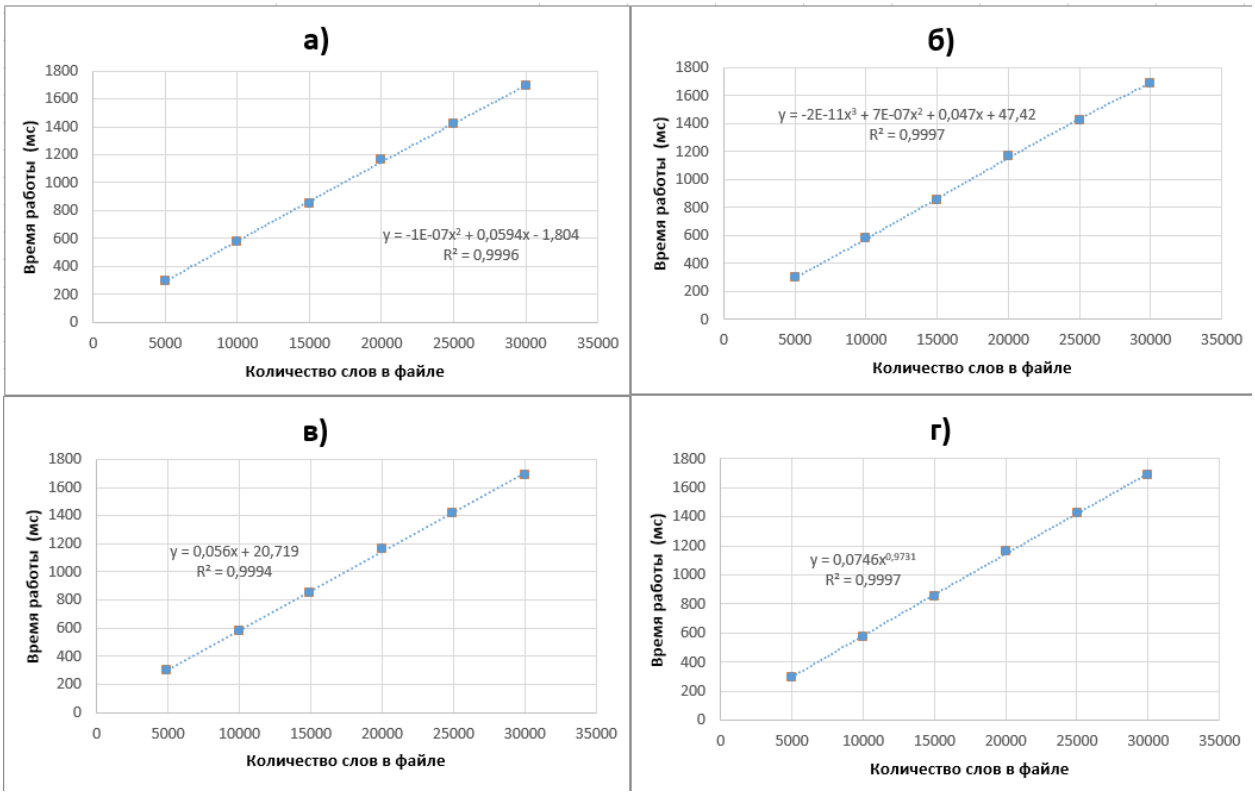


Рис. 4.3. Аппроксимация функции, представленной на графике 4.3. различными видами функциональных зависимостей: а) полиномиальная 2-й степени, б) полиномиальная 3-й степени, в) линейная, г) степенная.

Анализируя величину R^2 для каждой линии тренда, можно сделать вывод, что наиболее подходящей аппроксимирующей функцией является линейная функция.

4.2.2. Зависимость от количества файлов

Цели эксперимента:

– Построить графические зависимости, отражающие среднее время классификации в зависимости от количества файлов;
– Определить аналитическую зависимость, отражающие среднее время классификации в зависимости от количества файлов.

Ход эксперимента:

Разместить N файлов в диапазоне $5 \leq N \leq 20$ с шагом 5. При каждом фиксированном N алгоритм отрабатывает 100 раз.

Таблица 4.5.

Время классификации в зависимости от количества файлов

Кол-во файлов	Время работы (мс)
5	287,19
10	568,7
15	855,5
20	1139,44

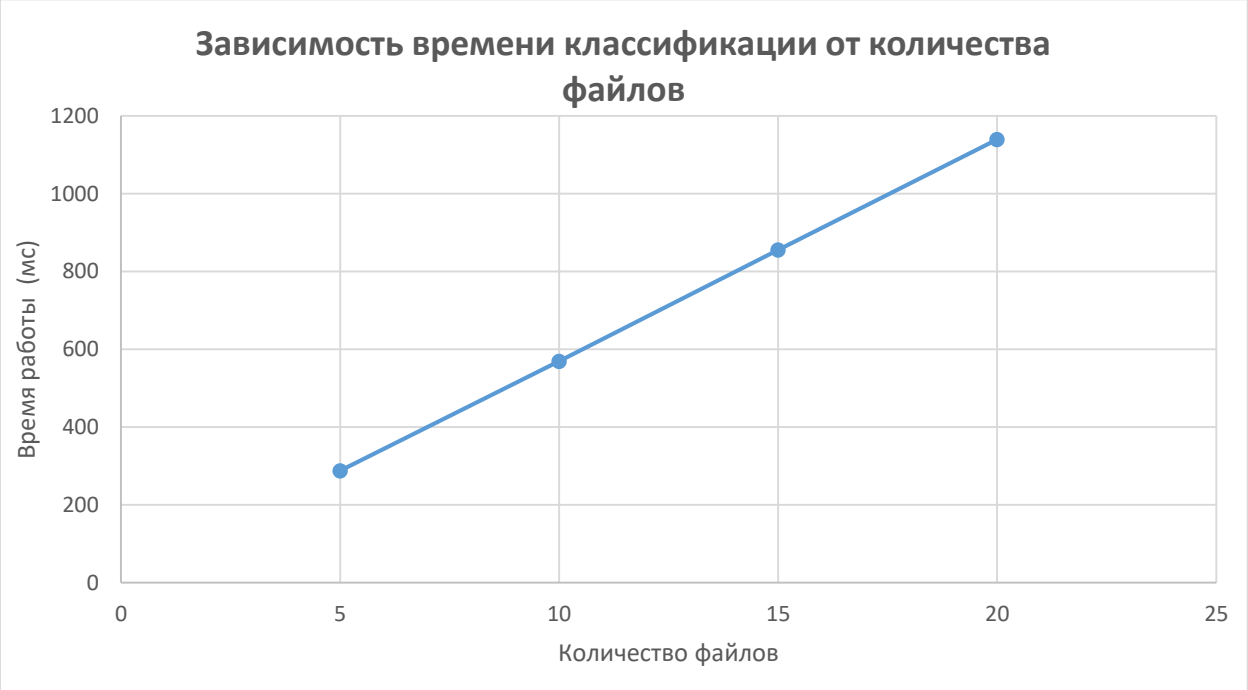


График 4.4. Зависимость времени классификации от количества файлов.

Для оценки вида зависимости проведем аппроксимацию следующими видами функций:

а. Полиномиальная 2-й степени
б. Полиномиальная 3-й ступени
в. Линейная
г. Степенная

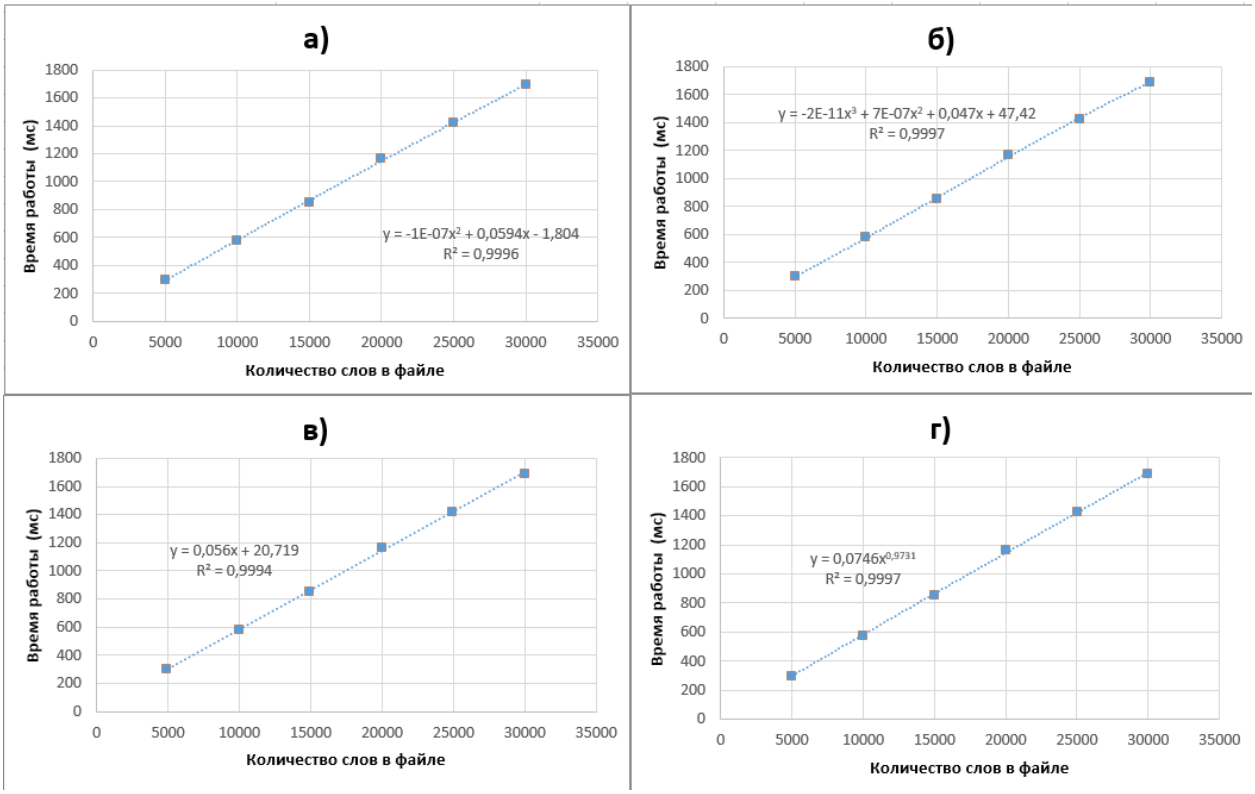


Рис. 4.4. Аппроксимация функции, представленной на графике 4.4. различными видами функциональных зависимостей: а) полиномиальная 2-й степени, б) полиномиальная 3-й степени, в) линейная, г) степенная.

Анализируя величину R^2 для каждой линии тренда, можно сделать вывод, что наиболее подходящей аппроксимирующей функцией является линейная функция.

4.3. Заключение

В главе были проведены различные эксперименты и построены графики соответствующих зависимостей.

ЗАКЛЮЧЕНИЕ

Данная выпускная работа посвящена кластеризации документов на базе методов машинного обучения. В рамках работы был проведен аналитический обзор, выбрана математическая модель, минимизирующая расстояние между элементами одной группы. Также разработан программный комплекс, реализующий мой алгоритм, проведена серия экспериментов, подтверждающая адекватность выбранной математической модели и эффективность программного комплекса.

Поставленные цели считаю достигнутыми, а задачи выполненными.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. http://tekhnosfera.com/razrabotka-metoda-avtomaticheskogo-formirovaniya-rubrikatora-polnotekstovyh-dokumentov
2. https://habr.com/ru/post/101338
3. https://javarush.ru/groups/posts/254-top-5-bibliotek-mashinnogo-obuchenija-dlja-java
4. Б. Дюран. Кластерный анализ. 128 с.
5. Воронцов К.В. Алгоритмы кластеризации и многомерного шкалирования. Курс лекций. МГУ, 2007.
6. Гитис Л. Х. Кластерный анализ в задачах классификации, оптимизации и прогнозирования. 104 с.
7. Информационно-аналитический ресурс, посвященный машинному обучению, распознаванию образов и интеллектуальному анализу данных — www.machinelearning.ru
8. Котов А., Красильников Н. Кластеризация данных. 2006.
9. Мандель И. Д. Кластерный анализ. — М.: Финансы и Статистика, 1988.
10. Пескова О.В. Разработка метода автоматического формирования рубрикатора полнотекстовых документов.
11. Прикладная статистика: классификация и снижение размерности. / С.А. Айвазян, В.М. Бухштабер, И.С. Енюков, Л.Д. Мешалкин — М.: Финансы и статистика, 1989.
12. Чубукова И.А. Курс лекций «Data Mining», Интернет-университет информационных технологий — www.intuit.ru/department/database/datamining

ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ

Application.java

```
package diplom;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import diplom.config.Config;
import diplom.gui.Frame;

import java.io.File;
import java.util.Arrays;
import java.util.List;

public class Application {

    public static Gson gson;
    public static Config config;
    public static File currentDir;
    public static boolean time;
    public static boolean debug;
    public static final int REPETITION_COUNT = 100;
    public static final List<String> SUPPORTED_FILE_EXTENSIONS =
Arrays.asList(".txt", ".docx");
    public static boolean experiments;

    public static void main(String[] args) throws Exception {
        gson = new GsonBuilder().setPrettyPrinting().create();
        config = Config.parse(args);
        currentDir = new File(System.getProperty("user.dir") + "/sets");
```

```
        time = false;
        debug = false;
        experiments = false;

        Frame.showForm();
    }

    public static boolean isSupportableFile(File file) {
        for (String s : SUPPORTED_FILE_EXTENSIONS) {
            if (file.getName().endsWith(s)) {
                return true;
            }
        }
        return false;
    }

}
```

Class.java

```
package diplom.classification;

import com.google.gson.annotations.SerializedName;

import java.util.Map;

public class Class {

    public String name;
```

```
    @SerializedName("terms_frequency")
    public Map<String, Integer> termsFrequency;

    @Override
    public String toString() {
        return "Class{" +
                "name='" + name + '\'' +
                ", termsFrequency=" + termsFrequency +
                '}';
    }
}
```

Classification.java

```
package diplom.classification;

import com.google.gson.reflect.TypeToken;
import diplom.Application;
import diplom.ExperimentsInterface;
import diplom.distance.Distance;
import diplom.nlp.FilteredUnigram;
import diplom.utils.Copy;
import diplom.utils.Reader;
import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;
import java.nio.charset.Charset;
import java.util.*;
import java.util.concurrent.atomic.AtomicReference;
```

```
import java.util.function.Function;
import java.util.stream.Collectors;

import static diplom.utils.Copy.SEPARATOR;

public class Classification implements ExperimentsInterface {

    private static final String OTHER_CLASS_NAME = "Прочее (!)";
    private Distance distance;
    private Map<String, double[]> dataMap;
    private List<String> vocabulary;
    private int CLASSES_END;
    private StringBuilder builder;

    private double initTime;
    private double workTime;

    public Classification(List<Class> classes, Distance distance) {
        initTime = System.nanoTime();

        this.distance = distance;
        this.dataMap = new LinkedHashMap<>();
        this.CLASSES_END = classes.size() + 1;
        this.builder = new StringBuilder();

        vocabulary = new ArrayList<>();

        for (Class c : classes) {
            String text = String.join(" ", c.termsFrequency.keySet());
            vocabulary.addAll(FilteredUnigram.get(text));
```

```
        }

        vocabulary = vocabulary.stream()
                .distinct()
                .sorted()
                .collect(Collectors.toList());

        System.out.printf("%10s - %s\n", "VOCABULARY", vocabulary);
        if (Application.debug)
            builder.append(String.format("%10s - %s\n", "VOCABULARY",
vocabulary));

        for (Class c : classes) {
            double[] dataRow = new double[vocabulary.size()];
            Arrays.fill(dataRow, 0.0);
            for (Map.Entry<String, Integer> entry : c.termsFrequency.entrySet()) {
                List<String> terms = FilteredUnigram.get(entry.getKey());
                for (String term : terms) {
                    int index = vocabulary.indexOf(term);
                    dataRow[index] = entry.getValue();
                }
            }
            dataMap.put(c.name, dataRow);
        }

        double[] dataRow = new double[vocabulary.size()];
        dataMap.put(OTHER_CLASS_NAME, dataRow);

        initTime = (System.nanoTime() - initTime) / 1000000;
```

```
        System.out.printf("ВРЕМЯ ИНИЦИАЛИЗАЦИИ (classify): %.2f мс\n",
initTime);
    }

    public static List<Class> parseClasses(File classesFile) throws IOException {
        String json = FileUtils.readFileToString(classesFile, Charset.defaultCharset());
        return Application.gson.fromJson(json, new TypeToken<List<Class>>() {
        }.getType());
    }

    public Map<String, List<String>> run(File[] path) throws IOException {
        workTime = System.nanoTime();

        List<File> rawFiles = Arrays.stream(path).collect(Collectors.toList());

        addFilesToMatrix(rawFiles);

        normalize();

        Map<String, List<String>> result = new TreeMap<>();
        List<String> otherFiles = new ArrayList<>();
        dataMap.entrySet().stream().skip(CLASSES_END).forEach(fileRow -> {
            AtomicReference<String> className = new AtomicReference<>("");
            AtomicReference<Double> dist = new AtomicReference<>((double) 0);
            AtomicReference<String> fileName = new AtomicReference<>("");

            dataMap.entrySet().stream().limit(CLASSES_END).forEach(classRow -> {
                double d = distance.calc(fileRow.getValue(), classRow.getValue());
                if (d > dist.get()) {
                    className.set(classRow.getKey());
```

```
                dist.set(d);
                fileName.set(fileRow.getKey());
            }
        });

        if (dist.get() == 0) {
            otherFiles.add(fileRow.getKey());
        } else {
            String cName = className.get();
            if (!result.containsKey(cName))
                result.put(cName, new ArrayList<String>() {{
                    add(fileName.get());
                }});
            else
                result.get(className.get()).add(fileName.get());

        }
    });

    result.put(OTHER_CLASS_NAME, otherFiles);

    if (Application.debug)
        builder.append("\nКЛАССИФИЦИРОВАННЫЕ ФАЙЛЫ:\n");
    else
        builder.append("КЛАССИФИЦИРОВАННЫЕ ФАЙЛЫ:\n");
    for (Map.Entry<String, List<String>> entry : result.entrySet()) {
        System.out.printf("%10s - %s\n", entry.getKey(), entry.getValue());
        builder.append(String.format("%s - %s\n", entry.getKey(),
entry.getValue()));
    }
```

```
        String dir = "sets" + SEPARATOR + "classification";
        Copy.group(dir, path, result);

        workTime = (System.nanoTime() - workTime) / 1000000;
        System.out.printf("ВРЕМЯ РАБОТЫ (classify): %.2f мс\n", workTime);
        if (Application.time)
            builder.append("\nВРЕМЯ РАБОТЫ: ").append(String.format("%.2f мс",
workTime));

        return result;
    }

    private void addFilesToMatrix(List<File> files) throws IOException {
        for (File f : files) {
            String text = Reader.readFile(f);
            List<String> strings = FilteredUnigram.get(text);

            double[] dataRow = new double[vocabulary.size()];
            for (Map.Entry<String, Long> term : strings.stream()
                    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()))
                    .entrySet()) {
                int index = vocabulary.indexOf(term.getKey());
                if (index > 0) // на случай, если в словаре нет такого термина
                    dataRow[index] = term.getValue();
            }
            dataMap.put(f.getName(), dataRow);
        }
```

```
        System.out.println("\nЧАСТОТЫ:");
        if (Application.debug)
            builder.append("\nЧАСТОТЫ:\n");
        for (Map.Entry<String, double[]> e : dataMap.entrySet()) {
            System.out.printf("%10s - %s\n", e.getKey(),
Arrays.toString(e.getValue()));
            if (Application.debug)
                builder.append(String.format("%10s - %s\n", e.getKey(),
Arrays.toString(e.getValue())));
        }
    }

    private void normalize() {
        for (Map.Entry<String, double[]> row : dataMap.entrySet()) {
            double[] items = row.getValue();
            double min = Arrays.stream(items).min().getAsDouble();
            double max = Arrays.stream(items).max().getAsDouble();
            for (int i = 0; i < items.length; i++) {
                items[i] = (items[i] - min) / (max - min);
                if (Double.isNaN(items[i])) {
                    items[i] = 0;
                }
            }
            dataMap.put(row.getKey(), items);
        }

        System.out.println("\nНОРМАЛИЗОВАННЫЕ ДАННЫЕ:");
        if (Application.debug)
            builder.append("\nНОРМАЛИЗОВАННЫЕ ДАННЫЕ:\n");
        for (Map.Entry<String, double[]> e : dataMap.entrySet()) {
```

```
            System.out.printf("%10s - %s\n", e.getKey(),
Arrays.toString(e.getValue()));
            if (Application.debug)
                builder.append(String.format("%10s - %s\n", e.getKey(),
Arrays.toString(e.getValue())));
        }
    }

    public String getBuilder() {
        return builder.toString();
    }

    public double getInitTime() {
        return initTime;
    }

    public double getWorkTime() {
        return workTime;
    }

}
```

Clustering.java

```
package diplom.clustering;

import diplom.Application;
import diplom.ExperimentsInterface;
import diplom.distance.Distance;
import diplom.utils.Copy;
```

```
import java.io.File;
import java.util.*;
import java.util.stream.Collectors;

import static diplom.utils.Copy.SEPARATOR;

public class Clustering implements ExperimentsInterface {

    private double separateValue;
    private Distance distance;

    private Vocabulary vocabulary;
    private List<Document> documents;
    private List<String> docNames;

    private double[][] similarityMatrix;

    private StringBuilder builder;

    private double initTime;
    private double workTime;

    public Clustering(double separateValue, Distance distance) {
        initTime = System.nanoTime();

        this.separateValue = separateValue;
        this.distance = distance;
        this.builder = new StringBuilder();
```

```
        this.vocabulary = new Vocabulary();
        this.docNames = new ArrayList<>();
        this.documents = new ArrayList<>();

        initTime = (System.nanoTime() - initTime) / 1000000;
        System.out.printf("ВРЕМЯ ИНИЦИАЛИЗАЦИИ (cluster): %.2f мс\n",
initTime);
    }

    public Map<String, List<String>> run(File[] files) throws Exception {
        workTime = System.nanoTime();

        builder.setLength(0);

        if (files == null || files.length == 0) {
            throw new Exception("docs == null || docs.length == 0");
        }

        for (File file : files) {
            documents.add(new Document(file, vocabulary));
            docNames.add(file.getName());
        }

        this.similarityMatrix = new double[files.length][files.length];
        documents.forEach(Document::calcTermsFrequency);

        initSimMatrix();

        //normalize();
```

```
List<Prim.Edge> edges = Prim.solve(similarityMatrix);

if (Application.debug)
    builder.append(docNames).append("\n");
System.out.println(docNames + "\n");

for (double[] matrix : similarityMatrix) {
    for (int j = 0; j < similarityMatrix.length; j++) {
        System.out.printf("%.2f  ", matrix[j]);
        if (Application.debug)
            builder.append(String.format("%.2f  ", matrix[j]));
    }
    if (Application.debug)
        builder.append("\n");
    System.out.println();
}

if (Application.debug) {
    builder.append("\n");
    edges.forEach(e -> builder.append(e.toString()).append("\n"));
    builder.append("\n");
}

System.out.println();
edges.forEach(System.out::println);
System.out.println();

List<List<Integer>> clusters = new ArrayList<>();
clusters.add(new ArrayList<>(Collections.singletonList(edges.get(0).s)));
```

```
        for (Prim.Edge edge : edges) {
            if (edge.weigh >= separateValue) {
                clusters.add(new ArrayList<>(Collections.singletonList(edge.t)));
            } else {
                int clusterID = relevantCluster(clusters, edge);
                clusters.get(clusterID).add(edge.t);
            }
        }

//      List<List<String>> collect = clusters.stream()
//              .map(docIDs -> docIDs.stream()
//                      .map(docID -> docNames.get(docID))
//                      .collect(Collectors.toList()))
//              .collect(Collectors.toList());

        Map<String, List<String>> collect = clusters.stream()
                .map(docIDs -> docIDs.stream()
                        .map(docID -> docNames.get(docID))
                        .collect(Collectors.toList()))
                .collect(Collectors.toMap(
                        this::getDocsKeyWords,
                        i -> i
                ));

        int i = 1;
        for (Map.Entry<String, List<String>> entry : collect.entrySet()) {
//          builder.append(String.format("Группа №%d: %s\n", i++, entry.getKey()));
            builder.append(String.format("Группа №%d:\n", i++));
            builder.append(String.format("Файлы: %s\n\n", entry.getValue()));
```

```
//                  builder.append(String.format("Файлы, принадлежащие группе №%d:\n%s\n\n", i++, entry.getValue()));
        }

        String dir = "sets" + SEPARATOR + "clustering";
        Copy.group(dir, files, collect, true);

        workTime = (System.nanoTime() - workTime) / 1000000;
        System.out.printf("ВРЕМЯ РАБОТЫ (cluster): %.2f мс\n", workTime);
        if (Application.time)
            builder.append("ВРЕМЯ РАБОТЫ: ").append(String.format("%.2f мс", workTime));

        return collect;
    }

    public String getBuilder() {
        return builder.toString();
    }

    private int relevantCluster(List<List<Integer>> clusters, Prim.Edge edge) {
        for (int i = 0; i < clusters.size(); i++) {
            if (clusters.get(i).contains(edge.s))
                return i;
        }
        return -1;
    }

    private void initSimMatrix() {
        for (int i = 0; i < documents.size(); i++) {
```

```
            for (int j = 0; j < documents.size(); j++) {
                similarityMatrix[i][j] = 1 - distance.calc(tfidf(documents.get(i)),
tfidf(documents.get(j)));
            }
        }
    }

    private double[] tfidf(Document doc) {
        double[] tf = tf(doc);
        double[] idf = idf(doc);
        double[] tfidf = new double[vocabulary.size()];

        for (int i = 0; i < vocabulary.size(); i++) {
            tfidf[i] = tf[i] * idf[i];
        }

        return tfidf;
    }

    private double[] tf(Document doc) {
        double[] tf = new double[vocabulary.size()];
        double termsCount = doc.termsCount();
        doc.termsFrequencyMap().forEach((k, v) -> tf[k] = v / termsCount);
        return tf;
    }

    private double[] idf(Document doc) {
        double[] idf = new double[vocabulary.size()];
        doc.termsFrequencyMap().forEach((k, v) -> idf[k] =
Math.log(documents.size() / (double) docsWithWord(k)));
```

```
        return idf;
    }

    private int docsWithWord(int termID) {
//        int count = 0;
//        for (Document doc : documents) {
//            if (doc.containsWord(termID))
//                count++;
//        }
//        return count;
        return (int) documents.stream()
                .filter(d -> d.containsWord(termID))
                .count();
    }

    private String getDocsKeyWords(List<String> docsName) {
        return docsName.stream()
                .limit(3)
                .map(name -> documents.get(docNames.indexOf(name))
                        .getTheHardestWord())
                .distinct()
                .collect(Collectors.joining(", "));
    }

    public double getInitTime() {
        return initTime;
    }

    public double getWorkTime() {
        return workTime;
```

```
    }

    private void normalize() {
        for (int i = 0; i < similarityMatrix.length; i++) {
            double min = Arrays.stream(similarityMatrix[i]).min().getAsDouble();
            double max = Arrays.stream(similarityMatrix[i]).max().getAsDouble();

            for (int j = 0; j < similarityMatrix[i].length; j++) {
                similarityMatrix[i][j] = (similarityMatrix[i][j] - min) / (max - min);
                if (Double.isNaN(similarityMatrix[i][j])) {
                    similarityMatrix[i][j] = 0;
                }
            }
        }

        System.out.println("НОРМАЛИЗОВАННЫЕ ДАННЫЕ:");
        if (Application.debug)
            builder.append("НОРМАЛИЗОВАННЫЕ ДАННЫЕ:\n");
        for (double[] matrix : similarityMatrix) {
            System.out.printf("%s\n", Arrays.toString(matrix));
            if (Application.debug)
                builder.append(String.format("%s\n", Arrays.toString(matrix)));
        }

        System.out.println();
        if (Application.debug)
            builder.append("\n");
    }
}
```

Document.java

```
package diplom.clustering;

import diplom.nlp.FilteredUnigram;
import diplom.utils.Reader;

import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

public class Document {

    private Vocabulary vocabulary;
    private List<String> terms;
    private Map<Integer, Integer> termsFrequency;
    private int termWithMaxFrequency;

    public Document(File doc, Vocabulary vocabulary) {
        this.vocabulary = vocabulary;
        this.termsFrequency = new TreeMap<>();
        this.termWithMaxFrequency = 0;

        try {
            String text = Reader.readFile(doc);
            this.terms = FilteredUnigram.get(text);
            vocabulary.addAll(terms);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    public void calcTermsFrequency() {
        for (String term : vocabulary.keySet()) {
            termsFrequency.put(vocabulary.get(term), 0);
        }

        int termID;
        for (String term : terms) {
            termID = vocabulary.get(term);
            Integer termsFreq = termsFrequency.get(termID);
            termsFrequency.put(termID, termsFreq + 1);

            if (termsFrequency.get(termWithMaxFrequency) < termsFreq) {
                termWithMaxFrequency = termID;
            }
        }
    }

    public int termsCount() {
        return terms.size();
    }

    public Map<Integer, Integer> termsFrequencyMap() {
        return termsFrequency;
    }

    public boolean containsWord(int termID) {
```

```
        return termsFrequency.get(termID) > 0;
    }

    public String getTheHardestWord() {
        return terms.get(termsFrequency.get(termWithMaxFrequency));
    }

}
```

Prim.java

```
package diplom.clustering;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Prim {

    public static List<Edge> solve(double[][] simMatrix) {
        List<Edge> edges = new ArrayList<>();

        int no_edge = 0;
        boolean[] selected = new boolean[simMatrix.length];
        Arrays.fill(selected, false);
        selected[0] = true;

        int x, y;
        double min;
        while (no_edge < simMatrix.length - 1) {
```

```
        x = y = 0;
        min = 1_000_000;

        for (int i = 0; i < simMatrix.length; i++) {
            if (selected[i]) {
                for (int j = 0; j < simMatrix.length; j++) {
                    if (!selected[j] && simMatrix[i][j] != 0) {
                        if (min > simMatrix[i][j]) {
                            min = simMatrix[i][j];
                            x = i;
                            y = j;
                        }

                    }
                }
            }
        }

        edges.add(new Edge(x, y, simMatrix[x][y]));
        selected[y] = true;
        no_edge++;
    }

    return edges;
}

public static class Edge {
    int s, t;
    double weigh;
```

```
        public Edge(int s, int t, double weigh) {
            this.s = s;
            this.t = t;
            this.weigh = weigh;
        }

        @Override
        public String toString() {
            return (s + 1) + " - " + (t + 1) + " : " + String.format("%.2f", weigh);
        }
    }

}
```

Vocabulary.java

```
package diplom.clustering;

import java.util.List;
import java.util.TreeMap;

public class Vocabulary extends TreeMap<String, Integer> {

    private int nextIndex = 0;

    public void add(String word) {
        if (!this.containsKey(word)) {
            this.put(word, nextIndex++);
        }
    }
```

```
    public void addAll(List<String> words) {
        for (String word : words) {
            this.add(word);
        }
    }

}
```

Distance.java

```
package diplom.distance;

public interface Distance {

    static Distance distance(String similarityMeasure) {
        switch (similarityMeasure) {
            case "cos":
                return new CosineDistance();
            case "euclid":
                return new EuclideanDistance();
        }

        return new CosineDistance();
    }

    double calc(double[] vector1, double[] vector2);

}
```

EuclideanDistance.java

```
package diplom.distance;

public class EuclideanDistance implements Distance {

    public double calc(double[] vector1, double[] vector2) {
        double sum = 0;
        for (int i = 0; i < vector1.length; i++) {
            sum += Math.pow(vector1[i] - vector2[i], 2);
        }
        return Math.sqrt(sum);
    }

}
```