МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное   
учреждение высшего образования   
«Южный федеральный университет»

Институт математики, механики   
и компьютерных наук им. И. И. Воровича

Николаев Андрей Валерьевич

РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ ДЛЯ СУММАРИЗАЦИИ ТЕКСТОВ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
по направлению подготовки  
02.04.02 – Фундаментальная информатика и информационные технологии,   
направленность программы  
«Разработка мобильных приложений и компьютерных игр»

**Научный руководитель** –   
доц., к. ф.-м. н. Абрамян Анна Владимировна

**Рецензент** –   
доц., к. т. н. Демяненко Яна Михайловна

Допущено к защите:  
руководитель   
образовательной программы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Демяненко Я. М.

Ростов-на-Дону – 2024

Оглавление

[Введение 3](#_Toc168254243)

[1. Постановка задачи 5](#_Toc168254244)

[2. Анализ существующих решений 6](#_Toc168254245)

[3. Разработка мобильного приложения 9](#_Toc168254246)

[3.1. Используемые инструменты и технологии 9](#_Toc168254247)

[3.2. Архитектура приложения 11](#_Toc168254248)

[3.3. UI/UX дизайн приложения 17](#_Toc168254249)

[3.4. Навигация в приложении 19](#_Toc168254250)

[3.5. Главный экран 22](#_Toc168254251)

[3.6. Экран сканирования документа 30](#_Toc168254252)

[3.7. Экран суммаризации 37](#_Toc168254253)

[4. Суммаризация текстов как задача NLP 40](#_Toc168254254)

[4.1. Экстрактивная суммаризация 41](#_Toc168254255)

[4.2. Абстрактивная суммаризация 44](#_Toc168254256)

[4.3. Методы оценки качества суммаризации 47](#_Toc168254257)

[5. Интеграция моделей машинного обучения 50](#_Toc168254258)

[Заключение 53](#_Toc168254259)

[Литература 54](#_Toc168254260)

[Приложение 1. Результаты работы алгоритмов суммаризации 56](#_Toc168254261)

# Введение

В мире существует огромное количество данных различного рода. Их нужно обрабатывать, систематизировать и анализировать. Достаточно большой процент всей информации представлен в виде текстовых данных. Естественно, при работе с такими объёмами информации, хочется сократить время, затраченное на изучение того или иного ресурса. И именно эту проблему решает суммаризация.

Автоматическая суммаризация генерирует «краткий пересказ», содержащий наиболее важные предложения, и включающий всю важную релевантную информацию из исходного документа. Таким образом, количество обрабатываемой информации сокращается, при этом не теряется первоначальный смысл документа.

В рамках данной работы рассмотрен процесс разработки Android-приложения для работы с суммаризацией текстов. Для создания приложения выбран нативный подход с использованием языка Kotlin в сочетании с принципами чистой архитектуры [1] и паттерном MVVM. Интерфейс приложения разработан с использованием декларативного подхода Jetpack Compose. Приложение, помимо основной функции суммаризации текста с возможностями настройки параметров суммаризации, позволяет просматривать историю обработанных текстов, а также сканировать документы в печатном виде с помощью камеры устройства с применением Google MLKit.

Кроме мобильного приложения в ходе исследования были разработаны алгоритмы суммаризации 2 типов, основанные на использовании методов машинного обучения:

* Экстрактивная суммаризация, составляющая сокращённый текст из фрагментов оригинального. Данный подход основан на применении кластерного анализа с использованием векторизации при помощи нейронных сетей.
* Абстрактивная суммаризация, генерирующая с нуля сокращённую версию исходного текста. В основе этого подхода лежит использование генеративных нейросетей, в частности, предобученной модели mT5. Обучение модели проводилось с применением трансферного обучения на датасете новостей изданий DailyMail, CNN и Газета.ру.

Для использования разработанных алгоритмов суммаризации были рассмотрены возможные методы интеграции моделей машинного обучения и описан подход с внедрением клиент-серверной архитектуры. Было создано серверное приложение на языке Python с использованием фреймворка FastAPI и библиотеки Uvicorn.

# Постановка задачи

Разработать мобильное приложение для суммаризации текстов с применением методов машинного обучения. Для решения поставленной задачи выполнить следующие этапы исследования:

1. Проанализировать различные подходы к суммаризации текстов и обучить модели машинного обучения для решения поставленной задачи.
2. Разработать мобильное приложение, основной функционал которого направлен на получение текстовых входных данных и дальнейшую их обработку с помощью разработанных алгоритмов суммаризации.
3. Интегрировать в приложение разработанные модели машинного обучения для автоматической обработки текста.

# Анализ существующих решений

Задача суммаризации текстов является достаточно популярной как среди исследователей области обработки естественного языка, так и среди обычных пользователей, и поэтому существует множество решений, которые в том или ином виде выделяют основную мысль в текстах.

Наиболее заметным достижением последних лет в сфере Natural Language Processing стали трансформеры, на основе которых построено огромное множество моделей. Эти модели достаточно мощны, чтобы решить задачу суммаризации с результатом, близким к идеалу. Поэтому для сокращения текста люди часто обращаются к сервисам, предоставляющим доступ к таким моделям, например, ChatGPT. Данный инструмент, помимо всех остальных потенциальных сфер его применения, отлично справляется с задачей абстрактивной суммаризации. Тем не менее, так как это сервис с широким функционалом, нужно четко сформулировать запрос к модели (его называют промпт), чтобы получить корректный ответ. Кроме того, ChatGPT трудно использовать для получения пересказа с использованием экстрактивной суммаризации. Любой ответ генерируется, и модель вследствие этого может изменить исходный текст, даже если в промпте указать необходимость сохранения исходной структуры.

Помимо языковых сервисов, использующих языковые модели широкого профиля, существуют ресурсы, заточенные только под задачу суммаризации. Одним из таких ресурсов является сайт SMMRY [2]. Он имеет простой интерфейс (см. рис. 2.1) и довольно гибкую настройку параметров суммаризации. Так, например, сервис позволяет настроить количество генерируемых предложений. Кроме того, вместо текста на вход можно предоставить текстовый файл, либо ссылку на сторонний ресурс, и сервис автоматически скопирует текст из этих источников и продолжит его обработку. Минусом данного ресурса является отсутствие возможности выбрать тип суммаризации, и все тексты по умолчанию генерируются, то есть используется абстрактивный подход без предоставления пользователю возможности выбрать экстрактивный. Кроме того, сервис не предоставляет функционала считывания текста с изображений, что может быть удобным при необходимости получить краткое содержание текста, к цифровому виду которого невозможно получить доступ.

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рис. 2.1. Интерфейс веб-сервиса SMMRY

В официальном магазине приложений для операционной системы Android, Google Play, по запросу «text summarizer» найдено 9 результатов. Самое скачиваемое из них, TLDR Summarize [3], имеет оценку 2.1 звезды из 5 возможных. Причиной такой низкой оценки является использование только экстрактивной суммаризации без предупреждения пользователей, ожидающих получить сокращенный текст другими словами, то есть с помощью абстрактивного подхода.

Несмотря на достаточно негативные отзывы, если рассматривать данное приложение как экстрактивный суммаризатор, оно справляется с задачей и предоставляет множество дополнительных функций (например, получение текста из PDF-документа, или по ссылке на веб-ресурс, или выделение ключевых слов в тексте после его суммаризации). Основной функционал приложения отображён на рис 2.2. В приложении достаточно много платных функций и рекламы. Некоторые функции, например сканирование документа камерой устройства, не работают, как предполагается (открывая вместо камеры интерфейс для выбора фото из галереи).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описаниеИзображение выглядит как текст, электроника, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 2.2. Основной функционал приложения TLDR Summarize

# Разработка мобильного приложения

Мобильное приложение было разработано для операционной системы Android. Согласно поставленной задаче, были выделены основные функции приложения, на основе которых были созданы 3 экрана:

* основной экран, на котором пользователь может ввести или вставить текст, который он хочет сократить и выбрать тип суммаризации, а также ознакомиться с историей уже обработанных текстов;
* экран сканирования текста, на котором с помощью камеры устройства можно получить оцифрованную версию текста некоторого печатного документа:
* экран суммаризации, на котором происходит непосредственно обработка входных данных с помощью нейросетей, а также предпросмотр результата сокращения исходного текста.

## Используемые инструменты и технологии

Для разработки приложения был выбран нативный подход с использованием языка программирования Kotlin. Несмотря на популярность кроссплатформенных фреймворков, таких как Flutter, они не очень гибки, и зачастую, если задача выходит за рамки простого клиент-серверного приложения, нативный подход является оптимальным решением. Так, в ходе анализа возможностей решения поставленной задачи было установлено, что для работы с нейросетями и камерой возможности кроссплатформенных решений (в частности, Flutter и Kotlin Multiplatform Mobile) весьма ограничены, и пришлось бы прибегнуть к внедрению нативного кода. Кроме того, сообщество разработчиков для нативного подхода намного больше, чем для любого кроссплатформенного, что означает большее количество релевантной информации, необходимой для решения поставленной задачи.

При разработке мобильного приложения основным ориентиром для выбора инструментов являлись последние рекомендации Google для разработки под ОС Android [4]. Так, языком программирования, использовавшимся при разработке мобильного приложения, является Kotlin, IDE для написания кода — Android Studio, а подход для построения пользовательского интерфейса — Jetpack Compose. Целевой SDK для приложения выбрана наиболее актуальная версия на момент разработки — API 34 (Upside Down Cake), в то время как минимально поддерживаемой версией Android является версия API 26 (Oreo). Согласно статистическим данным о распределении версий Android [5], таким образом приложение будет функционировать на 95.1% устройств с ОС Android.

Для упрощения различных аспектов разработки приложения были использованы сторонние библиотеки. Для работы с локальной базой данных используется библиотека Room, которая является надстройкой над SQLite. Она предоставляет удобный интерфейс для работы с базой данных, при этом используя всю мощь SQLite. Была использована и другая надстройка, но уже для внедрения зависимостей — Hilt. Она сокращает количество шаблонного кода, количество которого растёт вместе с количеством зависимостей. Как и Room, Hilt является упрощённой версией другой библиотеки — Dagger, сохраняя при этом все основные функции, необходимые при разработке. Для работы с обработкой изображений была использована библиотека MLKit от Google, собравшая в себе наиболее эффективные алгоритмы по обработке данных разного рода с помощью методов машинного обучения. Библиотека, являющаяся частью расширенной версии SDK, CameraX, использовалась для работы с камерой, а Jetpack Navigation, входящая в группу библиотек Jetpack, использовалась для навигации между экранами приложения. Для создания анимаций было использовано уже ставшее эталонным решение — Lottie от команды AirBnb.

Для тестирования разработанного решения были использованы как физические устройства, так и эмуляторы. Так, приложение было протестировано на смартфонах Samsung SM-S911B/DS (Android 14) и SM‑G973F (Android 10), а также на эмуляторах Google Pixel с версиями Android 12 и 14.

## Архитектура приложения

Архитектура приложения является первым и одним из самых важных шагов, с которого должна начинаться разработка любого решения. Это чёткий план, который описывает, на какие структурные компоненты делится приложение, как эти компоненты должны быть организованы и как они взаимодействуют друг с другом. Этот план также подразумевает наложение некоторых ограничений и правил, которые следует соблюдать, но в то же время благодаря этому в итоге приложение становится более поддерживаемым, расширяемым и тестируемым.

Долгое время для мобильных приложений, разрабатываемых для операционной системы Android, не было единого подхода к архитектуре. Большинство приложений основывались на собственных решениях (что порождало огромное количество шаблонного кода), либо же вообще писались без какой-либо архитектуры. В 2017 году на конференции для разработчиков Google I/O впервые была предложена единая концепция архитектуры для Android-приложений.

В основе предложенной концепции набор принципов, которые не являются непреложными правилами, но их соблюдение ведёт к построению более надёжного и чистого решения. Кроме того, без соблюдения этих концепций, построенная архитектура приложения будет уязвима к неявным зависимостям, что порождает трудно отлаживаемые ошибки.

Первым из этих принципов является принцип разделения ответственности [6], который гласит, что код нужно разделять на блоки, как можно меньше пересекающиеся по функционалу. Достаточно частая ошибка разработчиков, которую допускают, в основном, начинающие программисты, это писать весь код внутри активностей или фрагментов. Это плохой подход, потому что разработчик не владеет реализацией этих классов, и они, в своё время, являются «мостом» между системой Android и разрабатываемым приложением и обладают собственным жизненным циклом. Операционная система может уничтожить их в любой момент по разным причинам, и, если вся логика приложения содержится в них, это может привести к непредсказуемым ошибкам и нестабильной работе приложения. Лучшей практикой в этом случае является вынесение как можно большего количества логики, оставив только код, отвечающий за отображение данных на экране устройства и обработку событий, возникающих при работе приложения.

Второй принцип, принцип единого источника истины (Single source of truth), ограничивает количество источников данных определённого типа всего одним, централизующим все изменения. Чтобы достичь этого, источник должен раскрывать данные в неизменяемом виде, а чтобы их изменить, он должен содержать методы-события, которые могут вызываться из других мест слоя данных. Этот подход обладает рядом преимуществ:

* Все изменения, вносимые в данные проходят через одни и те же методы доступа к данным, становятся более отслеживаемыми, что упрощает поиск ошибок.
* Данные защищены от внешнего воздействия, и другие типы не могут случайно или намеренно их исказить.
* Данные, полученные в виде потока (flow), или любым другим способом, не могут быть изменены, что устраняет возможность случайной порчи данных.

Этот принцип используется вместе с паттерном однонаправленных потоков данных (Unidirectional Data Flow), который заключается в установлении правил «течения» потоков данных: состояния (неизменяемые сущности из единого источника истины) передаются только в одном направлении (от слоя Data к слою UI, от корня дерева компонентов, до компонента, отображающего этот вид данных), в то время как события, изменяющие данные, передаются только в противоположном направлении. Этот паттерн гарантирует целостность данных, делает потоки данных менее подверженными ошибкам, связанным с их случайным изменением и порчей, а также является легко отлаживаемым.

На основе этих принципов построена архитектура, рекомендуемая Google при создании мобильных приложений для ОС Android. Она состоит из 2 основных частей: UI-слой (слой пользовательского интерфейса), который отвечает за отображение данных, а также Data-слой (слой данных), отвечающий за получение и хранение данных, а также, при необходимости, обрабатывающий данные. Кроме того, в этой схеме существует опциональный Domain-слой (доменный слой). В него обычно выносят сложную бизнес-логику приложения. Описанная схема представлена на рис. 3.1.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рис. 3.1. Схема архитектуры Android приложения

Data слой, или слой данных, содержит в себе данные приложения, источники для их получения, а также бизнес-логику — правила хранения, обработки и изменения данных. Благодаря соблюдению принципа разделения ответственности разные части этого слоя можно переиспользовать в различных частях приложения, либо же тестировать логику без привязки к UI.

Этот слой состоит из репозиториев, каждый из которых может содержать один или несколько источников данных. Каждый отдельный репозиторий отвечает за свой отдельный тип данных, и их зоны ответственности не должны пересекаться. Так, в разработанном приложении присутствуют 2 репозитория: отвечающий за работу с кэшем, и отвечающий за работу с историей обработанных текстов. Репозитории берут на себя большую часть ответственности всего слоя данных, в частности:

* централизация изменений данных;
* работа с бизнес-логикой;
* объединение потоков данных из нескольких источников;
* предоставление данных другим слоям

Источники данных, в свою очередь — это интерфейс для доступа к разного рода системам хранения и получения данных, таким как базы данных, файлы кэша, либо API. Только репозитории могут обращаться к источникам данных, и они не должны использовать ни в одной другой части приложения. Кроме того, источники никак не должны обрабатывать получаемые данные. Их задача — получить данные по запросу и вернуть их, либо же выбросить исключение.

Для этого слоя важно, чтобы все концепции, перечисленные выше, были соблюдены. Так, каждый репозиторий должен объявлять единый источник истины, и все данные, предоставляемые им, должны быть получены именно из этого источника. Также важно, чтобы раскрываемые данные были неизменяемыми, чтобы исключить их повреждение в источнике.

Опциональный domain слой является промежуточным между двумя основными и служит для инкапсуляции сложной бизнес-логики, либо же логики, повторяющейся для нескольких различных точек в UI слое. Среди его преимуществ можно выделить сокращение повторений кода, повышение читаемости и тестируемости созданного решения, а также разделения ответственности для сложных случаев обработки данных. Все компоненты этого слоя должны быть максимально легковесными, и поэтому все юзкейсы (сценарии использования) должны быть ответственны только за одну функцию, и не должны содержать никаких состояний.

Важной особенностью домен-слоя является полное отсутствие связи с фреймворком Android, и все используемые классы (в том числе и модели, называемые сущностями) не должны содержать в себе типов, привязанных к Android. При соблюдении этого правила слой легко тестируем, а его логику можно легко переиспользовать. Это же правило касается и интерфейсов репозиториев, которые объявляются в данном слое. Таким образом работа происходит с «чистыми» данными, а их преобразование при передаче между слоями лежит вне зоны ответственности домена.

Ролью последнего компонента архитектуры, UI-слоя, исходя из его названия, является отображение данных приложения на экране, а также обработка событий пользователя. При любом изменении данных этот слой должен отреагировать изменением того, что пользователь видит на экране. При этом не все данные, получаемые из data-слоя, отображаются на экране или как-то влияют на интерфейс. Исходя из этого, UI слой можно определить как конвейер, который преобразует изменения данных приложения в форму, удобную для отображения, а затем показывает их на экране.

Одним из важнейших понятий для слоя отображения является состояние UI (или UI стейт). Оно определяет, как именно выглядит интерфейс для пользователя, и содержит не только данные, полученные из data-слоя, но и локальные состояния элементов интерфейса (например, значения изменяемых полей или выбранная опция в списке предложенных). Любые изменения в стейте (неважно, вследствие события пользовательского ввода или изменения данных в источнике) должны быть немедленно отражены в интерфейсе.

Классы, ответственные за хранение и производство состояния UI, а также первичную обработку возникающих событий, называются держателями состояния (state holders). Рекомендуемым решением для их реализации является паттерн MVVM (Model-View-ViewModel). В нём основным объектом является ViewModel, которая содержит и раскрывает методы для обработки событий, а также потоки (в частности, StateFlow) состояния UI, на обновление которых могут подписаться View (экраны приложения, либо отдельные их компоненты). При этом соблюдается концепция Unidirectional Data Flow: состояние UI передаётся только от ViewModel к View (и далее вниз по иерархии компонентов), в то время как события идут в противоположную сторону путём вызова соответствующих методов ViewModel. Этот факт хорошо прослеживается на схеме UI слоя (рис. 3.2).

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рис. 3.2. Схема UI-слоя

Важной особенностью архитектуры является различное представление данных на каждом из слоёв. Это необходимо, чтобы свести к минимуму зависимости между слоями, и оставить возможность заменить реализацию какого-то слоя, не затрагивая другие. Это накладывает обязанность контролировать преобразование данных между слоями (даже если их формат не меняется на всём пути от источника данных до UI). Для этого необходимо писать мапперы — вспомогательные объекты, единственная задача которых состоит в приведении формата данных из одного слоя в другой. При наличии слоя домена мапперы приводят данные к доменным сущностям, и они считаются основным форматом представления данных.

При наращивании архитектуры растёт количество зависимостей как внутри слоёв, так и между ними. И чтобы эти зависимости не приходилось создавать самостоятельно, необходимо использовать принцип Dependency Injection. Для этого была использована библиотека Hilt, строящая дерево зависимостей и обеспечивающая все части системы необходимыми компонентами с разрешением конфликтов на этапе компиляции.

Ещё одной важной вещью, связанной с архитектурой проекта, является структура приложения, то есть способ организации кода. Правильно определённая структура проекта ускоряет процесс разработки, облегчает поддержку приложения и упрощает его расширение. Во время разработки приложения для решения поставленной задачи экспериментальным путём была подобрана оптимальная структура при вышеописанных архитектуре и наборе библиотек. Это исследование послужило фундаментом для статьи [7].

## UI/UX дизайн приложения

При разработке дизайна приложения были использованы рекомендуемые практики для создания UI/UX дизайна в контексте мобильных приложений. В качестве основной дизайн-системы было выбрано рекомендуемое решение для Jetpack Compose — Material Design 3. Это адаптируемая система руководств, компонентов и инструментов, которые поддерживают лучшие практики проектирования пользовательского интерфейса. У этой системы открытый исходный код, что упрощает связь между дизайнером и разработчиком (в случае командной работы) и помогает быстро создавать красивые продукты.

Основой Material Design являются 3 подсистемы: цветовая схема, типографика и формы. Изменения в любой из них будут отражены во всех компонентах, использующихся на экранах приложения.

Контейнеры в приложении могут быть отображены с использованием различных форм. Формы привлекают внимание, идентифицируют компоненты и передают состояние. Для них в системе M3 существует шкала размеров с семью стилями, разнящимися степенью закругления (от полностью круглой формы до отсутствия закругления). Это отличает данную систему от предыдущего подхода, где компоненты были сгруппированы по размеру контейнера. В приложении используются среднезакруглённые формы (с диаметром закругления углов 16 dp) для полей ввода и крупных карточек, а также малозакруглённые (с диаметром закругления углов 8 dp) для малых карточек.

Шрифт, используемый в приложении, Roboto, принимает различный вид в зависимости от места использования благодаря системе типографики в M3. Все стили текста разделяются на типы по 2 признакам: область применения (display, headline, title, body, label) и размер (large, medium, small). Каждый из этих 15 типов определяется различными параметрами шрифта, такими как размер, вес, межбуквенное расстояние и др. Используя различные стили, можно акцентировать внимание на определённых предметах интерфейса, упрощать навигацию по экрану и визуально разделять логические блоки.

Цветовую схему в системе M3 разработчики сравнивают с раскраской по номерам: каждый отдельный элемент интерфейса как будто имеет цифру, а для каждой цифры существует определённый цвет, в который «красятся» все элементы интерфейса с этой цифрой. Каждый цвет имеет свою роль (основной, контейнер, поверхность и др.) и свой тон (первичный, вторичный, ошибки и др.). Роли цветов определяют, какие компоненты (или их части) будут окрашены в тот или иной цвет. Тон цветов контролирует фокус пользователя, заставляя обратить внимание на нужные зоны (например, кнопки действий). Эта система позволяет выделить компоненты экрана в группы, чаще всего требующие одинаковую степень внимания со стороны пользователя, и гибко менять их цвет в одном месте. Основная палитра, используемая в разработанном решении, представлена на рис. 3.3.

Конечно, не все решения в выборе цвета для того или иного элемента вписываются в данную схему. Поэтому существует возможность расширять систему собственными ролями и тонами. Так, например, на экране сканера выделена отдельная роль под полупрозрачную кнопку над предпросмотром камеры.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рис. 3.3. Цветовая палитра приложения

## Навигация в приложении

Навигация — это набор взаимодействий, позволяющих пользователю свободно перемещаться между частями приложения, а также сохранять историю его перемещений для корректной обработки нажатия кнопки «назад» на устройстве. Приложение состоит из нескольких экранов, каждый из которых отвечает за определённую часть функционала. Соответственно, чтобы пользователь мог легко перемещаться между этими частями, необходимо разработать граф перемещений, определяющий правила переходов между экранами. Этот граф изображён на рис. 3.4.

Так как интерфейс приложения строится с помощью Jetpack Compose, а также используется принцип Single Activity (который ограничивает количество активностей в приложении одной), для организации навигации была выбрана библиотека Jetpack Navigation.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Веб-сайт

Автоматически созданное описание

Рис. 3.4. Граф перемещений, содержащий переходы между экранами (черные сплошные стрелки), переходы внутри экранов (пунктирные стрелки), и допустимые возвраты по стеку (серые стрелки).

Основой системы навигации в используемой библиотеке являются 3 концепции, справедливые для любой её имплементации:

* Граф (Graph) — структура данных, определяющая «пункты назначения» в контексте переходов между экранами, а также сами переходы.
* Хост (Host) — элемент пользовательского интерфейса, который содержит в себе текущий узел навигационного графа. Таким образом, когда пользователь совершает переход в графе, он по сути меняет узел в хосте.
* Контроллер (Controller) — механизм, обеспечивающий переходы между узлами графа, меняющий состояние хоста и контролирующий бэкстек (стек предыдущих экранов, необходимый для корректных переходов назад).

В условиях декларативного подхода к построению интерфейса с Jetpack Compose хост является корневым компонентом UI-дерева и определяется в отдельном файле с использованием composable функции NavHost. Внутри неё определены настройки анимации для всех экранов (используются низкие коэффициент демпфирования и жесткость), а также узлы графа навигации — экраны приложения, в которые в качестве параметра передаётся контроллер для организации переходов.

При объявлении узла графа необходимо определять путь до него, а также набор параметров, передаваемых вместе с событием перехода на экран. Простой способ работы с этими данными — передача в качестве параметров соответствующих строк во всех необходимых местах, но такой подход может привести к трудно локализуемым ошибкам, например, из-за опечаток. Для избежания такой ситуации были созданы специальные абстрактные классы (пример одного из них для передачи 1 параметра отражен в листинге 3.1).

Листинг 3.1. Абстрактный класс для определения узла графа навигации с одним аргументом и интерфейс для экранов без аргументов

interface NavigationDestination {  
 val route: String  
}  
  
abstract class NavigationDestinationWithArg<T> : NavigationDestination {  
 abstract val routeBase: String  
 abstract val argName: String  
 override val route: String  
 get() = "$routeBase/{$argName}"  
  
 fun getDestinationWithArg(arg: T) = "$routeBase/$arg"  
}

Для каждого экрана необходимо определить наследника данного класса и переопределить названия аргументов и путей. Такой подход к определению узла обеспечивает согласованность и целостность во всех местах его использования.

## Главный экран

Первый экран, который видит пользователь, установивший и открывший приложение, это главный экран. Он является пунктом предварительной настройки параметров суммаризации, а также просмотра уже обработанных текстов. Данный экран реализует следующие функции:

* выбор типа суммаризации (экстрактивный или абстрактивный);
* настройка степени сокращения текста (при выбранном экстрактивном подходе);
* ввод текста для суммаризации;
* вставка текста из буфера обмена;
* запуск сканера для считывания текста с помощью камеры —устройства;
* просмотр истории суммаризованных текстов.

Главным элементом на экране является текстовый редактор. В его основе лежит стандартный компонент TextField, настроенный для ввода больших объёмов текста. В целях целостности интерфейса максимальное количество видимых строк ограничено 6, после чего внутри текстового поля появляется возможность прокручивания содержимого. Вместо того, чтобы хранить состояние поля прямо внутри composable функции редактора, оно хранится во ViewModel и передаётся вниз по дереву компонентов, а при каждом введенном символе срабатывает коллбэк из ViewModel. Такое поведение обусловлено соблюдением принципа Unidirectional Data Flow, и это упрощает дальнейшее использование введенного текста.

Для изменения настроек процесса суммаризации над полем ввода реализован переключатель типов (экстрактивный или абстрактивный). Это созданный с нуля компонент, построенный на основе динамически расширяемой LazyRow, состоящей из компонентов Text. Созданное решение является переиспользуемым, и поэтому определено в общем пространстве компонентов. В зависимости от выбранной опции становится доступным редактирование коэффициента сжатия исходного текста с помощью слайдера (см. рис. 3.5). Для этого используется специальный компонент AnimatedVisibility, позволяющий анимировать скрытие слайдера при изменении состояния. Значения слайдера дискретны, разбиты на 10 шагов и могут принимать значения от 0.1 до 1.0, что отражает желаемый размер сокращенного текста в отношении к исходному.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, Шрифт, Веб-сайт

Автоматически созданное описание

Рис. 3.5. Внешний вид интерфейса в зависимости от выбранного типа суммаризации

Все основные функции, связанные с вводимым текстом, находятся в блоке под полем ввода. Главная, (или же первичная) кнопка в строке, отражает главное действие на экране — запуск процесса суммаризации текста. По нажатию на эту кнопку открывается соответствующий экран, в который передаётся введённый текст, а также параметры суммаризации.

В ходе исследования подходов для передачи большого объёма текста между узлами графа навигации было выявлено, что передача текста через параметры навигации могла бы привести к снижению эффективности работы компонента навигации в силу того, что передача параметров происходит аналогично query-параметрам в терминах веб-разработки (это означает, что для навигации составляется запрос, состоящий из непосредственно пути к узлу и фактических значений параметров, как показано в листинге 3.1).

Для решения этой проблемы был выбран другой путь передачи: текст сохраняется во временный файл в кэше приложения, после чего в качестве фактического параметра между экранами передаётся название созданного файла. В качестве имени файла используется рандомно сгенерированный неизменяемый универсальный уникальный идентификатор (UUID). Файл создается в кэш-папке приложения, путь к которой можно получить из контекста. Данная директория отличается от других тем, что для работы в ней приложению не требуется никаких дополнительных разрешений. Но в то же время система Android может очищать её при недостатке памяти в любой момент, поэтому она не подходит для длительного хранения пользовательских данных. В данном случае файл нам необходим на протяжении нескольких сотен миллисекунд, поэтому вероятное удаление не грозит целостности данных. Так как работа с записью на диск — задача Data слоя, был создан отдельный источник данных, CacheDataSource, работу с которым ведёт CacheRepository. Соответственно, был также определён отдельный usecase в Domain слое.

После сохранения исходного текста в кэше приложении необходимо перейти на экран суммаризации для его последующей обработки, но эта задача, несмотря на кажущуюся простоту, не является тривиальной. Проблема состоит в асинхронной природе механизма композиции Jetpack Compose и общей концепции декларативного подхода к созданию интерфейса.

Императивное программирование (принятое в разработке с использованием Views) фокусируется на описании шагов, которые должно выполнить приложение для достижения некоторой цели (перехода на экран, в данном конкретном случае). Декларативное программирование, напротив, сосредоточено на описании того, что нужно достичь без указания конкретных шагов для достижения цели. Именно поэтому в Compose (и других похожих подходах, таких как Flutter) нельзя напрямую взаимодействовать с компонентами UI, определяя, что они должны делать. Кроме того, согласно принципу разделения ответственности ViewModel не может самостоятельно работать с компонентами системы Android, а должна быть только стейт-холдером.

Для решения этой проблемы необходимо воспользоваться преимуществами паттерна MVVM и создать внутренний data класс UiState, хранящий в себе состояние интерфейса. Этот класс помещается в StateFlow (реактивный поток данных, реализующий паттерн Observer), отслеживаемый в composable-функции главного экрана внутри LaunchedEffect — специальной функции, тело которой вызывается при первой композиции и всякий раз, когда изменяются переданные в неё ключи. Ключом в данном случае является собираемый из потока UiState, хранимый во ViewModel.

Для организации навигации в классе UiState было определено nullable-свойство summarizationNavigationArg, тип которого объединяет все параметры, необходимые для перехода на экран суммаризации. Его значение по умолчанию установлено в null, и общий механизм работы работает следующим образом:

* при нажатии на кнопку суммаризации ViewModel собирает все необходимые данные, объединяет их, и обновляет StateFlow, размещая в потоке новый объект с актуальным значением аргумента;
* обозреватель потока реагирует на новое значение в потоке и преобразует его в стейт компонента главного экрана;
* LaunchedEffect фиксирует изменения стейта и выполняет код внутри себя;
* ViewModel реагирует на вызов коллбэка, извещающего её о том, что событие перехода на новый экран было обработано и устанавливает summarizationNavigationArg обратно в null, предотвращая срабатывание LaunchedEffect ещё раз;
* выполняется переход на новый экран с использованием контроллера навигации.

Рядом с кнопкой для перехода на экран суммаризации текста располагаются две вторичные кнопки. Одна из них, расположенная слева, заменяет текст в поле ввода на текст из буфера обмена. Это поведение достигается путём использования стандартного компонента системы Android — ClipboardManager. Он позволяет получать текст из буфера обмена, а также информацию о нём. В случае, если в буфере отсутствует какая-либо текстовая информация, кнопка становится неактивной. Другая вторичная кнопка отвечает за запуск функции сканера. Процесс перехода на данный экран отличается от рассмотренного ранее варианта.

Главное отличие состоит в том, чтобы перед переходом на экран проверить факт получения разрешения на использование камеры от пользователя. На новых версиях Android использование камеры без подтверждения данного разрешения пользователем во время работы приложения запрещено. Именно поэтому перед работой необходимо проверить наличие разрешений, после чего переходить на экран, использующий камеру. Для запроса разрешений запускается специальная активность, которая показывает окно запроса разрешения у пользователя на экране (см. рис. 3.6. (а)). Если пользователь соглашается, выполняется прямой переход на соответствующий экран. Так как этот переход инициирован внутри composable функции, а также не требует никаких аргументов, нет необходимости организовывать его, как в примере, рассмотренном ранее. Достаточно просто обратиться к контроллеру и запустить переход на заданный узел. Если же пользователь отказался выдать разрешение на использование камеры, приложение выводит предупреждение, что эта функция недоступна (см. рис. 3.6 (б)).

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, Шрифт, веб-страница

Автоматически созданное описание

Рис. 3.6. (а) Окно подтверждения разрешения на использование камеры

(б) Предупреждение о невозможности использования сканера без разрешения на использование камеры

Проверка наличия разрешения совершается на основе запроса данной информации из контекста (объект, хранящий текущее состояние приложения, и системы). Механизм запуска получения разрешения основан на использовании лаунчера — контракта, описывающего запрос на запуск активности, а также действий, совершаемых по факту выполнения данного запроса. Лаунчер хранится в качестве состояния composable-функции и вызывается в случае, если при проверке оказалось, что необходимого разрешения не выдано (этот факт отражён в листинге 3.2). Для корректной работы запрашиваемое разрешение необходимо также объявить в Андроид манифесте, чтобы пользователь мог узнать о потенциальном использовании камеры во время установки приложения.

Листинг 3.2. Обработчик, срабатывающий при нажатии на кнопку сканирования

onBtnCameraClick = **{** if (ContextCompat.checkSelfPermission(  
 context,  
 android.Manifest.permission.*CAMERA* ) == PackageManager.*PERMISSION\_GRANTED* ) {  
 navController.navigate(ScannerDestination.route)  
 } else {  
 cameraPermissionLauncher.launch(android.Manifest.permission.*CAMERA*)  
 }  
**}**,

В отличие от перехода на экран суммаризации, экран сканирования должен вернуть данные — отсканированный текст. Это реализовано с помощью передачи данных через BackStack — структуру данных, которая хранит последовательность экранов или фрагментов, через которые прошел пользователь, позволяя реализовать навигацию «назад». В компоненте обозревается LiveData состояния BackStack, и при появлении нового значения запускается LaunchedEffect, обновляющий поле ввода переданной информацией.

В нижней части главного экрана располагается история суммаризаций. Она представляет из себя набор карточек, расположенных в LazyHorizontalGrid в 3 ряда, по нажатию на которые открывается подробный просмотр обработанного текста. Для того, чтобы не создавать визуальный шум, все карточки имеют одинаковый размер вне зависимости от длины итогового текста. При длине текста, большей чем может поместиться на карточку, текст обрезается троеточием в конце.

По клику на карточку открывается BottomSheetDialog — один из стандартных видов представления информации в Android. Он представляет собой диалоговое окно, выдвигающееся снизу экрана, при этом оставляя видимой часть основного контента для контекста. При необходимости его можно «вытянуть» снизу и открыть на весь экран (см. рис. 3.7). Для избежания повторения одного и того же кода для отображения данных похожего вида внутри диалога используется тот же самый компонент, что и в экране суммаризации для демонстрации результата работы алгоритма.

Механизм показа диалога похож на механизм перехода между экранами с аргументами, с той разницей, что вместо триггера LaunchedEffect данный диалог, по сути, является частью дерева компонентов. Кроме того, немного по-другому работает механизм извещения об окончании события показа: вместо моментального извещения сразу после появления в дереве компонентов, UiState меняется только после закрытия или отмены диалога.

Изображение выглядит как текст, электроника, снимок экрана, Шрифт

Автоматически созданное описаниеИзображение выглядит как текст, письмо, Шрифт, бумага

Автоматически созданное описание

Рис. 3.7. Просмотр элементов истории суммаризации в а) компактном виде; б) полном виде

Данные о предыдущих суммаризациях хранятся в локальной базе данных. Для работы с ней была использована библиотека Room, которая является абстракцией над SQLite и упрощает работу с базами данных за счёт использования аннотаций для определения сущностей и запросов. Она обеспечивает проверки на этапе компиляции, обеспечивая тем безопасность типов, а также поддерживает интеграцию с корутинами для асинхронной работы. Room состоит из трёх основных компонентов: Entity для представления моделей и настройки таблиц, DAO (Data Access Object) для определения методов доступа к данным и их изменения, а также Database для создания и настройки базы данных и связывания всех компонентов.

Для хранения истории обработанных текстов нужна только одна таблица, хранящая модели со структурой, отражённой в листинге 3.3. Id каждой записи генерируется автоматически при добавлении в базу.

Листинг 3.3. Модель для хранения истории суммаризации

@Entity(tableName = "summarized\_texts")  
data class SummarizedTextData(  
 @PrimaryKey(autoGenerate = true) val id: Long = 0,  
 val timestamp: Long,  
 val originalText: String,  
 val summarizedText: String,  
 val compressionRate: Int  
)

Для получения данных из базы необходимо использовать облегченный вариант языка SQL. Запросы пишутся в виде аннотаций методов интерфейса DAO, на основе которых во время этапа компиляции генерируется код. Данные извлекаются в виде Flow, который преобразуется в StateFlow внутри ViewModel с использованием viewModelScope. Это позволяет интерфейсу реагировать на изменения данных в базе в реальном времени.

## Экран сканирования документа

Альтернативой вводу текста вручную, используя поле ввода на главном экране приложения, является сканирование текста с помощью камеры устройства. Эта функция может оказаться особенно полезной в случае необходимости обработки информации, которая содержится в документе, представленном в печатном виде. Как уже было рассмотрено в предыдущей главе, на данный экран можно попасть только с полученным разрешением на использование камеры устройства, поэтому дополнительных проверок перед началом работы не выполняется.

Работа с камерой в системе андроид производится с использованием библиотек Camera2 (пришедшей на замену устаревшей Camera), а также CameraX, являющейся часть набора библиотек Jetpack. Camera2 — это мощный инструмент, позволяющий работать с камерой на низком уровне абстракции. Но в то же время при её использовании даже простые задачи требуют работы с многими различными сущностями и написания большого объёма кода. Подход, лежащий в основе библиотеки — моделирование работы камеры в виде конвейера, который принимает входные запросы для захвата одного кадра, захватывает этот кадр в соответствии с переданным запросом, а затем выводит единый пакет метаданных о результатах захвата, а также набор буферов выходных изображений.

В качестве альтернативы к многофункциональной, но перегруженной библиотеке Camera2 компания Google представила надстройку к ней — CameraX. Она отличается простотой использования и меньшим количеством кода для решения тех же простых задач. Например, сервис Monzo при переходе с API Camera2 на CameraX сократил кодовую базу проекта на 9000 строк кода, а также уменьшил количество неудавшихся регистраций в 5 раз [8]. Основным компонентом в этой библиотеке являются юзкейсы — наиболее распространённые сценарии работы с камерой (превью, съёмка фото и видео, анализ изображений). Их можно комбинировать и гибко настраивать, что позволяет сфокусироваться на выполнении поставленной задачи, а не на тонкой настройке технических параметров систем съёмки. Кроме того, библиотека занимается унификацией различных параметров на устройствах разных производителей, ограждая разработчика от необходимости учитывать многообразие технических нюансов работы камеры на разных устройствах.

Существенным недостатком библиотеки CameraX является отсутствие поддержки Jetpack Compose на момент разработки мобильного приложения, а это означает, что все её компоненты ориентированы на работу с Views. Но в то же время библиотека Compose построена таким образом, что созданные при помощи этого подхода компоненты можно было использовать во Views, и наоборот, любые Views могут быть встроены в composable функции. Механизм, позволяющий осуществить это, заключен во встроенном компоненте AndroidView. Эта функция принимает в качестве параметра блок factory, который вызывается ровно один раз при первой композиции. Внутри этой фабрики создаётся сам View, а также инициализируются необходимые параметры.

Главной частью камеры, как приложения на мобильном устройстве, является превью, то есть изображение, получаемое с камеры в реальном времени и отображаемое на экране. Именно имплементация превью была выполнена с помощью механизма AndroidView. В фабрике возвращается представление PreviewView — компонента библиотеки CameraX, отвечающего за показ превью на экране. Для того, чтобы привязать жизненный цикл камеры к жизненному циклу экрана, используется ProcessCameraProvider, при привязке которого обрабатываются все используемые юзкейсы. После привязки провайдер возвращает объект Camera, который сохраняется внутри компонента для дальнейших взаимодействий с параметрами съёмки. Камера привязывается к жизненному циклу LocalLifecycleOwner, что обеспечивает корректное поведение при сворачивании приложения, переходами между экранами, приостановке и возобновлению работы приложения (например, при поступившем телефонном звонке) и других случаях.

Помимо объекта самой камеры внутри компонента должны содержаться все юзкейсы, используемые для решения задачи. Схема работы ограничена просмотром превью и съёмкой изображения, передаваемого далее на обработку и анализ, и в компоненте содержатся два соответствующих юзкейса, отвечающих за каждое из действий. Оба сценария содержат механизм автоматического выбора разрешения (одного и того же, чтобы получать одинаковый результат при съёмке и просмотре превью). Стратегия подбора основывается на выборе соотношения сторон вместо качества изображения. При этом предпочитается соотношение 16х9, а если оно по каким-то причинам недоступно, выбирается любое другое. Помимо выбора разрешения, настраиваются такие нюансы, как режим работы вспышки и соотношение качества съёмки и её скорости.

Изображение выглядит как текст, снимок экрана, Печать

Автоматически созданное описание

Рис. 3.8. Интерфейс камеры

Для камеры доступны настройки приближения и вспышки (см. рис. 3.8). Они реализованы в качестве слоя управляющих элементов над превью, при изменении состояния которых меняется состояние родителя, передающееся вниз по дереву компонентов и обозреваемое LaunchedEffect. Внутри эффектов вызываются соответствующие методы уже композированного объекта Camera. Аналогичным образом происходит процесс захвата изображения, только с использованием сохранённого внутри компонента юзкейса для съёмки. После успешного захвата дальнейшее взаимодействие с изображением передаётся вверх по цепочке событий к ViewModel.

После получения изображения начинается его обработка. Перед началом процесса распознавания текста необходимо убедиться, что в кадре нет ничего лишнего, что может помешать сканированию текста. Кроме того, пользователю необходимо дать выбор, какая часть захваченного изображения его интересует в большей степени. Например, на фото листа А4 с большим объёмом текста может понадобиться лишь несколько абзацев, а при отсутствии возможности выбрать не обходимый объём в итоговую версию суммаризованного текста попадёт контекст из неактуальных частей документа.

Изображение выглядит как текст, снимок экрана, Печать, письмо

Автоматически созданное описание

Рис. 3.9. Инструмент обрезки изображения

Для решения вышеописанной проблемы был реализован функционал обрезки изображения по произвольному прямоугольнику (см. рис. 3.9). Пользователь может двигать углы прямоугольника, тем самым меняя его размер и положение на изображении. Кроме того, прямоугольник можно перетаскивать, начиная жест из области внутри него.

В основе решения для обрезки изображений лежит компонент BoxWithConstraints, состояние которого зависит от действий пользователя. Его особенностью является возможность, в отличие от стандартного Box, предоставлять дочерним компонентам доступ к значениям ограничений размера — минимальным и максимальным значениям высоты и ширины, вычисляемым на этапе размещения компонентов, части механизма рендеринга UI библиотеки Jetpack Compose [9]. Эти значения используются для ограничения размера обрезающей рамки, расчёте её изначальных размеров и положения на экране, а также при её позиционировании.

Для корректного изменения состояния рамки используется набор коллбэков, вызываемых при взаимодействии с её углами. Каждый из них задаёт изменения положения верхней левой точки рамки, а также её высоты и ширины. При пересчёте позиции рамки учитываются также и параметры исходного изображения, чтобы избежать ситуации выхода рамки за его пределы. Процесс обрезки заключается в создании новой Bitmap, «вырезая» её из оригинальной. При этом для каждого компонента ограничивающей рамки выполняется перевод из значений Dp (Density independent Pixels, пиксели, не зависящие от плотности — абстрактная единица измерения, основанная на физической плотности экрана) в пиксели по формуле:

— исходное значение в Density independent Pixels;

— ширина и высота экрана в пикселях;

— диагональ экрана в пикселях.

Обрезанное изображение передаётся на следующий этап сканирования, обработку с помощью нейросетей. Для совершения этого преобразования используется ML Kit, набор инструментов, разработанный Google, для решения частых задач мобильной разработки с использованием техник машинного обучения. Среди особенностей данного подхода при решении задачи распознавания текста можно выделить простоту интеграции данного решения, выделение структуры обрабатываемого текста при распознавании и возможность проводить процесс распознавания текста прямо на устройстве.

Предобработанное изображение, содержащее текст, в формате Bitmap передаётся определённому на уровне корневого компонента приложения классу MLManager. Данный компонент является обёрткой над MLKit, инкапсулирующей в себе первичную настройку и работу с API данного решения. Эта сущность выделена для того, чтобы работа с библиотекой была сосредоточена в одном месте, что упрощает внесение изменений и поддержку работы функции сканера. Сканирование запускается в отдельной корутине, использующей диспетчер для затратных вычислительных операций, что помогает избежать блокирования UI-потока. В результате работы модели на экран выводится распознанный текст и кнопка его копирования (см. рис 3.10 (а)). Редактирование полученного результата будет доступно на главном экране приложения.

Несмотря на простоту имплементации, высокие скорость и качество работы, выбранный подход обладает недостатками, характерными для проприетарных решений. Например, компания Google не раскрывает подробностей, какие именно архитектуры нейросетей лежат в основе распознавания текста. Нет было найдено никаких статей и официальных заявлений на этот счёт, поэтому к данному механизму стоит относиться как к чёрному ящику, корректно выполняющему свою работу, но внутреннее устройство которого неизвестно. Кроме того, ML Kit не поддерживает кириллицу, поэтому при сканировании текста, написанного на русском языке все символы распознаются, как соответствующие знаки латиницы (либо же их сочетания, напоминающие русские буквы). Эта ситуация отражена на рис. 3.10 (б). При этом невозможно предпринять какие-либо действия по исправлению ситуации, потому что реализация алгоритма сканирования текста скрыта. Именно поэтому в данной ситуации было принято решение оставить сканирование только латиницы. При этом ввод кириллицы всё так же доступен вручную или путём вставки из буфера обмена.

Изображение выглядит как текст, Шрифт, письмо, снимок экрана

Автоматически созданное описаниеИзображение выглядит как текст, письмо, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 3.10. Результат распознавания текста, написанного

а) латиницей и б) кириллицей

## Экран суммаризации

После ввода текста путём сканирования, вставки из буфера обмена или введения вручную в поле ввода пользователь нажимает на кнопку суммаризации и попадает на экран обработки текста. Исходный текст передаётся с главного экрана через кэш-файл (механизм передачи больших объёмов текста между экранами с использованием кэша описан в главе 3.5). Помимо текста передаются параметры суммаризации: её тип (абстрактивная или экстрактивная), а также желаемая длина текста в случае выбора экстрактивной суммаризации.

Процесс суммаризации запускается сразу же, как пользователь переходит на экран. Обработка происходит в отдельном потоке, чтобы не нагружать UI-поток. Для того, чтобы показать состояние обработки текста, был выбран подход с бесконечной анимацией, проигрывающейся, пока не будет получен результат обработки текста. Для воспроизведения анимации была выбрана библиотека Lottie — open-source проект от команды AirBnb. Это решение стало эталонным в сфере анимации на Android. Все анимации основаны на векторной графике, что позволяет им без проблем масштабироваться на любые размеры экранов (в отличие от использования GIF-анимаций). Кроме того, библиотека просто интегрируется в проект на Jetpack Compose, не теряя при этом в производительности. Анимации представлены в едином формате JSON, в который можно экспортировать из большинства популярных решений для создания анимаций.

По завершении процесса обработки текста результаты сохраняются в локальную базу данных, обращаясь к SummarizationRepository. Благодаря этому полученные результаты сразу же добавляются к истории обработок на главном экране, и после выхода с экрана суммаризации отображаются в списке обработанных текстов. Кроме того, результаты показываются на экране, разделённом на 2 части по горизонтали, где сверху можно увидеть исходный текст, а снизу — суммаризованный (см. рис. 3.11). Пропорции разделения выбраны 2:3, где большую часть занимает обработанный текст. При таком соотношении акцент внимания пользователя больше сосредоточен на результате суммаризации, при этом оставляя возможность ознакомиться с исходным текстом для сравнения. Поля, содержащие текст, поддерживают копирование и другие действия с текстом, поддерживаемые системой, но при этом являются неизменяемыми.

Изображение выглядит как текст, письмо, Шрифт, бумага

Автоматически созданное описание

Рис. 3.11. Экран просмотра результата суммаризации

Для обработанного текста вычисляется коэффициент сокращения текста, отражающий, на сколько процентов был сокращён исходный текст. Вычисление этого значения ведётся по количеству символов в обработанном тексте в отношении к исходному.

Процесс обработки текста заключается в отправке запроса к серверу, обрабатывающему его и возвращающему сокращённый текст. Работа с сетью выполняется при помощи библиотеки Retrofit. Она использует кодогенерацию на основе интерфейсов и аннотаций. Таким образом все запросы к сети выглядят как набор абстрактных функций, помеченных аннотациями, отражающими методы запросов и пути на сервере. Функции помечены модификатором suspend, то есть они поддерживают работу с корутинами. Все запросы выполняются в контексте потока ввода-вывода и не являются блокирующими.

# 4. Суммаризация текстов как задача NLP

Суммаризация текстов в контексте обработки естественного языка (Natural Language Processing или NLP) представляет собой процесс автоматического создания краткого содержания исходного текста. Основной целью суммаризации является извлечение наиболее важной информации и представление её в сжатой форме, при этом сохранив основную идею исходного текста. Область исследований по обобщению текста изучается с середины 20-го века, и впервые была упомянута Луном [10]. Решение этой задачи актуально, например, для новостных издательств, которым необходимо генерировать большое количество резюме для выпускаемых новостных статей. Суммаризация находит применение и в обычной жизни: каждый человек сталкивается с большими объёмами информации, которые хотелось бы прочитать и проанализировать в кратчайшие строки, будь то отчёт о проделанной работе, либо научная статья.

Суммаризация текстов подразделяется на несколько категорий. Первая плоскость разделения — количество используемых документов в процессе формирования резюме. Согласно ей, суммаризация может быть однодокументной и многодокументной. При использовании одного документа создается резюме, полученное из одного источника. В то же время многодокументная суммаризация основана на анализе нескольких различных источников, в которых обсуждается одна и та же тема.

С другой стороны, задача суммаризации разделяется на две большие подзадачи: абстрактивную и экстрактивную суммаризацию. Экстрактивная суммаризация — это краткое изложение, при котором итоговый текст полностью состоит из неизменённых частей исходного текста, то есть результатом её работы является набор предложений или слов, полученных из оригинального текста. В отличие от экстрактивной суммаризации, предложения, генерируемые абстрактивным подходом, представляют собой совершенно новые предложения, представляющие из себя изложение исходного текста с возможным использованием слов, отсутствующих в исходном тексте. Соответственно, абстрактивный подход намного более сложен, чем экстрактивный, поскольку создание абстрактных изложений требует генерации текста, являющейся одной из самых сложных задач NLP.

## Экстрактивная суммаризация

Задача экстрактивной суммаризации заключается в выделении фрагментов текста, которые представляют из себя наиболее содержательные части и содержат основной смысл исходного текста. Таким образом, её можно свести к другим задачам машинного обучения, таким как классификация или сегментация. Но при выполнении такого преобразования становится проблематичной разметка обучающей выборки вследствие постановки задачи суммаризации и неопределённости её результата. Кроме того, обобщение опыта суммаризации различных текстов может привести к эффектам недообучения или переобучения модели из-за разницы в контекстах обучающих примеров.

По этой причине более удачным подходом является группа методов машинного обучения без учителя, в частности, кластеризация. В этих методах не учитывается опыт других суммаризаций, и, помимо того, для них не требуется размечать входные данные. Кластерный анализ заключается в разбиении некоторого множества объектов на подмножества похожих друг на друга объектов. При этом процесс определения кластера не опирается на результаты предварительного обучения, а учитывает только признаки распределяемого объекта. Это свойство алгоритма кластеризации компенсирует вышеупомянутые особенности решения задачи суммаризации для экстрактивного подхода.

Идея, лежащая в основе разработанного решения, состоит в особой интерпретации сформированных групп объектов. В классической задаче кластеризации наибольший интерес представляют объекты, попавшие в один кластер и их взаимосвязь. Для извлечения наиболее важных предложений в тексте, напротив, основное внимание было обращено лишь на одного наиболее характерного представителя каждого кластера. Алгоритм разделяет предложения исходного текста по контексту, и близкие по смыслу предложения попадают в один кластер. Из каждого набора предложений, попавших в один кластер, выбирается одно, наиболее репрезентативное из группы. Таким образом происходит фильтрация предложений, содержащих похожую информацию и из текста извлекается заданный объём текста для построения суммаризации.

До применения алгоритма построения кластеров текст должен пройти несколько этапов обработки. Первое, что необходимо сделать — разбить текст на синтаксические единицы. В качестве них могут выступать предложения, отдельные слова или их сочетания (например, биграммы — сочетания двух любых слов). Для решения поставленной задачи было выбрано разбиение исходного текста на предложения. Этот процесс был произведён при помощи библиотеки NLTK, а точнее содержащейся в ней моделью машинного обучения Punkt [11]. Она позволяет учитывать сокращения и обращения (которые содержат точки, потенциально распознаваемые как конец предложения), а также другие случаи, затрудняющие разбиение текста на набор предложений.

Следующим после токенизации этапом является извлечение признаков из каждого отдельного предложения. В контексте обработки естественного языка этот процесс заключается в преобразовании текста в вектор чисел, его характеризующих (поэтому процесс называют векторизацией). На основе исследования [12] лучшие результаты для алгоритма кластеризации можно получить на векторах, созданных нейросетевыми подходами. Одним из таких подходов является векторизация с помощью модели BERT. В основе данной модели лежат трансформеры, построенные на механизмах внутреннего внимания. Эти механизмы похожи на свёртки, часто применяемые в области обработки изображений, и они показали свою эффективность для решения различных задач, связанных с обработкой текстовых данных. Для векторизации была использована модель MPNet, основанная на методе маскированного перестановочного языкового моделирования [13]. В результате векторизации каждое предложение представляется в виде набора из 768 значений, содержащих контекст и смысл предложения.

В результате предварительной обработки текста на выходе получаются векторы фиксированного размера. Их можно рассматривать в качестве точек в многомерном пространстве и оценивать их близость по косинусному расстоянию. Этот факт лежит в основе применяемого алгоритма кластерного анализа — K-Means. Данный подход относится к классу алгоритмов на основе прототипов. Это означает, что каждый кластер однозначно представляется центроидом — точкой, являющейся его «центром». Была использована реализация алгоритма из библиотеки scikit-learn [14]. На рис. 4.1. изображён результат работы алгоритма кластеризации при понижении размерности векторов до двумерного пространства.

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Рис. 4.1. Результат кластеризации для текста из 24 предложений при целевом размере суммаризации 5 предложений

После определения кластеров для каждого предложения вычисляется расстояние до центроида, после чего для каждого кластера выбирается вектор, расположенный ближе остальных к центроиду. Затем каждый вектор преобразуется обратно в текстовое представление и все они конкатенируются в порядке появления в оригинальном тексте. Таким образом результатом работы алгоритма является набор предложений из исходного текста, сохраняющих его основную мысль, что соответствует определению экстрактивной суммаризации.

## Абстрактивная суммаризация

Абстрактивный подход, в отличие от экстрактивного, не использует фрагменты исходного текста для построения результата. По этой причине необходимо использовать генеративные подходы машинного обучения для решения данной задачи. В последние годы заметным трендом в обработке естественного языка стали генеративные нейросети. Многие из них, в том числе архитектуры семейства GPT и BERT основаны на механизме внутреннего внимания, который позволяет моделям по-разному оценивать важность различных частей входного текста, улучшая их понимание контекста и возможности для решения задач.

Для решения задачи абстрактивной суммаризации применение генеративных нейронных сетей является наиболее удобным решением. Но перед непосредственным решением задачи необходимо обучить нейросеть для корректной генерации текста вне рамок задачи суммаризации. Несмотря на множество исследований и большое разнообразие архитектур, обучение такой нейросети требует огромного количества ресурсов. Кроме того, обучение происходит на огромном количестве данных. Например, MPNet, использовавшаяся для векторизации текста при решении задачи экстрактивной суммаризации обучена на датасете размером более 160 ГБ [13]. Длительность обучения малой модели GPT-3 Small оценивается в 2.25E+20 флопсов [15]. Эта мера означает, что при использовании графического процессора NVIDIA TESLA V100 обучение нейросети этой архитектуры займёт около 371 дня. По этой причине обучение модели с нуля нецелесообразно. Для решения этой проблемы был использован подход Transfer learning (трансферное обучение), заключающийся в дообучении уже натренированной модели для решения определённой задачи. Это позволяет сэкономить ресурсы и использовать все накопленные моделью знания.

В качестве основы для трансферного обучения была выбрана модель mT5 Small (multilingual Text-to-Text Transfer Transformer), представленная компанией Google в 2021 году [16]. Она обучена на корпусе текстов на 101 языке и наследует особенности своей родительской архитектуры, T5. Так, все задачи, решаемые при дообучении этой модели, должны сводиться к задаче text-to-text, то есть генерация текста на основе другого. Задача суммаризации подходит под это определение. Структура этой модели основана на правиле энкодер-декодер, где энкодер преобразует входной текст в некое промежуточное состояние, а декодер, в свою очередь, генерирует текст на основе этого состояния.

Для дообучения модели на задачу абстрактивной суммаризации был составлен датасет из новостей из различных источников на 2 языках: русском и английском. Для русского языка был использован датасет заголовков издания Gazeta.ru, а для английского — датасет CNN/DailyMail. Данные из обоих датасетов были объединены в один в пропорции 1:1.

Изображение выглядит как График, диаграмма, снимок экрана

Автоматически созданное описание

Рис. 4.2. Распределение длин исходных текстов в датасете

Распределение средних длин новостей в итоговом датасете отражено на рис. 4.2. Из него видно, что средняя длина текста среди всех вхождений около 4000 знаков. Датасет разбивается на 3 выборки: тренировочную, используемую для обучения модели (30000 экземпляров), валидационную, для контроля над промежуточными результатами обучения (4000 экземпляров) и тестовую (4000 экземпляров).

Перед обучением входные данные предобрабатываются с помощью токенизатора SentencePiece, который в автоматическом режиме разбивает текстовые данные на токены в формате, на котором была обучена модель. Важной особенностью является ограничение максимального количества токенов для модели. В случае с версией small максимальной входной пропускной способностью является 512 токенов, и именно этим значением сверху ограничивается представление входных данных.

Дообучение модели производилось с помощью библиотек Transformer и Keras. Для токенизированных данных определяется обработчик данных DataCollatorForSeq2Seq, преобразующий набор токенов в тензоры input\_ids и decoder\_input\_ids и маску внимания. Во время работы он также обрабатывает целевые данные, сдвигая метки на одну вправо для корректного обучения декодера и исключения возможности его переобучения. Кроме того, он выравнивает наборы токенов до тензоров одного размера, дополняя их набор специальным токеном [PAD].

В качестве основы для тренировки модели был выбран TFAutoModelForSeq2SeqLM. Он автоматически скачивает веса модели, а также определяет оптимальные настройки для её обучения. Для тренировки модели был подобран оптимальный набор гиперпараметров. Так, использовался размер батча (количество примеров, обрабатываемых за одну итерацию обучения) равным 8, learning rate (скорость обучения) равной 5E-5. Длительность обучения установлена в 4 эпохи. График потерь при обучении модели представлен на рис. 4.3.

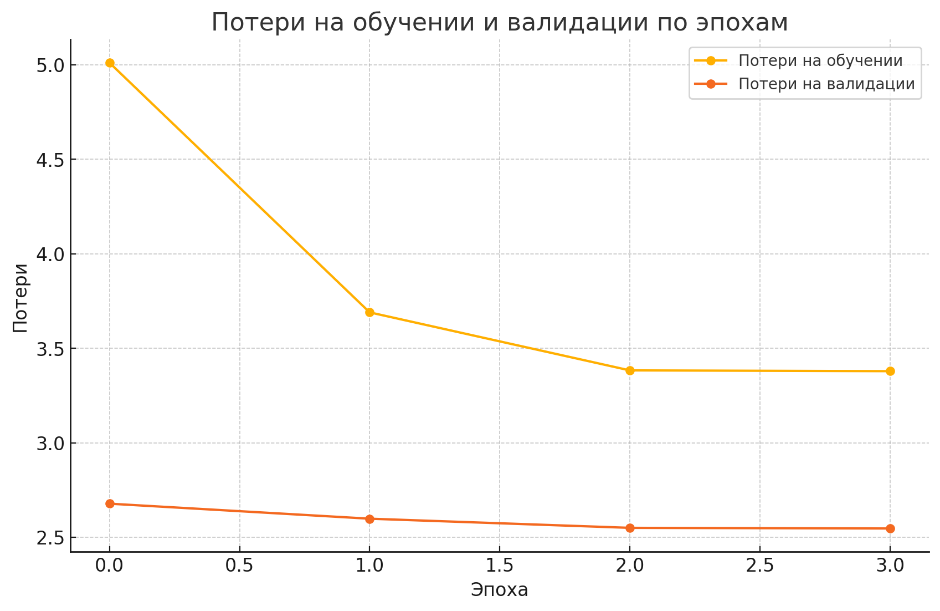


Рис. 4.3. Ход обучения модели абстрактивной суммаризации

## Методы оценки качества суммаризации

Процесс оценивания качества полученных результатов является важной частью исследования. Для многих задач класса алгоритмов обучения с учителем, таких как классификация или сегментация, существует множество различных метрик, позволяющих оценить качество результатов работы модели. Это обусловлено наличием в разметке данных для обучения моделей такого типа однозначно определённых целевых меток, на основании которых можно сделать вывод о корректности работы разработанного алгоритма. В случае с задачей суммаризации задача осложняется отсутствием возможности сформулировать однозначную метрику «правильности» сокращённого текста. Можно руководствоваться определёнными эвристиками, выделяя те или иные аспекты в результате, например, близость по смыслу или количество слов из оригинального текста. Но это не позволяет в полной мере сравнить результат работы двух разных алгоритмов и с точностью сказать, какой из них лучше смог сократить исходный текст.

Именно поэтому оценка качества текста часто проводится с использованием аннотаций (оценок качества суммаризованного текста) от людей. Аннотатор (обычно в его роли выступает человек, не принимающий участия в разработке модели) оценивает полученный текст по нескольким критериям, например, с точки зрения ясности текста, качества извлечения контекста, на основе содержания и другим эвристикам. Другой подход оценки заключается в сравнении аннотатором результатов машинного изложения с эталонным изложением эксперта. Оценка может проводиться путем анализа схожести с точки зрения грамматики, согласованности и отсутствия избыточности. Частично эти способы были использованы при разработке описываемого решения, где в качестве аннотатора выступал автор работы.

К сожалению, «ручное» оценивание результатов работы алгоритма не позволяет в полной мере отразить его качество. Для этого необходима метрика, вычисляемая автоматически. Очевидно, что прямое сравнение результативного текста с эталоном не даст объективной оценки, и при таком подходе даже человек, которого попросили сократить текст, покажет не очень хорошие результаты. Но тем не менее концепция сравнения результата с одним или несколькими эталонами лежит в основе одной из наиболее применяемых для оценки суммаризации метрик, ROUGE (Recall-Oriented Understudy for Gisting Evaluation). Суть метрики заключается в анализе пересекающихся множеств слов и n-грамм по оценке точности и полноты. Кроме того, вычисляется f1-мера — гармоническое среднее этих двух оценок.

В контексте вычисления метрики ROUGE метрика точности оценивает насколько релевантным сгенерированный суммаризованный текст является по сравнению с эталоном. В то время как полнота показывает, насколько точно эталонный сокращённый текст воспроизводится в результативном. Вычисляются эти метрики по следующим формулам:

— количество n-грамм (или, в простом случае, слов), содержащихся и в оцениваемом, и в эталонном тексте;

— количество слов в эталонном тексте;

— количество слов в оцениваемом тексте.

Обычно оценка с помощью метода ROUGE идёт сразу по нескольким типам разбиения текста: rouge1 оценивает по отдельным словам в тексте, rouge2 — по биграммам (парам слов), а rougeL — по наибольшей длине самой длинной общей подстроки в эталонной и оцениваемой суммаризациях. Результаты оценки разработанных моделей представлены в таблице 4.1.

Таблица 4.1. Результаты оценки суммаризации разработанных методов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип суммаризации | Метрика | Precision | Recall | F1-мера |
| Абстрактивная | rouge1 | 0.7333 | 0.1058 | 0.1849 |
| rouge2 | 0.3571 | 0.0485 | 0.0855 |
| rougeL | 0.5333 | 0.0769 | 0.1345 |
| Экстрактивная | rouge1 | 0.4231 | 0.3121 | 0.3592 |
| rouge2 | 0.1068 | 0.0786 | 0.0905 |
| rougeL | 0.2019 | 0.1489 | 0.1714 |

С результатами работы алгоритмов, описанных в этой главе, можно ознакомиться в приложении 1.

# Интеграция моделей машинного обучения

Важной частью разработки решения поставленной задачи суммаризации текстов является внедрение обученной модели машинного обучения в мобильное приложение. Существует два основных подхода к интеграции: внедрение моделей для их использования на мобильном устройстве, а также проектирование клиент-серверной архитектуры с сервером, обрабатывающим данные и приложением, являющимся инструментом для их сбора и отображения результатов обработки.

Внедрение нейросетей непосредственно в мобильное приложение кажется наилучшим подходом, ведь в таком случае приложение не зависит от подключения к интернету и может самостоятельно управлять работой модели. Этот подход изначально рассматривался как основной вариант реализации. Но после получения первых результатов исследования предметной области стало понятно, что внедрение нейросетей непосредственно в мобильное приложение может стать проблемой. Существует множество решений по внедрению нейросетей в Android-приложения, таких как TFLite или PyTorch Mobile. Они успешно применяются для решения различных задач, но в случае с обработкой текста есть важная особенность. Модели для работы с текстом обучаются на больших объёмах данных, и поэтому количество весов в них может достигать миллиарды единиц. Это влияет на объём памяти, занимаемой обученной моделью. Так, обученная модель mT5 для абстрактивной суммаризации занимает 2 ГБ памяти. Это может показаться небольшим объёмом, но не для мобильных устройств, которые ограничены в своих возможностях как в оперативной, так и в постоянной памяти. Использование такого большого объёма памяти нецелесообразно для предоставляемого приложением функционала.

Подход с клиент-серверным взаимодействием, напротив, позволяет разгрузить мобильное приложение и перенести логику работы с нейросетями на сторону сервера. Таким образом решается проблема с количеством памяти, необходимой для установки и работы мобильного приложения. Именно этот подход был выбран для организации взаимодействия приложения с обученными моделями. Общая схема взаимодействия отражена на рис. 5.1.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

Рис. 5.1. Схема взаимодействия мобильного приложения с сервером

Серверная часть разработана с использованием языка Python. В качестве основы для сервера использованы библиотеки uvicorn и FastAPI. Первая имплементирует ASGI-сервер (Asynchronous Server Gateway Interface), который можно рассматривать как связующее звено между сервером, запросами пользователей и фреймворками для обработки запросов. Она позволяет тонко настраивать параметры сервера и подключения, а также поддерживает асинхронную работу. Вторая библиотека по сути является фреймворком для создания веб-приложения. С её помощью, например, определяются доступные эндпоинты (или конечные точки, пути относительно адреса сервера, на которые можно отправить сетевой запрос) на сервере, валидируются данные и организуется работа с клиентом.

При запуске сервера инициализируются все необходимые данные: обученные модели для суммаризации и векторизации, модель Punkt для разбиения текста на предложения и токенизатор для модели mT5. На сервере определены алгоритмы суммаризации, рассмотренные в главе 4. Они выполнены в виде функций с одним входным параметром — обрабатываемым текстом, и возвращают одну строку — результат обработки.

Для принятия и обработки запросов реализованы 2 эндпоинта, для экстрактивной (/summarize/extractive) и абстрактивной (/summarize/abstractive) суммаризации соответственно. Оба из них имплементируют метод POST, принимая на вход JSON (интерпретируемый в Python как обычный словарь), содержащий текст, который необходимо суммаризовать и целевое количество предложений в случае экстрактивной суммаризации. Для согласованности интерфейсов ответ возвращается также в формате JSON с единственным полем — обработанным текстом.

Помимо предобработки данных, необходимой для их обработки нейросетью, для экстрактивной суммаризации необходимо интерпретировать заданное пользователем значение желаемой степени сокращения текста. Оно приходит на сервер в виде вещественного числа от 0.0 до 1.0. Количество предложений в суммаризации вычисляется по простой формуле: количество предложений в тексте (полученное после его разбиения) умножается на меру сокращения, заданную пользователем, и округляется до ближайшего целого числа. Полученное число становится количеством кластеров при применении метода кластерного анализа K-means.

# Заключение

В ходе проделанной работы было разработано мобильное приложение для суммаризации текстов с применением методов машинного обучения. При решении поставленной задачи были реализованы следующие этапы:

1. Проведён анализ различных подходов к суммаризации текстов. Были определены наиболее подходящие методы для решения данной задачи. Для абстрактивного подхода с применением трансферного обучения была дообучена модель mT5 на датасете новостей на 2 языках (английском и русском). Для экстрактивного подхода был разработан алгоритм на основе кластеризации векторов, полученных применением векторизатора MPNet.
2. Разработано мобильное приложение для ОС Android с использованием нативного подхода. Приложение состоит из 3 основных экранов, реализующих взаимодействие с сервером и настройку параметров суммаризации для применения разработанных алгоритмов, сканирование текста с помощью камеры устройства и просмотр истории суммаризованных текстов. В основе архитектуры приложения лежат принципы чистой архитектуры, а также паттерн MVVM.
3. Произведено внедрение разработанных моделей машинного обучения с применением клиент-серверной архитектуры. Разработан сервер с применением фреймворка FastAPI для обработки запросов клиента с применением разработанных алгоритмов суммаризации.

Результаты, полученные в ходе разработки решения поставленной задачи легли в основу опубликованных статей «Сканирование документов в системе Android с применением OpenCV» (XXX научная конференция СИТО, 2023) и «Структура Android приложения при использовании библиотек Jetpack» (XXXI научная конференция СИТО, 2024).

Код реализации решения описанных выше задач доступен на Github [17].

# Литература

1. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2021. — 352 с.
2. Сервис по сокращению текстов SMMRY — URL: https://smmry.com/ (дата обращения 22.04.2024)
3. Мобильное приложение Text Summary TLDR Summarize https://play.google.com/store/apps/details?id=com.aya.textsummarizer (дата обращения 22.04.2024)
4. Рекомендации по разработке Android-приложений от Google — URL: https://developer.android.com/modern-android-development (дата обращения 20.03.2024)
5. Распределение версий Android по пользователям Google — URL: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide (дата обращения 11.04.2024)
6. Dijkstra E. On the role of scientific thought. Selected writings on computing: a personal perspective, 1982 – P. 60–66.
7. Николаев А. В., Абрамян А. В. Структура Android приложения при использовании библиотек Jetpack // Современные информационные технологии: тенденции и перспективы развития, 2023 – С. 322–324
8. Исследование о результатах применения нового подхода к работе с камерой с использованием CameraX для сервиса Monzo — URL: https://developer.android.com/stories/apps/monzo-camerax (дата обращения 23.04.2024)
9. Jetpack Compose phases (фазы механизма Jetpack Compose) https://developer.android.com/develop/ui/compose/phases (дата обращения 24.04.2024)
10. Lun H. The automatic creation of literature abstracts. // IBM Journal, 1958 – P. 159–165.
11. Модель Punkt для токенизации текстов в предложения https://www.nltk.org/api/nltk.tokenize.punkt.html (дата обращения 23.04.2024)
12. Николаев А. В. Применение методов машинного обучения для решения задачи кластеризации текстов. Неделя науки 2022. Сборник тезисов : в 2 ч. Южный федеральный университет. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2022 – С. 735–738
13. Song K., Tan X., Qin T., Lu J., Liu T. MPNet: Masked and Permuted Pre-training for Language Understanding // 34th Conference on Neural Information Processing Systems, 2020 – P. 16857–16867.
14. Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg, V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M., Duchesnay, E. Scikit-learn: Machine Learning in Python // Journal of Machine Learning Research, Vol. 12, 2011 – P. 2825–2830.
15. Brown T., Mann B., Ryder N., Subbiah M., Kaplan J. D., Dhariwal P., Neelakantan A. et al. Language models are few-shot learners // Advances in neural information processing systems, 33 , 2020 – P. 1877–1901.
16. Xue L., Constant N., Roberts A., Kale M, Al-Rfou R, Siddhant A, Barua A., Raffel C. mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer. // Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021 – P. 483–498.
17. Репозиторий с кодом, написанным в ходе решения поставленной задачи — URL: https://github.com/vaimon/MastersDegree (дата обращения 02.06.2024)

# Приложение 1. Результаты работы алгоритмов суммаризации

Пример 1. Экстрактивная суммаризация текста

Оригинальный текст:

The Moon may be the final frontier for mankind, but what will we eat when we get there? Pasta and protein bars made out of thin air are just the beginning. Space fever is approaching at warp speed. In the next two years, Nasa plans to send astronauts back to the Moon via its Artemis programme and scientists are looking seriously into the possibility of manned deep space missions. Food is something that keeps astronauts sane, says Dr Sonja Brungs, astronaut operations deputy lead at the European Space Agency. Good food, proper food with a lot of variety, tailored to the needs of the individual astronauts is crucial for a successful deep space mission. I think people underestimate how important it is. Astronauts who are in space for just six months miss crunchiness and texture. It is really important for mental wellbeing to have a variety of textures, and especially for deep space missions, having a variety of foods to eat. In 2021, Nasa launched a Deep Space Food Challenge to discover new ways to create food in space with limited resources. Solar Foods, based in Helsinki, is one of the eight companies that has reached the challenge's final phase. Their remarkable concept: using space waste to create protein. We make food out of thin air, quite literally, says Artuu Luukanen, Solar Foods' senior vice president in Space and Defence. His company discovered an edible microbe in the Finnish countryside that grows by feeding on a mixture of carbon dioxide, hydrogen and oxygen. So what we are talking about here is really not just a food manufacturing technology for space, but something that will be an integral part of the environmental control and life support system.

Результат:

Space fever is approaching at warp speed. Good food, proper food with a lot of variety, tailored to the needs of the individual astronauts is crucial for a successful deep space mission. I think people underestimate how important it is. His company discovered an edible microbe in the Finnish countryside that grows by feeding on a mixture of carbon dioxide, hydrogen and oxygen.

Пример 2. Абстрактивная суммаризация на английском языке

Оригинальный текст:

English contains more words than any other language on the planet and added its millionth word early Wednesday, according to the Global Language Monitor, a Web site that uses a math formula to estimate how often words are created. The Global Language Monitor says the millionth word was added to English on Wednesday. The site estimates the millionth English word, Web 2.0 was added to the language Wednesday at 5:22 a.m. ET. The term refers to the second, more social generation of the Internet. The site says more than 14 words are added to English every day, at the current rate. The Million Word March, however, has made the man who runs this word-counting project somewhat of a pariah in the linguistic community.

Результат:

NEW: The World Language Monitor says the millionth word was added to English on Wednesday

Пример 3. Абстрактивная суммаризация на русском языке

Оригинальный текст:

На этих выходных в Берлине прошли крупные акции протеста против введенных для борьбы с коронавирусом ограничений. Несмотря на то что протест оказался массовым, выступления носили «преимущественно мирный характер», уверен Линдеманн. По его словам, исключением стала только ситуация у немецкого парламента. Там «несколько странных участников демонстрации попытались штурмовать бундестаг в знак протеста против правительства».

Результат:

В Берлине прошли крупные акции протеста против введенных для борьбы с вирусом ограничений