СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ....4
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ....5
ВВЕДЕНИЕ....6
1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ....8
1.1 Анализ предметной области....8
1.2 Обзор и сравнение аналогов....9
1.2.1 Проект «Автострада»....10
1.2.2 Vaisala Xweather RoadAI....11
1.2.3 Проект «Дороги России»....13
1.2.4 Сравнительный анализ....14
1.3 Требования к реализации....16
2 АРХИТЕКТУРА СИСТЕМЫ....17
2.1 Организация системы....17
2.2 Компоненты и интерфейсы....21
2.2.1 Kafka Broker....21
2.2.2 Kafka MQTT Proxy....23
2.2.3 СУБД Clickhouse....24
2.2.4 Интерфейсы....26
2.3 Мобильное приложение....28
2.4 Сервис прогнозирования....29
2.5 Сервис пользовательских запросов....29
2.6 Сервис обработки результатов....30
3 РЕАЛИЗАЦИЯ СИСТЕМЫ....32
3.1 Мобильное приложение....32

3.1.1 Структура.....32
3.1.2 Состояния.....33
3.1.3 Графики.....34
3.1.4 Накопление и отправка данных датчиков.....36
3.1.5 Карта оценок дорожного покрытия.....38
3.1.6 Конфигурации.....41
3.1.7 Уведомления и логирование.....42
3.1.8 Данные о пользователе.....43
3.2 Сервис прогнозирования.....43
3.2.1 Подготовка данных и обучение алгоритма регрессии.....44
3.2.2 Разработка сервиса.....47
3.3 Сервис пользовательских запросов.....49
3.4 Сервис обработки результатов.....50
4 ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ.....51
ЗАКЛЮЧЕНИЕ.....56
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....59
ПРИЛОЖЕНИЕ А.....63
ПРИЛОЖЕНИЕ Б.....65

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

GPS — Global Positioning System

RTOS — Real-Time Operating System

KMZ — ZIP-compressed KML (Keyhole Markup Language)

БД — База Данных

СУБД — Система Управления Базами Данных

MQTT — MQ (Message Queues) Telemetry Transport

HTTP — Hypertext Transfer Protocol

REST — REpresentational State Transfer

AMQP — Advanced Message Queuing Protocol

OLAP — OnLine Analytical Processing

RPC — Remote Procedure Call

SOAP — Simple Object Access Protocol

API — Application Programming Interface

JSON — JavaScript Object Notation

AOP — Aspect-Oriented Programming

ПО — Программное Обеспечение

FSD — Feature Sliced Design

DTO — Data Transfer Object

SDK — Software Development Kit

ACK — ACKnowledgement

NAK — Negative-ACKnowledgement

CART — Classification And Regression Trees

IIR — Infinite Impulse Response

CPU — Central Processing Unit

MSE — Mean Squared Error

MAE — Mean Absolute Error

RMSE — Root Mean Square Error

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Брокер — программный компонент, выступающий в роли посредника между частями системы.

Топик — логическая единица объединения сообщений внутри брокера или системы управления сообщениями по определенным критериям.

Репликация — механизм копирования и хранения данных в нескольких местах для повышения отказоустойчивости, доступности и/или производительности.

Партиционирование — механизм разделения данных на независимые части, называемые партициями, для повышения производительности запросов.

Шардирование — тип партиционирования, при котором разделение данных происходит по разным экземплярам баз данных.

Прокси-сервер — сервер, выступающий в роли посредника между двумя сторонами общения (например, пользователем и конечным сервером).

Горизонтальное масштабирование — процесс разбиения системы на более мелкие компоненты и их разнесения по дополнительным вычислительным ресурсам для увеличения общей производительности.

Кеширование — механизм размещения данных в местах временного хранения с, как правило, более быстрым доступом для ускорения выполнения операций чтения.

Задача регрессии — задача машинного обучения, целью которой является прогнозирование числовой непрерывной величины на основе списка входных значений.

Фича — атрибут или признак, используемый в качестве входных данных модели нейронной сети.

Семафор — примитив, синхронизирующий потоки исполнения в многопоточных приложениях, в основе которого лежит счетчик.

Инференция — процесс применения обученной модели.

ВВЕДЕНИЕ

Согласно данным из глобального доклада о безопасности дорожного движения, предоставленного Всемирной организацией здравоохранения в 2023 году, дорожные инциденты занимают 12-е место среди основных причин смертности [1]. Исследователи отмечают, что плохое состояние дорожного покрытия оказывает значительное воздействие на вероятность и серьезность дорожно-транспортных происшествий [2].

Важно понимать, что плохое состояние дорог не только угрожает безопасности участников дорожного движения, но и негативно сказывается на экономике. Исследования показывают, что несвоевременное проведение дорожных работ в 1.5-3 раза увеличивает расходы на содержание дорог [3]. Кроме того, из-за ухудшения условий движения возрастают издержки на содержание и ремонт транспортных средств, уменьшается срок эксплуатации [4], увеличивается время вождения. Ко всему прочему, развитость дорожной инфраструктуры способствует расширению рынка товаров и услуг, а также снижению их итоговой стоимости [5].

Для осуществления проверок состояния дорожного покрытия, как правило, нанимается бригада, подготавливаются передвижные лаборатории и оборудование с технически сложными приборами, например ультразвуковыми датчиками. Данный подход затратен с точки зрения времени (из-за дополнительных затрат на подготовку, транспортировку, сбор данных), людских ресурсов, а также требует наличия специализированных средств для сбора данных и диагностики. Альтернативным решением является использованием датчиков, таких как акселерометр, гироскоп и GPS, которыми обладает большинство современных устройств, например мобильные телефоны. Поскольку в текущее время мобильными телефонами обладает большая часть населения страны, это способствует увеличению как количества, так и скорости получения данных. Это особенно важно для отдаленных регионов, областях с низкой плотностью населения, а также

местах с неблагоприятными погодными условиями. Таким образом, использование мобильных устройств является целесообразным решением для сбора данных о состоянии дорожного покрытия.

Целью данной работы является обеспечение частичного отказа от использования специализированной аппаратуры, предназначенной для сбора данных, благодаря ее замене на датчики мобильных устройств и увеличению объема собираемых данных.

Для достижения цели были поставлены следующие задачи:

1. Анализ предметной области и состояния работ.
2. Разработка программной архитектуры.
3. Программная реализация средства анализа состояния дорожного покрытия.
4. Сбор данных и их обработка для получения результатов.
5. Анализ результатов и принятых решений.

Результатом выпускной квалификационной работы является система, способная собирать данные с датчиков мобильных устройств (акселерометр, гироскоп, GPS), при помощи алгоритмов машинного обучения получать оценки о состоянии дорожного покрытия, визуализировать это состояние на определенный момент времени.

Дальнейшая часть текста структурирована следующим образом. В Главе 1 произведен анализ существующих решений и технических достижений, сделавших возможным выполнение поставленных целей, а также представлены требования к реализации. В Главе 2 объяснена архитектура системы и выбранные технологии. В Главе 3 описаны алгоритмы в пределах элементов системы и приведены демонстрации. В Главе 4 произведено тестирование системы и анализ результатов.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Анализ предметной области

Большинство современных мобильных устройств с сенсорным управлением имеют схожую компонентную базу, включающую в себя такие датчики, как акселерометр, гироскоп, магнитометр, барометр, GPS. В основном все они предназначены для определения положения устройства и навигации. Рассмотрим каждый из них более подробно.

Датчик GPS (глобальной позиционной системы) — компонент, обеспечивающий определение местоположения пользователя, а именно latitude (широта), longitude (долгота) и altitude (высота). Как правило, помимо географических координат прибор позволяет узнать ошибку диапазона расположения устройства, как радиус, в пределах которого он может находиться. Этот параметр зависит от погодных условий, блокировки или отражения спутникового сигнала, используемых датчиков и прочих факторов. Средняя ошибка составляет 4.9 м [6], однако может быть уменьшена благодаря совместному использованию с акселерометром, гироскопом, магнитометром и барометром для определения перемещения устройства, а также доступа к WiFi и сотовым вышкам.

Основным назначением датчика акселерометра является измерение линейного ускорения, то есть изменения скорости движения в пространстве. Также может использоваться для определения уклонов (относительно гравитационного поля Земли). Частота дискретизация варьируется, может достигать 60 Гц.

Датчик гироскопа предназначен для измерения угловой скорости вращения вокруг своих осей. Дискретизация аналогична датчику акселерометра, может варьироваться.

Магнитометр используется для определения силы магнитного поля окружающей среды и определения сторон света.

Датчик барометра предназначен для определения высоты над уровнем моря.

Исследования показали, что датчики акселерометра, гироскопа и GPS, которыми снабжены современные мобильные телефоны, являются дешевыми средствами, позволяющими довольно точно и надежно получать данные о положении и движении устройства при различных внешних воздействиях, связанных с перемещением [6-8].

Следует также отметить, что в операционных системах, таких как Android и iOS, для которых разрабатывается мобильное приложение, невозможно получить значения датчиков акселерометра, гироскопа, GPS в моменты времени с равным интервалом, так как алгоритм получения основывается на модели событий. Также это связано с тем, что указанные выше операционные системы не являются системами реального времени (RTOS). В связи с этим, работа программного обеспечения не должна зависеть от данного предположения.

1.2 Обзор и сравнение аналогов

На момент написания дипломной работы имеется большое количество аналогов, целью которых является отказ от использования специализированных систем для отслеживания состояния дорожного покрытия. Такие аналоги, как правило, информируют об опасных участках при помощи системы меток (на картах отмечаются соответствующие точки) или оценивания дорожных отрезков.

Кроме того, уже существовали или существуют альтернативные решения, которые выполняют схожие задачи, но при этом отличаются по принципам работы, а именно: виду анализа данных датчиков, месту обработки (сервер или мобильное устройство), типам используемых датчиков и необходимости дополнительного оборудования, степени полноты и способу хранения получаемых результатов.

Далее будет приведена сравнительная характеристика наиболее популярных аналогов. Выявление преимуществ и недостатков уже существующих подходов позволяет создать более комплексное и эффективное решение. На основании этого в Подразделе 1.3 сформированы соответствующие требования к реализации.

1.2.1 Проект «Автострада»

Проект «Автострада»[1] основывается на системе отзывов. Пользователи выставляют оценки к определенным участкам трассы, на основании которых формируется итоговый балл. Помимо веб-приложения имеется мобильная версия с аналогичным функционалом.

Все последние итоговые результаты визуализируются на карте в цвете градиента от зеленого к красному, где зеленый — хороший участок дороги, красный — плохой. Помимо просмотра имеется возможность прокладки маршрута в соответствии с требованиями к качеству трассы.

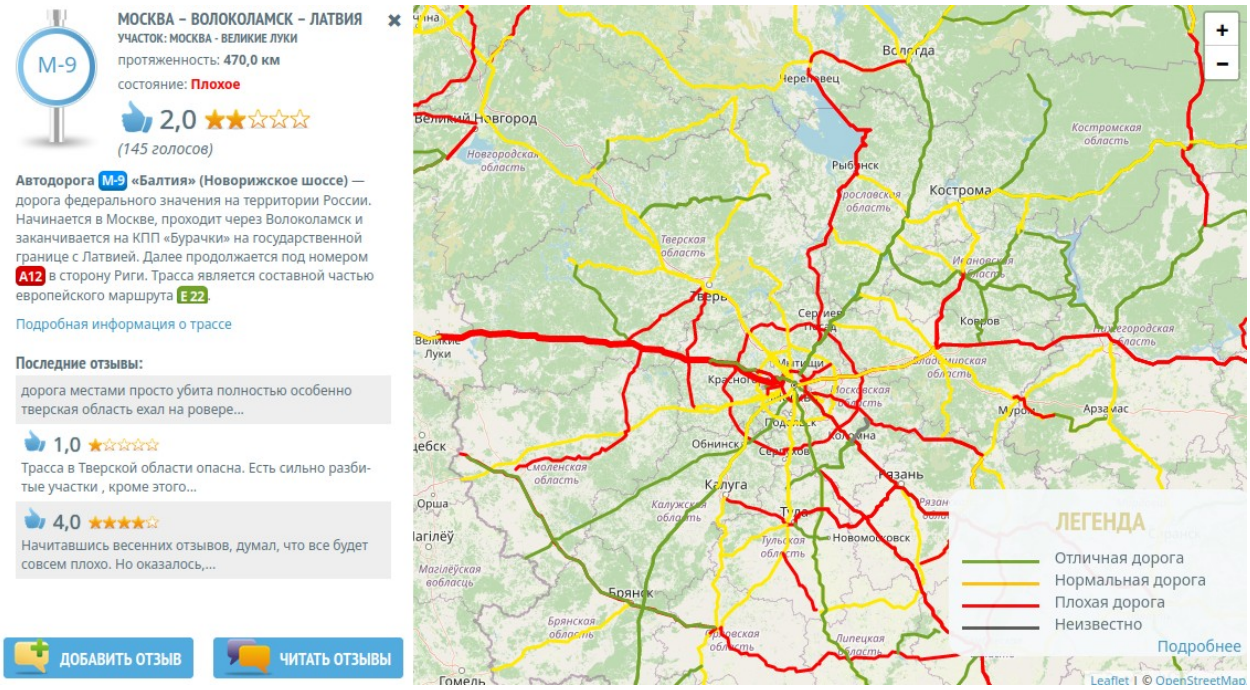


Рисунок 1 - Карта проекта «Автострада»

[1] Основная страница проекта «Автострада» [Электронный ресурс]. Режим доступа: https://autostrada.info/ru, свободный (дата обращения: 01.05.2024)

Преимуществами данного подхода являются:

1. Визуальная наглядность: обеспечивается благодаря низкой детализации.
2. Низкие требования к аппаратному обеспечению из-за небольшого объема пересылаемых данных и отсутвию сложной алгоритмов на мобильном устройстве.
3. Отсутствие погрешностей результатов из-за неточности измерений при помощи датчиков и алгоритмов обработки данных.

Основными недостатками данного решения являются:

1. Отсутствие оценок состояния дорог ниже уровня федерального, соединительного и регионального значения.
2. Необходимость ручного ввода отзывов, что может создать аварийную ситуацию на дороге, а также вероятность неточного итогового результата по причине их субъективности отзывов.
3. Невозможность просмотра деталей о местах конкретных дорожных участков, что вместе с пунктом 2 также может вводить в заблуждение.

1.2.2 Vaisala Xweather RoadAI

RoadAI[2] представляет из себя комплекс программ, предназначенных для определения фактического состояния дорожного полотна. Для сбора данных используется мобильное устройство, которое фиксируется на крепеже и направляется с сторону движения транспортного средства. В качестве источника данных используются кадры с камер телефона, которые затем на сервере обрабатываются с помощью алгоритмов компьютерного зрения. Благодаря им осуществляется детальный анализ состояния полотна и дорожных обозначений. Для визуализации результатов используются различные слои карт. При нажатии на участок открывается панель с

[2] Основная страница Vaisala Xweather RoadAI [Электронный ресурс]. Режим доступа: https://www.xweather.com/roadai, свободный (дата обращения 01.05.2024)

Рисунок 2 - Интерфейс пользователя Xweather RoadAI

возможностью просмотра записанного видео и соответствующего ему во времени графика изменения различных характеристик.

Из-за сложности производимых вычислений для получения результатов основной целевой аудиторией данного программного обеспечения являются службы дорожных инспекций. Кроме того, оно не предназначено для открытого использования.

Основные преимущества:

1. Наличие детализированных результатов на карте благодаря использованию сложных алгоритмов обработки данных.
2. Возможность просмотра характеристик прямо на видеозаписи, что позволяет удостовериться в правильности и полноте работы алгоритмов.

Основные недостатки:

1. Невозможность работы программного обеспечения с большим числом пользователей из-за перегрузки, в связи с высокой алгоритмической сложностью и объемом сохраняемых данных.
2. Снижение качества результатов при ухудшении погодных условий из-за использования алгоритмов компьютерного зрения.
3. Необходимость наличия личного кабинета, а также платы за использование программного продукта.

1.2.3 Проект «Дороги России»

«Дороги России»[3] - это некоммерческий проект, основной целью которого является предоставление информации об оценках состояния дорог. Сбор данных осуществляется приложением при помощи датчиков мобильного телефона (акселерометр, GPS). Для их обработки реализована математическая модель.

Отличительной особенностью мобильного приложения является то, что алгоритм реализован прямо на мобильном устройстве, что позволяет не отсылать данные, если устройство в данный момент не используется для их сбора.

Для получения результатов пользователь скачивает файл формата KMZ, который затем может быть импортирован в Google Maps для загрузки отдельного слоя с метриками.

На момент написания дипломной работы поддержка данного проекта приостановлена, сайт проекта недоступен, приложение удалено из Google Play.

[3] Основная страница проекта «Дороги России» [Электронный ресурс]. Режим доступа: https://web.archive.org/web/20181105021214/http://rusdorogi.ru/, свободный (дата обращения 01.05.2024)

Основные преимущества:

1. Возможность работы системы с большим количеством пользователей из-за небольшого объема данных и нагрузки на серверную часть, по сравнению с «RoadAI».
2. Независимость качества работы от погодных условий.
3. Возможность просмотра результатов для конкретных участков дорог.

Основные недостатки:

1. Повышенная нагрузка на мобильное устройство из-за внедренного в приложение алгоритма анализа дорог.
2. Ограниченность в выборе алгоритма анализа состояния дорожного покрытия по причине его реализации на мобильном устройстве, что в свою очередь влияет на точность анализа.

1.2.4 Сравнительный анализ

Ниже представлена Таблица 1 сравнения выше рассмотренных аналогов по следующим критериям:

- способность к обработке большого объема запросов (серверная нагрузка): рассчитывается на основе количества получаемых на сервере данных от обычного пользователя, сложности алгоритмов для обработки его запросов;
- использованные алгоритмов анализа данных (алгоритмы);
- детализация получаемых результатов (детализация);
- автономность работы (автономность);
- нагрузка на мобильное устройство при использовании приложения (мобильная нагрузка);
- ограничения на использование (ограничения).

Таблица 1 - Сравнительный анализ альтернатив

	«Автострада»	RoadAI	«Дороги России»
серверная нагрузка	Низкая	Высокая	Средняя
алгоритмы	-	CV (Компьютерное зрение)	Математические алгоритмы
детализация	Низкая	Высокая	Средняя
автономность	Нет	Да	Да
мобильная нагрузка	Низкая	Средняя	Высокая
ограничения	-	Необходимость фиксации устройства, хорошей погоды и освещенности	-

Рассмотрим недостатки каждого из аналогов.

Так как использование телефона во время вождения запрещено, то система должна быть автономной. Кроме того, если процесс сбора данных может быть автоматизирован, то это также может ухудшить пользовательский опыт. Данному критерию не соответствует проект «Автострада» из-за необходимости самостоятельного добавления отзывов.

Рассмотрим серверную и мобильную нагрузку. При использовании сложной логики внутри приложения, что реализовано в проекте «Дороги России», мобильное устройство быстрее теряет заряд батареи. В связи с этим, если это возможно, функционал выносится на серверную часть, которая должна обладать соразмеримым загрузке аппаратным обеспечением. Кроме того, использование видео, как в «RoadAI» значительно увеличивает нагрузку приложения и сети передачи данных, а также требует фиксации мобильного устройства на дорогу.

Так как разрабатываемое программное обеспечение рассчитано на большую аудиторию, то необходимо найти компромисс между детализацией и загрузкой серверной части. Реализация «RoadAI» не рассчитана на большое число пользователей, получающих доступ к одной базе результатов, поэтому используемый в данном решении подход противоречит идеи разрабатываемого приложения.

1.3 Требования к реализации

На основании поставленной цели и сделанных в процессе анализа выводов сформулированы следующие требования к разрабатываемому решению:

1. Мобильное приложение должно собирать данные с датчиков автономно.
2. Мобильное приложение не должно быстро расходовать заряд батареи. Одним из возможных решений этого является использование ресурсов сервера для проведения основных рассчетов, если таковые имеются.
3. Мобильное приложение должно продолжать нормальное функционирование в ситуациях, когда нет доступа к серверу (например, на трассах, где нет интернет соединения).
4. Система должна обрабатывать получаемые от мобильных приложений результаты.
5. Система должна обеспечить сохранение результатов для дальнейшего доступа пользователями к общей базе оценок состояния дорожного покрытия.
6. Система должна предоставлять доступ к полученным результатам в ходе обработки, а именно на карте должны быть отображены участки с рассчитанными оценками состояния дорожного полотна.

2 АРХИТЕКТУРА СИСТЕМЫ

Прежде чем переходить к реализации системы, следует сначала ее спроектировать. Это позволяет предварительно определить структуру системы, выбрать эффективный способ решения поставленной задачи, чтобы снизить возможные риски и ограничения до начала разработки.

Данный раздел состоит из нескольких подразделов. В первом описана общая организация системы без приведения конкретных реализаций сервисов. В следующем представлена архитектура с уже выбранными имплементациями компонентов, приведены протоколы коммуникации между элементами. Далее описаны технологии, используемые для проектирования остальных частей системы: мобильного приложения и сервисов с обоснованием используемых языков программирования и библиотек.

2.1 Организация системы

Работа мобильного приложения зависит от элементов системы, которые обеспечивают обработку данных с датчиков устройств и доступ к получаемым результатам. Такими компонентами являются сервисы, обеспечивающие прогнозирование по данным датчиков и обработку пользовательских запросов, база данных, очереди сообщений и другие вспомогательные элементы системы. Помимо самих элементов также требуется описать принцип коммуникации между ними. Данные части системы представлены на Рисунке 3.

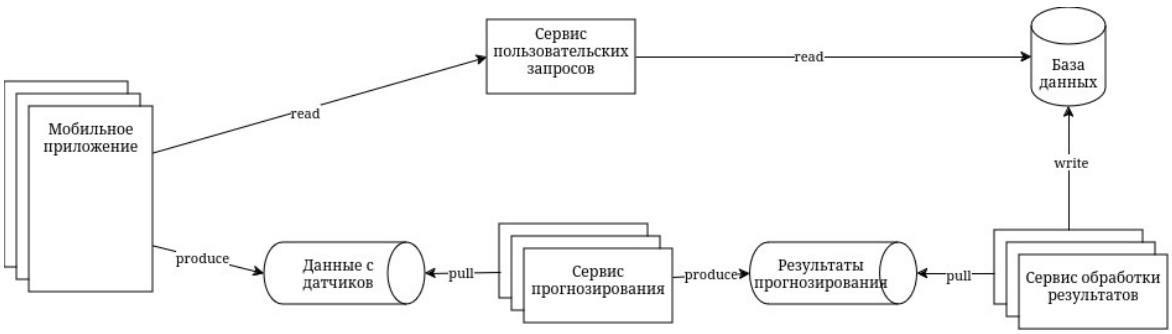


Рисунок 3 - Общая архитектура системы

Система состоит из следующих компонентов:

1. Мобильное приложение — получает данные с датчиков и визуализирует результаты. Приложение формирует сообщения, состоящие из собранных значений акселерометра, гироскопа и GPS, и записывает (produce) в очередь «Данные с датчиков». Для визуализации оценок состояния дорожного покрытия обращается к сервису пользовательских запросов, запрашивая (read) из него результаты прогнозирования в соответствии с API.
2. Очередь «Данные с датчиков — используется в качестве промежуточного буфера между мобильным приложением и сервисом прогнозирования. Мобильное приложением записывает сообщения в очередь, а сервис прогнозирования считывает их по мере поступления. Таким образом, очередь работает по принципу pull, о чем описано далее.
3. Сервис прогнозирования — на основании сообщений, записываемых в очередь «Данные с датчиков» мобильным приложением, прогнозирует оценки состояния дорожного покрытия в соответствующих переданным значениям GPS местах.
4. Очередь «Результаты прогнозирования» — буферизирует результаты прогнозирования, записанные сервисом прогнозирования для дальнейшей обработки сервисом обработки результатов. Работает аналогично очереди «Данные с датчиков».
5. Сервис обработки результатов — считывает сообщения из очереди «Результаты прогнозирования» и пакетно (сразу несколько значений одним запросом) записывает обработанные результаты в БД.
6. База данных — используется для сохранения результатов прогнозирования состояний дорожного покрытия сервисом обработки результатов и их считывания сервисом пользовательских запросов посредством соответствующих запросов.

7. Сервис пользовательских запросов — в соответствии с описываемой спецификацией (API) данный сервис обрабатывает запросы пользователей, в случае необходимости обращаясь к БД для получения состояния и дальнейшей генерации ответа.

Таким образом, согласно архитектуре системы, существует 2 основных потока данных: пользовательских запросов и обработки данных с датчиков мобильных устройств. Первый в детальном объяснении не нуждается, так как соответствует типовой клиент-серверной модели, поэтому рассмотрим второй поток обработки.

До поступления результатов в БД данные проходят несколько этапов. Очереди выступают в качестве посредников между несколькими сервисами, занимающимися обработкой данных на разных этапах, для разделения связности между ними. Благодаря этому не требуется явного указания сервисам адресов отправителей и получателей, и одновременного нахождения обеих сторон в активном состоянии [9]. Кроме того, в очереди могут накапливаться сообщения, когда сервис читатель не справляется с темпом поступления новых заявок, например в часы пик, или если он в данный момент недоступен, а также по причине недетерминированности времени обслуживания и моментов поступления заявок. Это обеспечивает снижение времени ожидания сервисов до момента пока появится возможность отправки заявки на следующий этап обработки и уменьшение числа отказанных заявок (если в случае неактивности сервиса, к которому происходит обращение, заявка отклоняется), а также увеличивает загрузку системы в целом, но при этом увеличивает среднее время обработки заявки (из-за наличия дополнительных этапов чтения-записи с очередями и уменьшения числа отказов) [10]. Кроме того, в случае отклонения заявки на промежуточном этапе это приводит к невозможности корректной обработки ошибок на предыдущих этапах (без дополнительных потоков обработки ошибок). Так как мобильное приложение при отправке данных с датчиков не

нуждается в получении ответов от сервисов, и длительность обработки данных до поступления в БД, по принципу построения архитектуры системы, не важна, то использование очередей является обоснованным и рациональным.

Очереди могут работать как по механизму pull (читатель периодически обращается к очереди, узнавая, поступили ли новые данные), так и push (очередь немедленно передает сообщения читателям или уведомляет их о приходе новых сообщений). В данной системе более предпочтительно использование механизма pull для очередей, поскольку данные датчиков не требуют мгновенной обработки [11], и нет необходимости в группировке сообщений на большое количество топиков. Кроме того, в pull очередях, как правило, реализован более эффективный алгоритм обработки сообщений пачками (сразу несколько сообщений за один запрос), что снижает нагрузку на сеть. Следствием использования pull очередей является необходимость постоянного опроса читателями о приходе новых сообщений, что приводит к дополнительной нагрузке на систему. Для снижения затрачиваемых ресурсов на выполнение данной операции, может быть использован механизм длинных опросов (long polling).

Разделение этапов на сервис прогнозирования и сервис обработки результатов обосновано различными зонами ответственности. Так сервис прогнозирования только считывает данные из очереди и получает результаты состояния дорожного покрытия в определенных точках, а сервис обработки результатов занимается эффективной вставкой соответствующих результатов в БД и постобработкой. Кроме того, это упрощает поддержку системы, так как обновление сервисов происходит независимо друг от друга, так например в случае перезапуска первого, не будут сброшены буферы второго. Также сервис обработки результатов может быть заменен на уже существующие реализации, что рассмотрено в следующем подразделе.

2.2 Компоненты и интерфейсы

В данном подразделе приведены и обоснованы используемые сервисы и интерфейсы. На Рисунке 4 представлена более детализированная архитектура, по сравнению с указанной в предыдущем подразделе, включающая конкретные реализации соответствующих использованных сервисов.

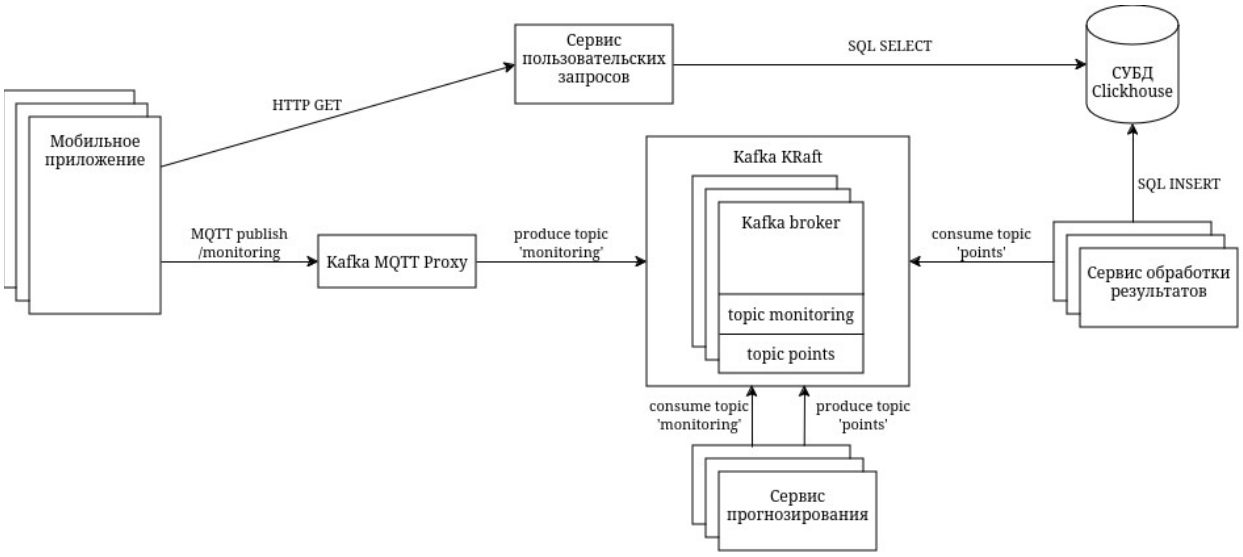


Рисунок 4 - Архитектура системы с использованными сервисами

Рассмотрим каждый из них по отдельности.

2.2.1 Kafka Broker

Подразделом выше было установлено, что очереди, работающие по механизму pull, являются более подходящими для использования в данном приложении, поскольку ожидается высокая нагрузка на сеть, связанная с большим количеством сообщений. Рассмотрим список наиболее популярных альтернатив и выберем подходящую реализацию.

1. Apache Kafka — брокер сообщений, работающий по принципу pull. Имеет высокую пропускную способность, позволяет сохранять данные топика в течение времени удержания (retention time), может осуществлять сжатие данных, репликацию, в том числе и с учетом географического местоположения, и партиционирование. Также

поддерживает горизонтальное масштабирование при помощи Zookeeper или алгоритма KRaft. Помимо брокера сообщений существует развитая экосистема с большим количеством различных сервисов и поддерживаемых языков.

2. Apache Pulsar — брокер сообщений, который работает по механизму push. В отличие от Kafka, архитектура системы Pulsar разделена на уровни обслуживания и хранения, которые могут расширяться независимо друг от друга. Также поддерживается репликация, в том числе с учетом местоположения, и автоматическое масштабирование с использованием Zookeeper. Сообщество разработчиков расширилось недавно, поэтому в Pulsar в настоящее время еще не настолько, как в Kafka, развитая экосистема и подробная документация.
3. RabbitMQ — брокер сообщений, основанный на push. Значительно более прост в настройке по сравнению с Pulsar и Kafka, имеет более низкое время ответа по сравнению с ними, однако при высоких нагрузках она значительно деградирует. Порог пропускной способности на порядок ниже по сравнению с Apache Kafka. Также поддерживает репликацию данных.
4. NATS — брокер сообщений, работающий по принципу push. Поддерживает горизонтальное масштабирование. В отличие от первых двух альтернатив NATS имеет значительно более низкую задержку ответа, но при этом плохо работает с сообщениями, приходящими с высокой интенсивностью, и имеет более низкую пропускную способность. Репликация данных по умолчанию не поддерживается.

Поскольку ключевыми критериями являются возможность горизонтального масштабирования, сохранения данных в течении заданного промежутка времени и производительность системы, то в качестве

подходящих решений были выбраны Apache Kafka и Apache Pulsar. На основании тестов производительности [12-13] невозможно точно определить, какая из имплементаций имеет более высокую пропускную способность, поскольку при разных тестовых сценариях они показывают различные результаты. Ключевым фактором для выбора подходящего решения являлась полнота документации. В Kafka она подробнее, к тому же Kafka имеет более развитую экосистему. Так в случае Kafka существует сервис Kafka MQTT Proxy, а для Apache Pulsar альтернативы нет, поэтому брокером сообщений была выбрана Apache Kafka.

2.2.2 Kafka MQTT Proxy

Использование одних и тех же протоколов на стороне пользователя и серверной части не всегда является рациональным решением, например, как в данном случае, использование Kafka на мобильных устройствах. Для решения данной проблемы могут использоваться различные прокси-серверы, которые переводят одни протоколы в другие, если это, конечно, возможно для заданных протоколов. Рассмотрим возможные альтернативы протоколу Kafka для использования на стороне пользователя, а именно MQTT, HTTP REST и AMQP, а также почему был выбран именно MQTT.

Протокол Kafka не подходит для данного решения, так как не предназначен для большого числа отправителей (producers), требует наличия стабильного интернет соединения с низкой задержкой, что не согласуется с требованиями, поскольку система должна работать с большим количеством мобильных устройств, интернет которых не является постоянным [11].

В свою очередь MQTT является более легковесным протоколом по сравнению с AMQP, HTTP REST и Kafka, оптимизированным под некачественное интернет соединение без необходимости обеспечивания гарантии доставки. В отличие от него, HTTP REST работает по принципу клиент-сервер, и требует устанавливать двустороннее соединение, а AMQP

предоставляет дополнительный функционал, который не нужен в данном приложении. Именно поэтому предпочтение было отдано именно MQTT.

Поскольку, из-за выбора MQTT на стороне отправителя, на разных сторонах общения существуют разные протоколы, то требуется добавить функционал, который будет обеспечивать перевод данных из MQTT в формат сообщений Kafka. Для решения данной проблемы рассмотрим наиболее популярные подходы, предполагающие использования одного из следующих вариантов:

1. Confluent Kafka MQTT Proxy — реализация от разработчиков Confluent Kafka.
2. MQTT брокер с собственным приложением.
3. MQTT брокер с MQTT Connector.

Первый подход, в отличие от остальных, не требует лишней пересылки данных через брокера, поэтому является более эффективным решением. Таким образом, был выбран Confluent Kafka MQTT Proxy.

2.2.3 СУБД Clickhouse

Выбор СУБД зависит от многих причин, которые уникальны для различных систем. Для разрабатываемой системы рекомендуется, чтобы СУБД имела следующие технические характеристики:

1. Возможность работы с большими объемами данных, а именно с огромным числом записей состояния дорожного покрытия в определенных точках.
2. Ориентированность на OLAP — пользователю необходимо получать значения состояния дорожного покрытия в определенных географических зонах, а также требуется возможность обеспечения агрегирования запросов. То есть, нормальными запросами является выборка большого числа данных согласно определенным условиям,

над которыми далее потенциально возможно осуществление агрегации.

3. Наличие асинхронной репликации — необходимо обеспечить отказоустойчивость данных, и в конечном счете реплики должны быть в согласованном состоянии.
4. Недорогостоящее добавление новых полей в коллекцию, так как при расширении системы в процессе дальнейших исследований могут появиться дополнительно расчитываемые метрики.
5. Возможность динамического сжатия и удаления старых данных для уменьшения занимаемого дискового пространства старыми или ненужными записями.
6. Эффективные операции вставки больших объемов структурированных данных с точки зрения скорости и затрат ресурсов системы, поскольку требуется высокая пропускная способность.
7. Возможность шардирования данных для увеличения пропускной способности и распределения объема данных по нескольким БД.

Также предпочтительно, чтобы программное обеспечение было открытым и не было ограничений на возможность осуществления горизонтальное масштабирование.

На основании описанных выше требований и анализа альтернатив был выбран Clickhouse. Согласно документации он обладает всеми описанными выше техническими возможностями, которые позволяют эффективно работать с сохраненными значениями состояний дорожного покрытия: осуществлять вставку и производить выборку.

2.2.4 Интерфейсы

Согласно архитектуре приложения в системе существует 2 основных потока данных: пользовательские запросы к API и обработка данных с датчиков. Рассмотрим формат передаваемых сообщений на каждом из них.

Для пользовательских запросов наиболее подходящим является использование HTTP REST. В качестве альтернатив также рассматривались SOAP и RPC. В конечном результате HTTP REST был выбран по следующим причинам:

1. Сервис пользовательских запросов не хранит сессии (stateless), для получения состояния используются запросы к СУБД. Это позволяет легко осуществлять горизонтальное масштабирование.
2. Для запросов типа GET может быть реализовано кеширование на стороне сервера или прокси.
3. Структура запросов, используемых в HTTP REST, является более простой по сравнению с RPC и SOAP.
4. Большое число библиотек на разных языках программирования поддерживают HTTP REST. Кроме того, существует множество различных утилит для упрощения разработки и тестирования.

Благодаря возможности реализации кеширования и сжатия HTTP запросов, использование форматов сериализации, отличных от JSON, вероятно, не даст существенного преимущества в работе системы. В случае необходимости, формат может быть заменен на более эффективный по памяти и скорости конвертации, например messagePack, protobuf или avro. На архитектуру это решение никак не повлияет, поскольку коммуникация осуществляется между пользователем и конечным сервисом.

Рассмотрим поток передачи данных с датчиков. Выбор протоколов был осуществлен выше, теперь требуется произвести выбор формата сериализации сообщений. Так как обработка для получения результатов была вынесена за пределы мобильного приложения для снижения нагрузки на

устройства, то увеличился сам объем передаваемых данных. По этой причине требуется выбрать такой формат, который бы снизил затраты по пропускной способности путем уменьшения количества передаваемых байтов и нагрузку на их конвертацию.

MQTT и Kafka позволяют передавать данные в бинарном формате. Кроме того, MQTT по умолчанию не имеет алгоритмов сжатия. С учетом этого, выберем наиболее подходящий формат сериализации из наиболее популярных решений:

1. Avro — бинарный формат, сравнение на скорости и размеру сообщений с Protobuf описано в пункте 2. Позволяет поддерживать совместимость с другими версиями при помощи схем, описываемых в человекочитаемом формате.
2. Protobuf — бинарный формат, согласно тестам с Avro [13] чуть менее компактен, но приблизительно в 2 раза быстрее сериализуем. Также позволяет поддерживает совместимость. Рекомендуется, когда требуется скорость сериализации и десереализации.
3. JSON — текстовый формат, который значительно уступает остальным альтернативам в скорости сериализации и компактности сообщений.
4. MessagePack — имеет больший размер сообщений по сравнению с Protobuf, поскольку передается название имен полей.

Другими альтернативами являются FlatBuffers (для больших сообщений), Thrift (чуть медленнее, чем Protobuf).

На основании плюсов и минусов был выбран Protobuf, поскольку требуется быстрая сериализации и десериализация для обработки большого числа сообщений, несмотря на чуть больший размер по сравнению с Avro.

Для определения случаев несовместимости схем для Avro и Protobuf может быть использован отдельный сервис Schema Registry, однако в

разрабатываемой системе он не используется, что никак не влияет на архитектуру, если понадобится его добавление.

Таким образом, для потока пользовательского API используется HTTP REST с сериализацией JSON, а для датчиков на каждом из этапов при передаче сообщений через брокера сообщений — Protobuf.

2.3 Мобильное приложение

Мобильное приложение используется пользователем для отправки данных с датчиков и просмотра результатов. Так как приложение разрабатывается для Android и iOS, то использование языков программирования Java или Kotlin и Swift соответственно для реализации двух независимых мобильных приложений не является рациональным. Для этого лучше использовать кроссплатформенное решение. Таковым является Flutter — он позволяет на языке Dart писать реализации не только мобильных, но и Windows, Mac или веб приложений.

В ситуациях, когда невозможно, по причине несовместимости интерфейсов в операционных системах и библиотеках, использовать кроссплатформенные реализации, Flutter позволяет реализовать код платформозависимо [14]. Несмотря на это, на момент написания, имелось множество библиотек, в том числе для работы с датчиками (sensors_plus, geolocator), сериализацией (protobuf), файловой системой (path, path_provider, shared_preferences), графиками (syncfusion_flutter_charts), сетью (http, mqtt_client) и картами (flutter_map), что позволило реализовать требуемую функциональность без использования платформозависимого кода. Кроме того, чтобы ускорить процесс верстки приложения, можно использовать библиотеки компонентов, например cupertino, material.

Таким образом, в качестве кроссплатформенного решения, был выбран фреймворк (framework) Flutter.

2.4 Сервис прогнозирования

Основным назначением данного сервиса является чтение данных датчиков из топиков monitoring Kafka, их обработка, прогнозирование состояния дорожного покрытия, запись результатов в топик points Kafka. Язык программирования должен иметь соответствующие библиотеки для работы с матрицами и массивами данных (фильтрация, интерполяция, агрегирование), нейронными сетями и деревьями решений, работой с Kafka и сериализацией Protobuf. Таким языком программирования является Python. Несмотря на то, что язык является скриптовым, следовательно имеет низкую скорость работы и потребляет много памяти, основная логика (сериализация, работа с массивами и матрицами, сетью) осуществляется внутри библиотек, код которых предварительно оптимизирован и скомпилирован под конкретные платформы, соответственно выполняется быстро, что обеспечивает хорошую скорость работы и низкое потребление памяти.

Таким образом, по указанным выше причинам, для реализации сервиса прогнозирования был выбран язык Python.

2.5 Сервис пользовательских запросов

Сервис пользовательских запросов является микросервисом, основной задачей которого является выборка точек из базы данных согласно определенному критерию. Для реализации данного приложения был выбран язык Go по следующим причинам:

1. Язык является стабильным, при этом достаточно новым. В нем мало языковых конструкций, поэтому написанный код легко поддерживать.
2. Благодаря таким конструкциям, как горутины и каналы, довольно просто реализовывать многопоточные приложения. К тому же, горутины являются зелеными потоками, поэтому переключение контекста между ними выполняется быстрее, чем если бы

аналогичные действия выполняла операционная система (компилятор лучше знает, когда переключать потоки, поскольку имеет больше информации о приложении; и сохранение с переключением контекста внутри пользовательского пространства выполняется быстрее).

3. Язык является компилируемым, со строгой статической типизацией, явной обработкой исключений и отсутствием AOP как такового, благодаря чему значительная часть ошибок обрабатывается на этапе компиляции.
4. В Go существуют встроенные средства профилирования, отладки и отслеживания состояний гонок данных, поэтому процесс оптимизации и нахождения уязвимых мест упрощается.
5. Для данного языка программирования реализовано огромное число библиотек, например для написания REST приложений можно использовать библиотеки chi, gorilla-mux, gin, net/http и т. п, для подключения к СУБД Clickhouse clickhouse-go, ch-go, что предоставляет гибкость и адаптивность под специфические требования к проекту.
6. Язык программирования является довольно быстрым, потребляет мало памяти, имеет автоматическую сборку мусора.

Таким образом, с учетом перечисленных выше причин и наличия необходимых библиотек, для разработки требуемого микросервиса был выбран язык Go.

2.6 Сервис обработки результатов

Данный сервис читает данные из топика Kafka и обеспечивает вставку в СУБД Clickhouse. Альтернативными решениями использования данного сервиса являются:

1. Настройка Kafka Connector для работы с Protobuf и Clickhouse. Для реализации данного подхода и работы с Protobuf требуется настройка Schema Registry, однако выше было решено не использовать его по причине отсутствия необходимости реализации обратной совместимости на момент разработки.
2. Применение Clickhouse Engine. Чтобы это реализовать требуется указать список брокеров Kafka при создании таблицы (очереди) и путь до файла со схемой Protobuf. Данное решение не является удобным, поскольку при смене адресов брокеров или схемы Protobuf требуется пересоздание таблицы (очереди). Кроме того, этот подход ограничивает возможность добавления обратной совместимости схем в дальнейшем из-за необходимости явного указания файла.

На основании изложенного было принято решение написать микросервис, работающий аналогично Kafka Connector, но с ограниченными возможностями и отсутствием необходимости указания Schema Registry. Так при переходе на Kafka Connector не потребуется редактирование схемы базы данных.

Данный сервис был разработан на Go по тем же причинам, что указаны при выборе языка для серверного приложения.

3 РЕАЛИЗАЦИЯ СИСТЕМЫ

В данной главе описаны детали реализации разработанных сервисов и мобильного приложения: структура, используемые библиотеки и алгоритмы работы, а также приведены демонстрации работоспособности элементов и фрагменты реального кода.

Структура сообщений очередей представлена в Приложении А.

3.1 Мобильное приложение

3.1.1 Структура

Структура приложения была сформирована таким образом, чтобы слои частично соответствовали архитектурной методологии FSD [15] для упрощения модификации кода в случае необходимости изменений или добавления новой функциональности. Разделение на части (slices) и сегменты (segments) не производилось по причине размера приложения. Дополнительно к базовой модели архитектуры был добавлен слой состояния. В итоге, код приложения делится на следующие уровни (код одного уровня не использует части, определенные выше; список приведен сверху-вниз):

1. app — тема приложения, маршруты и корневой компонент.
2. pages — страницы, по которым осуществляется маршрутизация.
3. widgets — компоненты, используемые страницами.
4. state — состояния приложения, при изменении которых могут вызываться определенные события, например — отрисовка, перерисовка и условная визуализация компонентов, обновление других состояний.
5. features — вспомогательные функции и компоненты, используемые слоями выше.
6. gateway — функции взаимодействия с локальным хранилищем и сетью. Вынесено из entities по причине отсутствия сегментов (segments) FSD.

7. entities — классы (model, DTO) со вспомогательными функциями.
8. shared — переиспользуемые функции, сгенерированный код (например, классы Protobuf).

3.1.2 Состояния

Библиотека компонентов регистрирует события, например — пользовательское нажатие на текстовое поле, при возникновении которых обновляется состояние компонентов. Помимо пользовательских, в системе существуют внешние события, такие как получение данных датчиков от системы, загрузка файла конфигурации приложения и таймеры. В связи с этим, в приложении дополнительно был создан слой state, отвечающий только за данную возможность. Таким образом, при изменении состояния для компонентов, которые были на него подписаны, осуществляется вызов соответствующих функций обработчиков. Данный функционал реализуется стандартным SDK Flutter при помощи класса ChangeNotifier и вызова метода read или watch у объекта BuildContext.

Помимо обновления компонентов, при реализации потребовалось обновлять другие состояния. Для того чтобы не дублировать код, добавляя состояния в иерархию компонентов и указывая в каждом из них необходимые обработчики, была использована библиотека provider. Реальный пример представлен в Листинге 1.

Листинг 1 - Регистрация зависимостей состояний

```
class App extends StatelessWidget {
  const App({super.key});

  @override
  Widget build(BuildContext context) {
    ...
    return MultiProvider(providers: [
      // Configuration
      ChangeNotifierProvider(create: (_) => ConfigurationState()),
      // Sensors
      ChangeNotifierProxyProvider<ConfigurationState, GpsState>(
        create: (_) => GpsState(),
        update: (_, ConfigurationState value, GpsState? previous) {
```

Продолжение Листинга 1 - Регистрация зависимостей состояний

```
          previous ??= GpsState();
          previous.updateConfiguration(value.configurationData);
          return previous;
        },
      ),
      ...
    ], child: app);
  }
}
```

На листинге показано, что был зарегистрирован объект ConfigurationState, отвечающий за конфигурирование приложения. Далее создан объект GpsState, который при изменении состояния объекта конфигурации вызывает метод updateConfiguration, таким образом реагируя на изменение.

Помимо ConfigurationState и GpsState в системе существуют состояния:

- AccelerometerState — акселерометр,
- GyroscopeState — гироскоп,
- AccelerometerWindowState — окно записей акселерометра, для визуализации графиков,
- GyroscopeWindowState — аналогично AccelerometerWindowState, но для гироскопа,
- ChartState — триггер на перерисовку графиков,
- SensorTransmitter — буфер отправки данных датчиков на сервер.

3.1.3 Графики

Данные графиков отображаются во вкладке «Sensors» мобильного приложения. На странице указаны графики акселерометра и гироскопа (по оси абцисс — время, по оси ординат — ускорение линейной скорости и угловая скорость соответственно) и последние значения GPS (широта и долгота). На графиках могут быть включены и выключены ломаные линии осей X, Y и Z. Снимки страниц приведены на Рисунке 5.

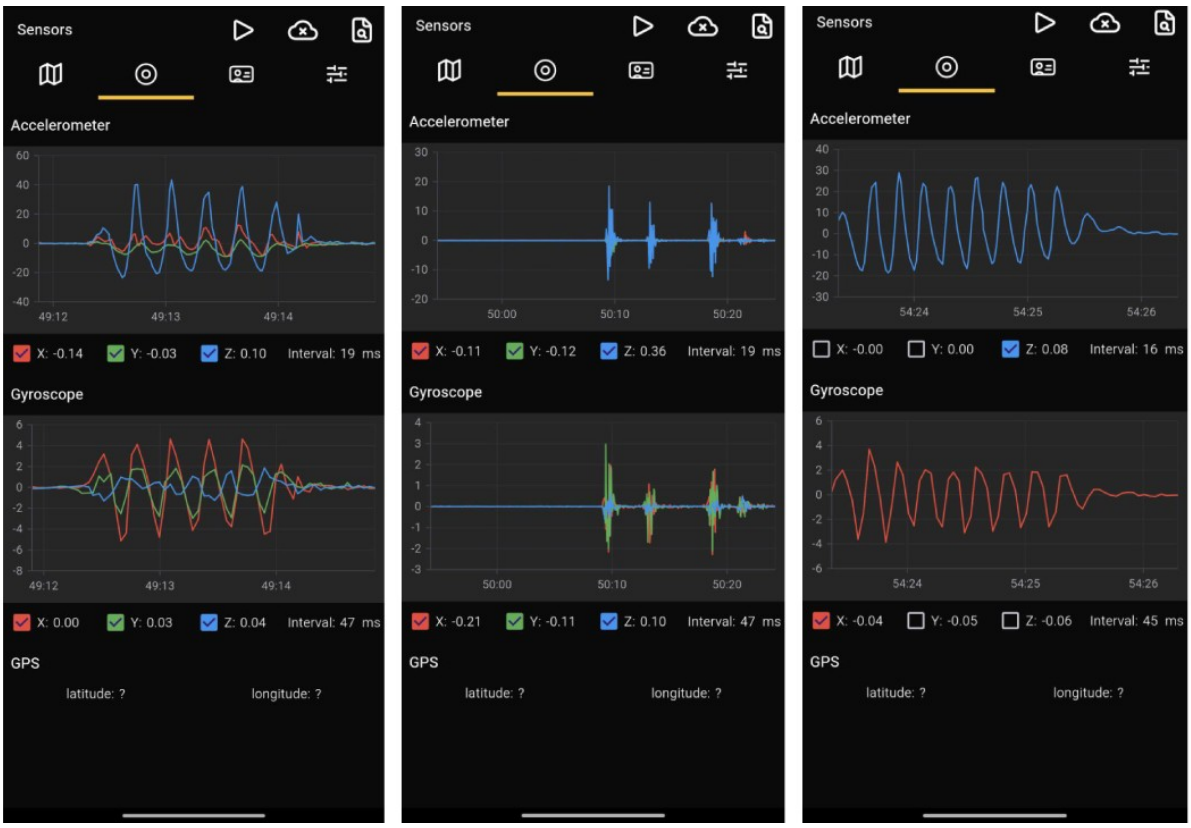


Рисунок 5 - Страница «Sensors» мобильного приложения

Графики на данной странице позволяют пользователю наблюдать изменения значений акселерометра, гироскопа, а также последние значения GPS. Это может быть полезно для проверки работоспособности датчиков. Первоначально даннай экран был разработан, чтобы отладить и протестировать работу программного обеспечения, а также исследовать как изменяются значения в зависимости от характера внешнего воздействия: ускорения и поворотов автомобиля, наездов на различные неровности, такие как ямы, кочки и уклоны.

Значения для графиков представлены связными списками, которые хранятся в объектах AccelerometerWindowState и GyroscopeWindowState. Перед добавлением нового элемента удаляются значения с начала списка, разница времен между вставленным и текущим временем которых превосходит длительность окна. По прошествии указанного времени в объекте ChartState происходит изменение состояния, и компоненты, подписанные на это событие, а именно графики, обновляются.

3.1.4 Накопление и отправка данных датчиков

Перед тем, как отправить данные датчиков, требуется предварительно их накопить. Для этого используется объект SensorTransmitter. Данные на сервер не отправляются до тех пор, пока разность времен между первой и последней записью GPS не привысит заданное значение. После соблюдения этого условия вызывается функция отправки данных, очищается список с накопленными записями, добавляется последнее значение GPS. Если не выполнять последнее действие, будет пропущено большое число записей до прихода следующего события GPS.

Перед отправкой проверяется соединение с брокером MQTT. Новое соединение устанавливается в случае, если его еще не было, или поменялся адрес назначения. После этого данные о пользователе и собранных значениях датчиков сериализуются в сообщение Monitoring формата Protobuf, схема которого указана в Приложении А. При неудачной отправке выбрасывается исключение, и данные сохраняются локально, чтобы в дальнейшем их можно было послать повторно.

Для этого при первом открытии приложения создается индекс, в котором указываются смещения чтения (номер строки), информация об окончании файла и путь до него. Пример файла с индексами представлен в Листинге 2.

Листинг 2 - Файл с индексами

```
{
  "indexes":[
    {
      "location":"2024-03-09-07:19:07.json",
      "position":0,
      "pending":0,
      "is_end":false
    }, ...
  ]
}
```

Файл с данными создается при каждом открытии приложения. Данные, подлежащие отправке, сериализуются в формат JSON, при этом каждая запись формируется с новой строки, что позволяет считывать их друг за другом. Благодаря тому, что файлов может быть несколько, неотправленные с предыдущего запуска записи также могут быть перепосланы. Сохраняемые записи могут быть сконвертированы в сообщения для отправки, структура которых указана в Приложении А, без потерь.

Для более компактного хранения данных может быть использован другой алгоритм сериализации или способ упаковки, например SQLite (библиотека sqflite). Однако на момент разработки было принято решение в сторону JSON для удобства переноса данных датчиков через файловую систему без использования сервера.

При возникновении события «включения сети» начинается переотправка данных. Сначала осуществляется загрузка файла индекса, в цикле группами указанного размера (в коде 50), и для того чтобы не перечитывать файл заново после каждого чтения одной записи, происходит считывание и десериализация данных. В процессе считывания группы обновляется файл индекса, а именно суммарное значение pending увеличивается на число прочитанных записей, и для полностью прочитанных файлов поле isEnd становится истинным (в случае чтения нескольких файлов для каждого поля position каждого из индексов значение увеличивается на количество прочитанных в этом файле новых записей). После чтения начинается отправка прочитанных записей, при успешной отправке (ACK) которых значения поля position соответствующее число прочитанных элементов, аналогично, как увеличивается pending. При возникновении ошибки (NACK) дальнейшая отправка прерывается, значение position как и выше увеличивается на K записей, но pending и isEnd возвращаются на этап, как если бы было прочитано только K записей. В процессе описываемой обработки (ACK, NACK) удаляются файлы, у которых в индексе значение

position равно pending и isEnd истинно, поскольку эти данные были прочитаны до конца. Для файла с данными текущей сессии поле isEnd не может быть истинным. Чтобы избежать конкурентного доступа к списку индексов, в коде используются блокировки (lock).

Подводя итог, при наличии соединения с сервером и отсутствии ошибок, данные отсылаются на него, в противном случае они сохраняются локально, чтобы в дальнейшем их можно было послать повторно.

3.1.5 Карта оценок дорожного покрытия

Результаты прогнозирования могут быть просмотрены во вкладке «Map» основной страницы. Для визуализации был реализован компонент MapWidget. Рассмотрим его более подробно.

Сам компонент состоит из двух частей:

1. FlutterMap — карта из библиотеки flutter_map.
2. MapControlsWidget — компонент для управления параметрами визуалиции.

На FlutterMap отображается несколько слоев:

1. TileLayer — визуализирует непосредственно карту. В качестве провайдера используется OpenStreetMap’s Standard, поскольку его использование бесплатно, и он имеет относительно низкую задержку обновления (до 1 дня).
2. CurrentLocationLayer — отмечает текущее местоположение пользователя на карте. Предоставляется библиотекой flutter_map_location_marker.
3. MapPointsLayer — компонент, реализованный для визуализации состояния дорожного покрытия. Далее будет приведено описание принципа его работы.

Компонент MapPointsLayer принимает на вход функцию loadFunction с четырьмя параметрами: адрес сервера, масштаб (zoom), номер тайла по

горизонтали и вертикали. Если при данном масштабе ячейка еще не была загружена, то вызывается функция, возвращающая список загруженных с сервиса точек.

Принцип загрузки точек MapPointsLayer схож с TileLayer: для выбора ячеек, необходимых для загрузки из объекта типа Camera, полученного из BuildContext, компонент получает видимые границы широты и долготы, а также текущий масштаб, после чего градусы преобразуются в номера ячеек в соответствии с масштабом, округленным вниз (алгоритм преобразования координат описан в подразделе «Сервис пользовательских запросов»). В результате получается видимый прямоугольник с ячейками. Затем, для каждого такого прямоугольника, вызывается функция loadFunction.

Чтобы снизить нагрузку на сервер при обновлении компонента MapPointsLayer, например при изменении местоположения пользователя, движении и измении масштаба камеры, необходимо кешировать уже загруженные ячейки. Для этого был добавлен словарь cachedPoints, обеспечивающий хранение уже загруженных ячеек, и список visibleMarkers, предназначенный для отображения всех точек текущего масштаба на карте.

Листинг 3 - Коллекции для загруженных точек

```
final Map<int, Map<Pair<int, int>, List<Marker>>> cachedPoints = HashMap();
final List<Marker> visibleMarkers = [];
```

До тех пор пока не изменится функция загрузки, а она может измениться только при изменении типа визуализации и параметров выбора точек, при переотрисовке компонента, если ячейка уже загружена, будут использоваться кешированные данные, в противном случае вызовется loadFunction, и результат будет записан в кеш.

Список visibleMarkers сбрасывается при изменении масштаба и заполняется точками текущего масштаба, иначе дополняется новыми загруженными точками.

Значение прогноза состояния дорожного покрытия находится в интервале от 0 до 1. Для отображения цветов, в зависимости от этого значения, используется функция градиента, представленная в Листинге 4.

Листинг 4 - Функция градиента точек прогноза

```
Color _getMarkerColor(PointResponse point) {
  final p = point.prediction;
  final r = min(255, 510 - 510 * p).floor();
  final g = min(255, 510 * p).floor();
  const b = 0;
  return Color.fromRGBO(r, g, b, 1);
}
```

Функция loadFunction пересоздается при изменении параметров, указываемых в компоненте MapControlsWidget, а именно типа selectedType, временных границ rawBegin, rawEnd и порогов прогнозируемых величин rawPredictionMin, rawPredictionMax (Листинг 5). При изменении функции сбрасываются кешированные и видимые точки.

Листинг 5 - Функция loadFunction

```
LoadFunctionT _getLoadFunction() {
  switch (selectedType) {
    case "Raw":
      return (apiUrl, z, x, y) async {
        final res =
            await getPointsBeginEnd(apiUrl, z, x, y, rawBegin, rawEnd);
        return res
            .where((e) =>
                e.prediction >= rawPredictionMin &&
                e.prediction <= rawPredictionMax)
            .toList();
      };
    default:
      return (apiUrl, z, x, y) => Future.value([]);
  }
}
```

Снимки экрана «Map» мобильного приложения приведены в разделе «Тестирование и анализ результатов» во избежание дублирования.

3.1.6 Конфигурации

В мобильном приложении, в частности на экране «Options», реализована возможность изменения различных конфигурационных значений (Рисунок 6), таких как:

- включение и выключение графиков акселерометра, гироскопа, GPS;
- длительность временного окна;
- время обновления графиков;
- включение и выключение границ точек на карте;
- радиус точек на карте;
- фильтр дистанции для датчика GPS;
- адреса API, брокера данных датчиков;
- размер буфера отправки в секундах;
- включение и выключение сбора данных датчиков, отправки данных на сервер (виджеты на верхней панели);

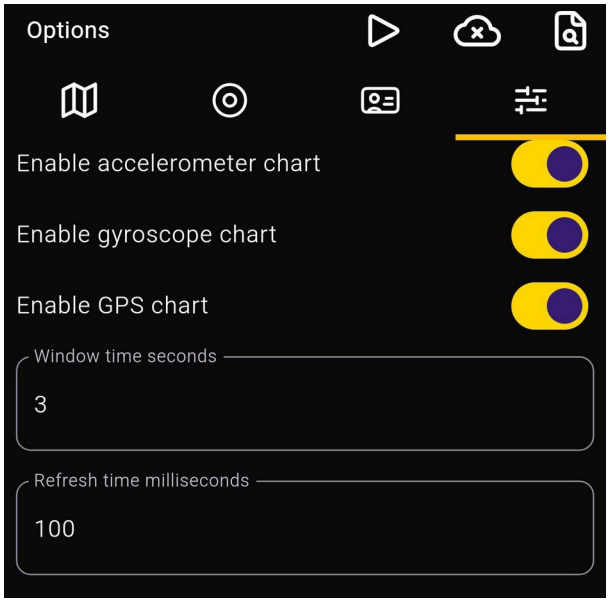


Рисунок 6 - Часть экрана «Options»

При изменении значений вызывается соответствующая функция ConfigurationState, с помощью библиотеки shared_preferences сохраняется JSON сериализованный объект ConfigurationData, и далее уведомляются подписанные на это событие компоненты и другие состояния. Благодаря

этому механизму, можно изменять конфигурацию в процессе работы приложения.

Во время загрузки приложения, в случае успеха считывания сериализованного объекта ConfigurationData, происходит замена состояния по умолчанию на прочитанное, и генерируется аналогичное событие.

3.1.7 Уведомления и логирование

Уведомления реализованы через библиотеку flutter_local_notifications. В приложении уведомление создается в случаях, когда не предоставлены требуемые права доступа, происходит разрыв соединения с сервером, и не включена геолокация. Пример представлен на Рисунке 7.

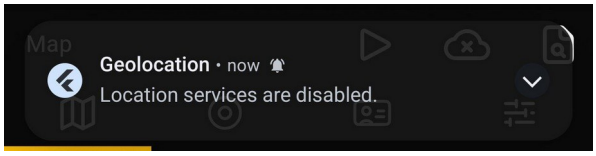


Рисунок 7 - Уведомления

Для упрощения разработки, поиска ошибок и получения обратной связи от пользователей, в мобильное приложение добавлен механизм логирования. Страница логирования открывается при нажатии на виджет сверху справа. На Рисунке 8 представлен список логов.

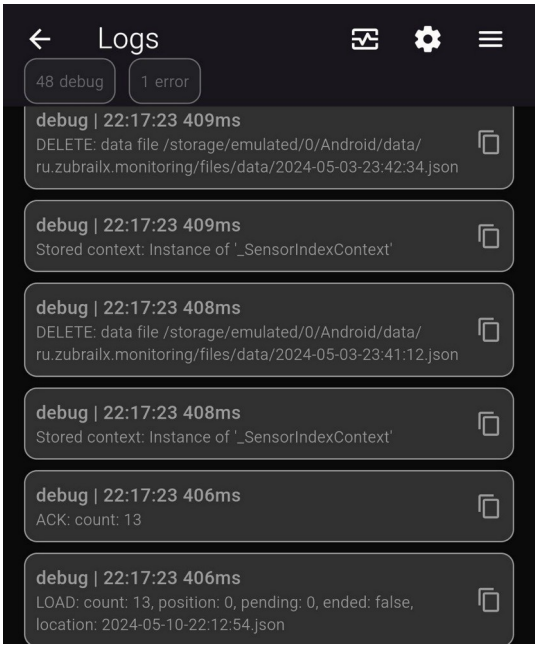


Рисунок 8 - Страница логирования

3.1.8 Данные о пользователе

Данные о пользователе используются для его идентификации при отправке значений датчиков.

Считывание и изменение данных происходит аналогично управлению конфигурациями (AccountData является вложенным полем объекта ConfigurationData). Уникальный идентификатор может быть сгенерирован автоматически при нажатии на кнопку «Generate ID». Для сохранения данных необходимо нажать на кнопку «Save».

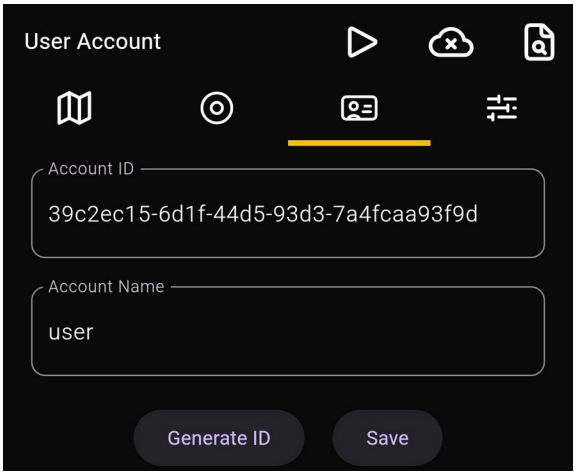


Рисунок 9 - Экран «User Account»

3.2 Сервис прогнозирования

Для обеспечения работоспособности системы требуется реализовать алгоритм, который по входным данным способен прогнозировать результат состояния дорожного покрытия, который принадлежит отрезку [0, 1]. Такой вариант был выбран по причине удобства интерпретации результатов, где 0 — плохая дорога, 1 — хорошая, поскольку во время постановки задач задумывалось работать с оценками (как в проекте «Автострада», но для конкретных участков), а не классами. Таким образом, для инференции должны либо использоваться алгоритмы регрессии, либо классификаторы, результаты которых отображаются на отрезок [0, 1].

Чтобы настроить и протестировать алгоритмы необходимы исходные данные для обучения. Для этого использовался датасет PVS — Passive Vehicular Sensors Datasets[4].

В подразделе «Подготовка данных и обучение алгоритма регрессии» описывается этап подготовки исходных данных, выбора и обучения алгоритма и сохранения модели для использования в сервисе прогнозирования, разработка которого приведена в подразделе «Разработка сервиса».

3.2.1 Подготовка данных и обучение алгоритма регрессии

Поскольку данные датасета отличаются от данных, поступающих с мобильных устройств, необходимо сначала их привести к нужному формату. Поэтому были удалены неиспользуемые столбцы, сконвертирован timestamp в целое число наносекунд, из значений акселерометра оси z вычтено среднее, так как с мобильных устройств приходят величины без учета влияния гравитации.

Варианты предобработки зависят от используемых алгоритмов инференции, по причине их различного уровня устойчивости к шумам [16], необходимости фиксированного размера входных данных [17] и т. п. Последующее описание предобработки рассмотрено для алгоритма регрессии XGBoost [18] без параметра скорости, так как он показал лучшие результаты в реальных условиях. Также были протестированы CART и Random Forest. Сравнительный анализ приведен в следующей главе.

Данные алгоритмы по причине возможного переобучения хорошо работают со входами небольшой размерности и при отсутствии сильных шумов, поэтому требуются дополнительно предобработать входные данные.

---

[4] J. Menegazzo and A. von Wangenheim, "Multi-Contextual and Multi-Aspect Analysis for Road Surface Type Classification Through Inertial Sensors and Deep Learning," 2020 X Brazilian Symposium on Computing Systems Engineering (SBESC), Florianopolis, 2020, pp. 1-8, doi: 10.1109/SBESC51047.2020.9277846.

Сначала требуется произвести интерполирование, поскольку частота опроса в обучаемых данных (100 Гц) отличается от предполагаемой в мобильных устройствах (20 Гц). Для данных мобильных устройств также требуется выполнить эту операцию, поскольку события акселерометра и гироскопа не происходят одновременно в равные промежутки времени. Так как выбор метода интерполяции зависит от характера сигнала, в качестве частоты интерполяции экспериментально было выбрано значение 40 Гц, поскольку при меньших частотах (20 Гц, 30 Гц) для мобильных устройств ошибка велика, а при дальнейшем увеличении решающим фактором становится увеличение объема данных.

В соответствии с используемым алгоритмом требуется разбить данные на группы одинакового размера для дальнейшего прогнозирования по ним. Так значения акселерометра и гироскопа были разбиты на окна размером 128 элементов, для каждого из которых были расчитаны значения GPS (интерполированы аналогичным образом) и прогнозирования (функция приведена в Листинге 6). Кроме того, по данным широты и долготы через гаверсинусы (формула нахождения расстояния между двумя точками на сфере) была рассчитана скорость движения транспортного средства [19].

Листинг 6 - Расчет значения прогнозирования

```
window = 128
slide = 0.5
tick = 25000

def harmonise_prediction(time_markers: numpy.array, predDf: np.DataFrame):
  data = []
  for marker in time_markers:
    start = marker - int(tick * window * slide / 2)
    end = marker + int(tick * window * slide / 2)
    predDfr = predDf[(predDf.time >= start) & (predDf.time <= end)]
    res = (predDfr.good_road * 1 + predDfr.regular_road * 0.5 + predDfr.bad_road *
0).mean()
    data.append((marker, res))
  return pd.DataFrame(data, columns=["time", "prediction"])
```

Если дефекты находятся на границе окна, высок шанс, что алгоритм прогнозирования их пропустит (с учетом того, что далее осуществляется фильтрация, при которой сигнал запаздывает), поэтому следующее окно перекрывает предыдущее на половину (на величину slide равную 0.5). Не до конца заполненные окна (в конце массивов данных датчиков) отбрасываются, так как не могут быть нормально обработаны.

После разбиения значения акселерометра и гироскопа фильтруются с помощью фильтра низких частот, чтобы избавиться от высокочастотных шумов, генерируемых автомобилем и мобильным устройством. В приложении используется фильтр IIR (с бесконечной импульсной характеристикой) Баттерворта порядка 4 с частотой 10 Гц, так как он быстро работает и хорошо убирает высокие частоты, но при этом результирующий сигнал имеет отставание от исходного.

Для снижения количества входов алгоритма регрессии значения акселерометра и гироскопа были переведены в магнитуды. Далее, чтобы избавиться от зависимости по времени, массив вычисленных магнитуд с помощью алгоритма быстрого преобразования Фурье был переведен в спектр сигнала, представляющий собой амплитуды частот. Так как для прогнозирования важны по большей части нижние частоты, то в список фич были добавлены только значения, не превышающие частоты фильтра низких частот. Дополнительно в список были включены значения максимальных магнитуд акселерометра и гироскопа, а также скорость автомобиля.

Чтобы дополнительно сократить размер входных данных, можно выбрать только часть параметров из входного списка. Экспериментально было принято решение оставить 24 фичи, наиболее коррелирующие с результатом прогнозирования (Листинг 7). Кроме того, вручную был исключен атрибут скорости, так как он плохо влияет на результаты прогнозирования в реальных условиях, несмотря на то, что он был в списке наиболее коррелирующих элементов.

Листинг 7 - Результат корреляционного анализа (имя фичи — корреляция)

```
[13322 rows x 67 columns] prediction 1.000000, 0 0.710628, 64 0.683036, 65 0.636230,
32 0.573590, …, 3 0.438936, 34 0.432453, 2 0.411814, 33 0.327395, 1 0.327106
```

Наконец, на данных датасета был обучен алгоритм. Для сохранения модели использовалась библиотека pickle. Сами результаты для различных алгоритмов приведены в следующей главе.

Листинг 8 - Обучение и сохранение модели

```
data = pd.read_csv(out_features_predictions_selected_fpath)

reg = xgb.XGBRegressor()

data_shuffled = features_prediction_df_selected.sample(frac=1)
X = data_shuffled.drop(['prediction'], axis=1)
y = data_shuffled['prediction']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

reg = reg.fit(X_train.values, y_train)

with open(os.path.join(model_dir, f"features-{k}{postfix}.pickle"), "wb") as file:
  pickle.dump(reg, file)
```

Итак, в результате была получена обученная модель и набор выбранных признаков. Благодаря разделению процесса предобработки и обучения на несколько независимых этапов была обеспечена возможность быстрой замены алгоритмов машинного обучения, параметров обучения, фильтров и дополнительных обработчиков.

3.2.2 Разработка сервиса

С помощью библиотеки kafka-python данный сервис подписывается на топик monitoring брокера Kafka. При получении сообщения потоку из пула (thread pool) делигируется вызов функции consumer_func, которая считывает и десериализовывает сообщение Monitoring Protobuf. Далее, аналогично тому, как описывается в Подразделе 3.2.1, осуществляется интерполяция, вычисляется скорость, удаляются шумы, получаются и выбираются фичи и происходит инференция загруженной моделью. Результат помещается в

список, который затем сериализуется в сообщение Points и отправляется в топик points. Код вышесказанного приведен в Листинге 9.

Обработка функции consumer_func происходит в пуле процессов. Размер группы pool_size задается в конфигурации KafkaConsumerCfg для реализованной обертки над kafka-python. Обработка запросов реализована аналогично проблеме producer-consumer: так как процесс KafkaConsumer блокируется при помощи семафора, считывание нового сообщения не происходит до тех пор, пока в очереди на выполнение (multiprocessing pool cache) находится заданное число асинхронных задач, а при освобождении блокировки в очередь добавляется новая задача. Благодаря такому подходу очередь не переполняется и не приводит к значительному увеличению потребляемой приложением памяти при превышении скорости считывания сообщений над возможной скоростью их обработки.

Листинг 9 - Обработка и прогнозирование

```
def consumer_func(msg):
  try:
    proto = Monitoring()
    proto.ParseFromString(msg.value)

    (acDf, gyDf, gpsDf) = get_raw_filtered_inputs(proto)
    (acDfi, gyDfi, gpsDfi) = interpolation.interpolate(acDf, gyDf, gpsDf)
    speedArr = gps.calculate_speed(gpsDfi)

    entries = interpolation.get_point_raw_inputs(acDfi, gyDfi, gpsDfi, speedArr)

    point_results = []
    for (acDfe, gyDfe, gpsDfe) in entries:
      acDfn, gyDfn = processing.reduce_noice(acDfe, gyDfe)
      features = processing.extract_features(acDfn, gyDfn, gpsDfe)
      selected_features = selector.select_features(features)
      prediction = predictor.predict_one(selected_features)
      point_results.append((prediction, gpsDfe))

    points = Points()
    points.point_records.extend(map(point_result_to_record, point_results))
    produce(points)
    # handle exception
```

3.3 Сервис пользовательских запросов

Данный сервис предоставляет HTTP REST API доступ к результатам прогнозирования. В качестве библиотеки маршрутизации была выбрана go-chi, так как в ней реализована обработка переменных и шаблонов маршрутизации на основе регулярных выражений в URL путях, добавление промежуточных обработчиков (middleware), например для аутентификации, и на момент разработки имела активную поддержку сообществом разработчиков.

В Листинге 10 представлен пример запроса для получения результатов прогнозирования. Возвращается JSON сериализованный список точек, записи которых включают координаты широты и долготы, время (в формате RFC3339) и значение прогнозирования.

Листинг 10 - Запрос получения результатов прогнозирования

```
"GET http://url:port/points/{z}/{x}/{y}?begin={begin_time}&end={end_time} HTTP/1.1"
```

В запрос передаются следующие аргументы:

- z — уровень масштабирования;
- x, y — номер ячейки;
- begin_time, end_time — фильтр времени (опционально).

Для получения позиции ячейки (x, y) из значении широты, долготы и масштаба используются формулы, аналогичные проекциям Меркатора [20]. В Листинге 11 представлена функция, выполняющая обратную задачу.

Листинг 11 - Функция перевода из проекции Меркатора в широту и долготу

```
func fromPointTile(x, y, z int) (float64, float64) {
	n := math.Pow(2, float64(z))

	longitude := float64(x)/n*360.0 - 180.0
	latitudeRad := math.Atan(math.Sinh(math.Pi * (1.0 - 2.0*float64(y)/n)))
	latitude := latitudeRad * 180.0 / math.Pi

	return longitude, latitude
}
```

Благодаря переводу конкретных координат в номера ячеек может быть реализовано кеширование запросов на стороне веб-сервера или базы данных или агрегирование результатов, для снижения нагрузки на выполнение запросов или отрисовку мобильным устройством.

3.4 Сервис обработки результатов

С помощью библиотеки kafka-go считывает результаты прогнозов из очереди. Она была выбрана по причине наличия подробной документации и реализованного механизма контекстов (context) Go, позволяющего обрабатывать ситуации, такие как завершение работы (graceful shutdown) и ошибки выполнения без принудительного разрыва соединения. Для чтения сообщений используется Reader, внутри которого эта операция оптимизирована с помощью считывания сразу нескольких сообщений. Так как в СУБД Clickhouse эффективно реализована вставка большого количества записей одним запросом, поэтому во время чтения перед вставкой записи сначала добавляются в батч (batch), а затем осуществляется запрос к СУБД. Операция вставки исполняется либо при превышении лимита записей, либо по прошествии заданного промежутка времени (Листинг 12).

Листинг 12: Буферизированная вставка в Clickhouse

```
points-consumer-1  | 2024/05/04 08:25:40 clickhouse: inserted 1000 rows
points-consumer-1  | 2024/05/04 08:25:43 clickhouse: inserted 1000 rows
points-consumer-1  | 2024/05/04 08:25:48 clickhouse: inserted 2 rows
points-consumer-1  | 2024/05/04 08:26:43 clickhouse: inserted 64 rows
points-consumer-1  | 2024/05/04 08:26:48 clickhouse: inserted 440 rows
points-consumer-1  | 2024/05/04 08:26:53 clickhouse: inserted 714 rows
points-consumer-1  | 2024/05/04 08:26:58 clickhouse: inserted 569 rows
points-consumer-1  | 2024/05/04 08:27:03 clickhouse: inserted 708 rows
points-consumer-1  | 2024/05/04 08:27:08 clickhouse: inserted 932 rows
points-consumer-1  | 2024/05/04 08:27:13 clickhouse: inserted 1000 rows
points-consumer-1  | 2024/05/04 08:27:17 clickhouse: inserted 1000 rows
```

В логах приложения, показанного в Листинге 12, заданы параметры:

- размер буфера — 2000 (поэтому допустима вставка 1000 записей);
- лимит — 1000, промежуток отправки — 5000 миллисекунд.

4 ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ

Работа мобильного приложения проверялась на устройстве под управлением операционной системы Android 14.

Для запуска сервисов использовался сервер со следующими характеристиками:

- операционная система — Ubuntu 22.04.4 LTS x86_64;
- CPU — 2 ядра 2.6GHz;
- оперативная память — 4 GB.

Для упаковки и запуска сервисов использовался инструмент Docker: для каждого из разработанных сервисов были предварительно созданы docker образы, написан файл docker-compose.yml.

С мобильного приложения были собраны значения датчиков, которые далее были обработаны при помощи сервиса прогнозирования и записаны в базу данных. В результате было получено 34 тысячи оценок в определенных географических точках.

Листинг 13 - Записи в БД результатов прогнозирования

```
SELECT
    time,
    latitude,
    prediction
FROM points
LIMIT 34000, 10

 1. | 2024-05-08 05:26:12 |  59.83308446719243 |  0.9991886 |
 2. | 2024-05-08 05:26:14 | 59.833086408468816 |  0.9991886 |
 3. | 2024-05-08 05:26:15 |   59.8330883497452 |  0.9991886 |
 4. | 2024-05-08 05:26:17 |   59.8330902910216 |  0.9991886 |
 5. | 2024-05-08 05:26:19 | 59.833092232297986 |  0.9991886 |
 6. | 2024-05-08 05:26:20 |  59.83309417357437 |  0.9991886 |
 7. | 2024-05-08 05:26:22 |  59.83309611485076 | 0.99708635 |
 8. | 2024-05-08 05:26:23 | 59.833098056127156 | 0.99708635 |
 9. | 2024-05-08 05:26:25 | 59.833099997403544 |  0.9996782 |
10. | 2024-05-08 05:26:27 |  59.83310193867993 |          1 |
```

Результаты визуализации точек приведены на Рисунке 10.

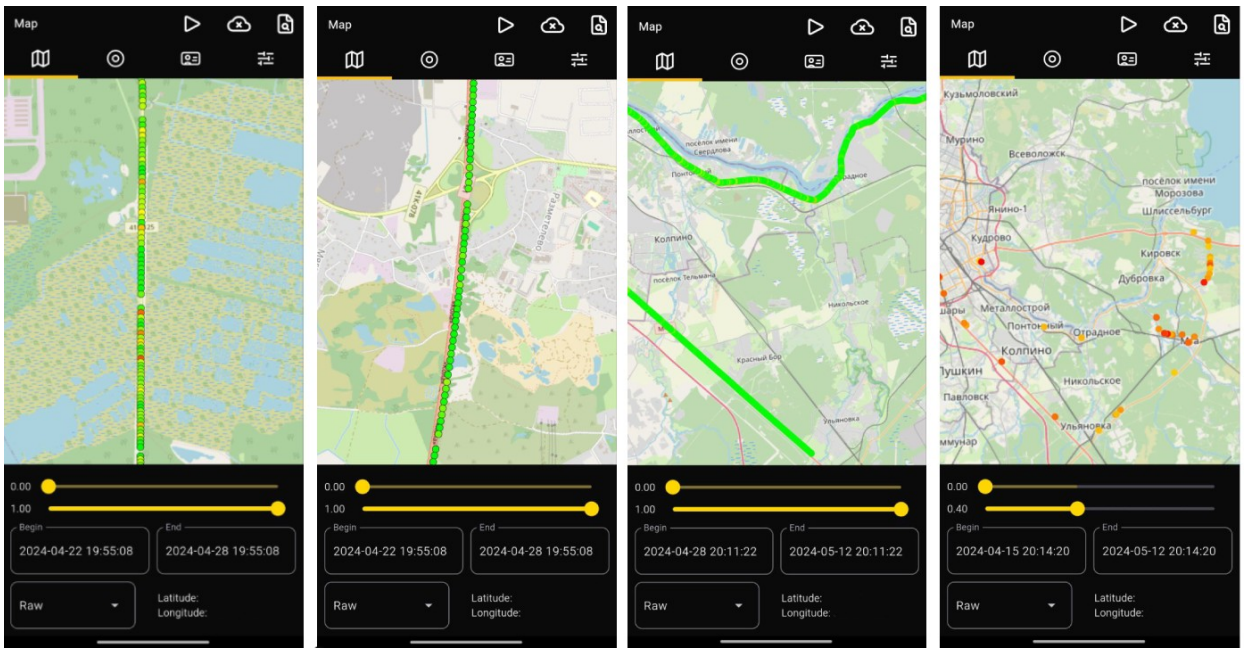


Рисунок 10 - Визуализация результатов на мобильном приложении

Через интерфейс Kafka UI может быть получена информация о топиках, статусах читателей (consumers), репликации, размере занимаемого дискового пространства (Рисунок 11).

Topic Name	Partitions	Out of sync replicas	Replication Factor	Number of messages	Size
IN __consumer_offsets	50	0	1	181636	48 KB
IN _confluent-command	1	0	1	0	0 Bytes
monitoring	1	0	1	274	22 MB
monitoring-loader	1	0	1	0	0 Bytes
points	1	0	1	108	84 KB

Рисунок 11 - Топики Kafka

Необходимо обратить внимание, что кроме описанных выше топиков присутствует топик monitoring-loader, который используется дополнительно реализованными сервисами для сохранения данных датчиков до момента их обработки сервисом прогнозирования. Таковыми сервисами являлись monitoring-keeper (читает данных из топика monitoring параллельно сервису прогнозирования и сохраняет это данные в коллекцию MongoDB), monitoring-loader (считывает данные из коллекции MongoDB и записывает тоже самое в топик monitoring-loader, чтобы monitoring-keeper не прочитал те же данные). Сохранение до прогнозирования было необходимо для проверки работы

различных алгоритмов прогнозирования на реальных данных: при смене алгоритма очищалась база данных Clickhouse, все сохраненные данные датчиков считывались из коллекции MongoDB, и по ним заново производилось вычисление. Факт сохранения показан на Рисунке 12.

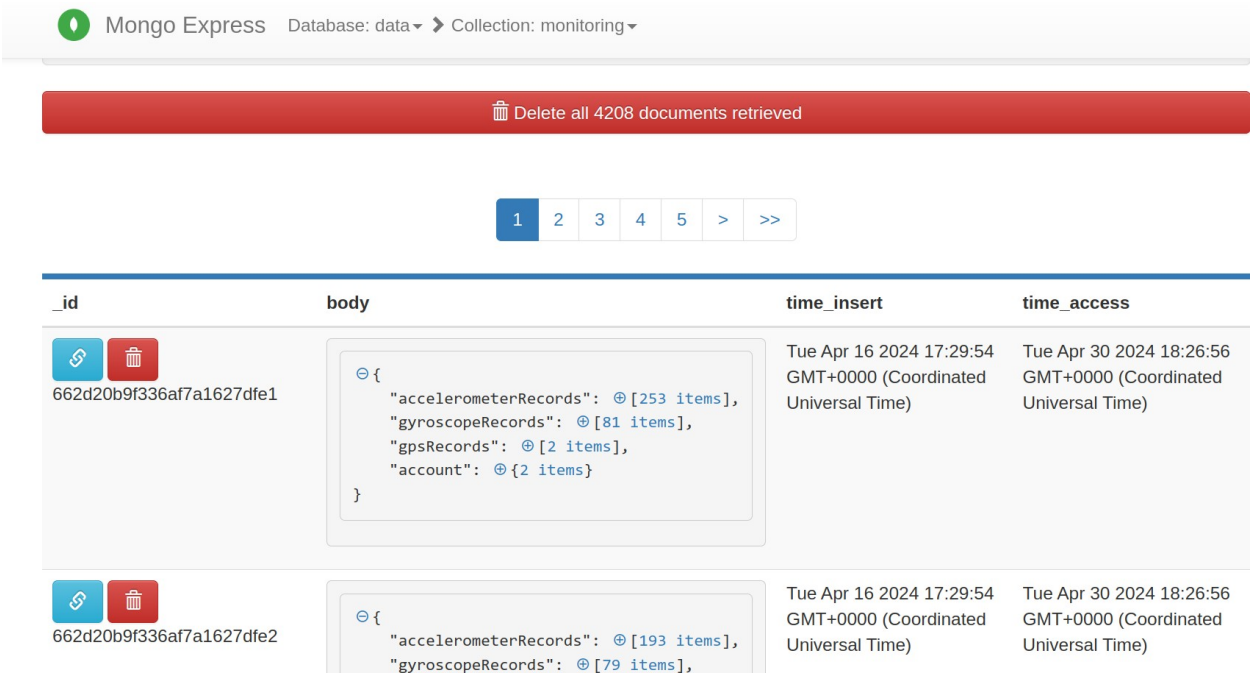


Рисунок 12 - Коллекция monitoring MongoDB

Для алгоритма XGBoost без параметра скорости, который был выбран в качестве основного, на данных PVS с разделением тренировочной и тестовой выборки в соотношении 7:3 был получен коэффициент детерминации R2 0.70. Несмотря на то, что этот же алгоритм с параметром скорости показал значение R2 0.89, данная модель не была выбрана по причине низкой чувствительности на реальных данных из-за их неполноты (значительно выше вероятность ошибочного определения плохих участков на дорогах как хороших): требуется расширение датасета. Глубокий перебор параметров для данного алгоритма не производился по аналогичной причине.

Помимо точности прогнозирования также был проведен анализ скорости. Чтобы исключить дополнительную нагрузку на считывание данных из MongoDB и запись в топик Kafka сервисом monitoring-loader, сохраненные данные были перенесены на другой сервер, где при помощи отдельной

утилиты mobile-imitator осуществлялась имитация отправки данных мобильными устройствами, при этом было отключено сохранение данных датчиков. Брокер Kafka, СУБД Clickhouse, Kafka MQTT Proxy, сервисы прогнозирования, обработки результатов и пользовательских запросов были подняты в одном экземпляре. Оптимизации конфигураций нереализованных сервисов не производилось.

В результате одного цикла тестирования отправлялось 4.2 тысячи сообщений с датчиками, из которых генерировалось суммарно 34 тысячи результатов прогнозирования. Итак, время, затрачиваемое на полную обработку всех запросов, при использовании в среднем составило 150 секунд. Отправка данных утилитой (в 4 процесса) занимала 45 секунд, что приводило к накоплению еще не обработанных сообщений в очередях Kafka и полной загрузке вычислительных ресурсов процессора. В процессе выполнения тестирования аномального увеличения объема занимаемой памяти в сервисах не наблюдалось (например, это могло происходить при накапливании заявок на прогнозирование после считывания данных из Kafka, когда скорость обработки ниже скорости считывания сообщений, или при отсутствии лимитов буферов в сервисе обработки результатов прогнозирования). Сравнение алгоритмов на обучающем датасете с указанием использованных параметров приведено в Приложении Б.

Благодаря отсутствию причинно-следственных связей между сообщениями внутри одного топика брокера отсутствует необходимость предоставления каких-либо гарантий упорядоченности сообщений, что позволяет производить горизонтальное масштабирование для каждого из этапов потока обработки данных датчиков. Это можно реализовать при помощи добавления дополнительных читателей (consumers) внутри одной группы Kafka, например, запустив несколько экземпляров реализованных сервисов (сервисы реализованы таким образом, что внутри них может быть только один объект KafkaConsumer). Кроме того, в сервисе прогнозирования

может быть задано число потоков обработчиков или в сервисе обработки результатов увеличено количество горутин, что позволяет увеличить скорость обработки в случаях когда чтение сообщений не является узким местом.

Подводя итоги, сравнивая с указанными выше аналогами, реализованная система имеет следующие характеристики:

1. Средняя серверная нагрузка — пользовательские запросы многочисленны, каждый из них требует предобработки и обработки малой/средней моделью машинного обучения.
2. Используются малые/средние алгоритмы машинного обучения.
3. Средняя детализация — результатом работы является аналогичная «Xweather RoadAI» карта состояний дорог, состоящая из точек прогнозов без привязки к дорогам.
4. Система работает автономно — после первоначального запуска и настройки приложения не требуется дополнительного вмешательства.
5. Низкая нагрузка на мобильное устройство — приложение только собирает данные с датчиков и отправляет их на сервер, а все алгоритмы обработки вынесены на серверную часть.
6. Ограничения на использование отсутствуют — не требуется фиксирования мобильного устройства (однако, вероятно, точность может быть выше), система работает аналогично вне зависимости от погодных условий.

ЗАКЛЮЧЕНИЕ

В данной работе было предложено следующее решение сбора данных о состоянии дорожного покрытия: мобильные устройства отправляют данные датчиков GPS, акселерометра и гироскопа в сервис прогнозирования, который при помощи алгоритмов машинного обучения получает результаты оценок состояния дорожного покрытия и отправляет их в сервис, обеспечивающий буферизированную обработку и вставку в БД. Кроме того, для мобильного приложения был разработан модуль визуализации результатов, получающий данные с сервиса пользовательских запросов.

Выбор и обучение алгоритмов машинного обучения были произведены на открытом датасете, для данных которого, как и для данных мобильного приложения во время реальной работы, дополнительно осуществлялись этапы предобработки: интерполяция, фильтрация, выбор фич, чтобы увеличить устойчивость и скорость инференции. Затем, после проведения тестирования, на основании скорости и точности прогнозирования на реальных и обучающих данных был выбран алгоритм XGBoost с определенными параметрами и входными фичами (выбран алгоритм с R2 0.70 без признака скорости по причине его большей чувствительности к дефектам на дорогах).

Нагрузочное тестирование, имитирующее отправку данных мобильными устройствами, показало, что система, запущенная на сервере с 2 ядрами 2.6 ГГц, 4 Гб оперативной памяти, может обработать 4.2 тысячи пользовательских сообщений, содержащих 30 секунд данных датчиков, сгенерировав 34 тысячи результатов, за 150 секунд. Следует также отметить, что система разрабатывалась с учетом возможности проведения горизонтального масштабирования без необходимости перестроения архитектуры. При этом дополнительно стоит обратить внимание, что сервисы прогнозирования и обработки результатов были разработаны

многопоточно с возможностью конфигурации буферов и количества потоков обработки.

Также были реализованы дополнительные оптимизации мобильного приложения: кеширование обращений к серверу и локальное сохранение данных датчиков в случае отсутствия интернет соединения.

Таким образом, разработанная система в сравнении с альтернативами, имеет следующие характеристики: мобильное приложение имеет низкую нагрузку на устройство из-за отсутствия вычислительно сложных задач на стороне пользователя, работает автономно, позволяет сохранять данные локально, при этом серверное ПО имеет среднюю нагрузку из-за большого объема передаваемых данных, малых/средних алгоритмов машинного обучения и необходимости предобработки сообщений.

В итоге, проделанная работа открывает новые перспективы для дальнейшего исследования в данной области:

1. Внедрение системы, благодаря полученным характеристикам, в качестве модуля в существующие картографические сервисы без ухудшения пользовательского опыта.
2. Увеличение точности и скорости инференции с помощью других конфигураций предобработки и алгоритмов машинного обучения, обученных на большем объеме данных.
3. Реализация кеширования запросов на стороне сервера или прокси.
4. Агрегация результатов для уменьшения объема запрашиваемых данных, ускорения ответов от сервера, получения значений в строго заданных временных интервалах. Это может обеспечить получение более точных оценок состояния дорожного покрытия, а также реализовать запрос на получение изменений состояния дорожного покрытия за заданный промежуток времени.

5. Добавление проекции результатов на кривую ближайшей дороги с карты с учетом данных об ускорении и направлении движения транспортного средства.
6. Включение новых характеристик об оценках состояния дорожного покрытия, таких как тип дороги (например, асфальт, гравий или грязь), дорожные объекты (лежачие полицейские, переезды через железнодорожные пути), характеристики используемого дорожного средства (тип подвески, амортизация), направление движения и т. п. Указанные характеристики позволят дополнительно увеличить точность работы и полноту информации о дороге.

В результате выполнения поставленных задач, можно заключить, что цель работы, заключавшаяся в обеспечении отказа от специализированной аппаратуры в пользу использования мобильных устройств для сбора данных о состоянии дорожного покрытия, была достигнута. Так, был произведен анализ предметной области, показавший возможность использования датчиков мобильных устройств для решения поставленной цели, существующих альтернатив, по которым выбраны характеристики для проведения сравнения, разработана архитектура и реализована система, получен обучающий датасет, обучен алгоритм машинного обучения на приведенных данных, протестирована работа в реальных сценариях использования, а также осуществлено сравнение полученного решения с альтернативами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. W. H. Organization. (2023) Global status report on road safety [Электронный ресурс]. Режим доступа: https://iris.who.int/bitstream/handle/10665/375016/9789240086517-eng.pdf, свободный (дата обращения: 01.05.2024).
2. Екатерина Александровна Пономарева, Александра Дмитриевна Савина Факторы, влияющие на смертность в ДТП // Экономическая политика. 2022. №4. URL: https://cyberleninka.ru/article/n/faktory-vliyayuschie-na-smertnost-v-dtp, свободный (дата обращения: 01.05.2024).
3. Шабазов М.М. ВЛИЯНИЕ КАЧЕСТВА ДОРОГ НА СТОИМОСТЬ ПЕРЕВОЗКИ ГРУЗОВ И НА РАСХОД ТОПЛИВА // Мировая наука. 2023. №5 (74). URL: https://cyberleninka.ru/article/n/vliyanie-kachestva-dorog-na-stoimost-perevozki-gruzov-i-na-rashod-topliva, свободный (дата обращения: 01.05.2024).
4. Дорожные условия, влияющие на надежность и долговечность автомобиля [Электронный ресурс]. Режим доступа: https://stroy-technics.ru/article/dorozhnye-usloviya-vliyayushchie-na-nadezhnost-i-dolgovechnost-avtomobilya, свободный (дата обращения: 01.05.2024).
5. Wan G. et al. The impact of road infrastructure on economic circulation: Market expansion and input cost saving //Economic Modelling. – 2022. – Т. 112. – С. 105854.
6. Official U.S. government information about the Global Positioning System (GPS) and related topics [Электронный ресурс]. Режим доступа: https://www.gps.gov/systems/gps/performance/accuracy/, свободный (дата обращения 01.05.2024)
7. Strongman C. et al. A Scoping Review of the Validity and Reliability of Smartphone Accelerometers When Collecting Kinematic Gait Data //Sensors. – 2023. – Т. 23. – №. 20. – С. 8615.

8. Umek A., Kos A. Validation of smartphone gyroscopes for mobile biofeedback applications //Personal and Ubiquitous Computing. – 2016. – Т. 20. – С. 657-666.
9. Korab J. Understanding Message Brokers: Learn the Mechanics of Messaging Though ActiveMQ and Kafka. – O'Reilly Media, 2017.
10. Алиев Т. И. Основы моделирования дискретных систем //СПб: СПбГУ ИТМО. – 2009. – Т. 363.
11. Apache Kafka and MQTT — Overview and Comparison [Электронный ресурс]. Режим доступа: https://www.kai-waehner.de/blog/2021/03/15/apache-kafka-mqtt-sparkplug-iot-blog-series-part-1-of-5-overview-comparison/, свободный (дата обращения 01.05.2024)
12. Benchmarking Apache Pulsar, Kafka, and RabbitMQ [Электронный ресурс]. Режим доступа: https://www.confluent.io/blog/kafka-fastest-messaging-system/, свободный (дата обращения 01.05.2024)
13. Data serialization tools comparison: Avro vs Protobuf [Электронный ресурс]. https://softwaremill.com/data-serialization-tools-comparison-avro-vs-protobuf/, свободный (дата обращения: 01.05.2024)
14. Flutter: documentation [Электронный ресурс]. https://docs.flutter.dev/, свободный (дата обращения: 01.05.2024)
15. Методология FSD [Электронный ресурс]. https://feature-sliced.design/docs/get-started/overview, свободный (дата обращения 01.05.2024)
16. Arora S. et al. Stronger generalization bounds for deep nets via a compression approach //International Conference on Machine Learning. – PMLR, 2018. – С. 254-263.
17. Yu Y. et al. A review of recurrent neural networks: LSTM cells and network architectures //Neural computation. – 2019. – Т. 31. – №. 7. – С. 1235-1270.

18. Chen T., Guestrin C. Xgboost: A scalable tree boosting system //Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. – 2016. – С. 785-794.
19. Van Brummelen G. Heavenly mathematics: The forgotten art of spherical trigonometry. – Princeton University Press, 2012.
20. Osborne P. The mercator projections. – Peter Osborne, 2013.
21. DB-engines: Knowledge Base of Relational and NoSQL Database Management Systems [Электронный ресурс]. https://db-engines.com/en/ranking, свободный (дата обращения 01.05.2024)
22. Кандидов В. П., Чесноков С. С., Шленов С. А. Дискретное преобразование Фурье //М.: Физический факультет МГУ. – 2019
23. Kleppmann M. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. – " O'Reilly Media, Inc.", 2017
24. Proakis J. G. Digital signal processing: principles, algorithms, and applications, 4/E. – Pearson Education India, 2007
25. Lin L. et al. A new FCM-XGBoost system for predicting Pavement Condition Index //Expert Systems with Applications. – 2024. – Т. 249. – С. 123696.
26. Agrawal N. et al. Design of digital IIR filter: A research survey //Applied Acoustics. – 2021. – Т. 172. – С. 107669.
27. Wang Y., Perera L. P., Batalden B. M. Coordinate conversion and switching correction to reduce vessel heading related errors in high-latitude navigation //IFAC-PapersOnLine. – 2023. – Т. 56. – №. 2. – С. 11602-11607.
28. McCaffrey P. An introduction to healthcare informatics: building data-driven tools. – Academic Press, 2020.
29. Struckov A. et al. Evaluation of modern tools and techniques for storing time-series data //Procedia Computer Science. – 2019. – Т. 156. – С. 19-28.

30. Kong Q., Siauw T., Bayen A. Python programming and numerical methods: A guide for engineers and scientists. – Academic Press, 2020.
31. Wahab M. F., Gritti F., O'Haver T. C. Discrete Fourier transform techniques for noise reduction and digital enhancement of analytical signals //TrAC Trends in Analytical Chemistry. – 2021. – T. 143. – C. 116354.

ПРИЛОЖЕНИЕ А

Структура сообщений очередей. Схемы Protobuf

```
// util.proto
syntax = "proto3";

message Timestamp {
  int64 seconds = 1;
  int32 nanos = 2;
}
```

```
// monitoring/monitoring.proto
syntax = "proto3";

package monitoring;

import "util.proto";

message UserAccount {
  string accound_id = 1;
  string name = 2;
}

message AccelerometerRecord {
  Timestamp time = 1;
  float x = 11;
  float y = 12;
  float z = 13;
  int32 ms = 21;
}

message GyroscopeRecord {
  Timestamp time = 1;
  float x = 11;
  float y = 12;
  float z = 13;
  int32 ms = 21;
}

message GpsRecord {
  Timestamp time = 1;
  double latitude = 11;
  double longitude = 12;
  double accuracy = 13;
  int32 ms = 21;
}

message Monitoring {
  repeated AccelerometerRecord accelerometer_records = 1;
  repeated GyroscopeRecord gyroscope_records = 2;
  repeated GpsRecord gps_records = 3;
```

```
  UserAccount account = 11;
}
```

```
// points/points.proto
syntax = "proto3";

package points;

import "util.proto";

message PointRecord {
  Timestamp time = 1;
  double latitude = 11;
  double longitude = 12;
  float prediction = 21;
}

message Points {
  repeated PointRecord point_records = 1;
}
```

ПРИЛОЖЕНИЕ Б

Характеристики сервера и параметры сервисов

- операционная система — Ubuntu 22.04.4 LTS x86_64;
- CPU — 2 ядра 2.6 ГГц;
- оперативная память — 4 Гб;
- GUESSR_POOL_SIZE=1;
- PC_BUFFER_SIZE=2000;
- PC_TRIGGER_THRESHOLD=1000;
- PC_TRIGGER_PERIOD=5000.

Все сервисы подняты в одном экземпляре в контейнерах.

Рассматриваемые алгоритмы прогнозирования

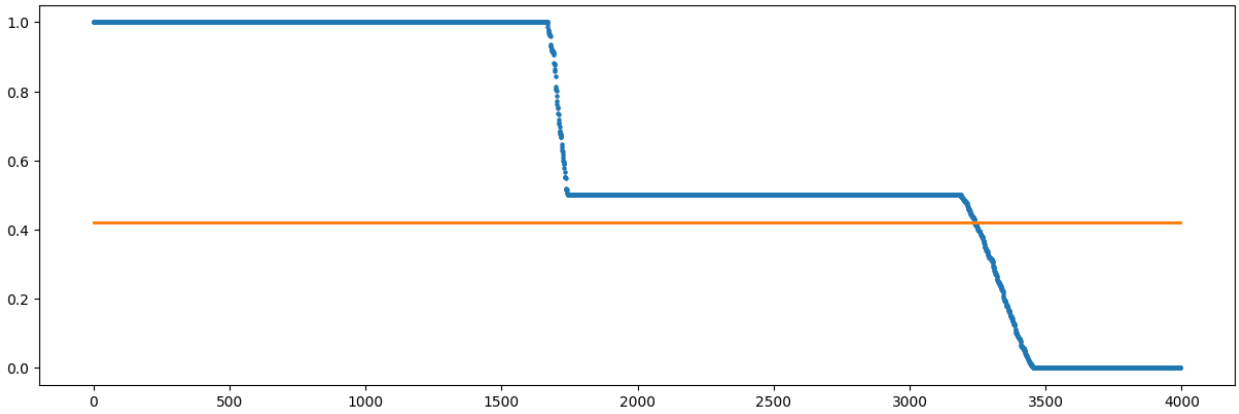
Перед всеми указанными далее алгоритмами входы интерполируются, сид для разбиения на тренировочную и тестовую выборки фиксированный, отношение разбиения 7:3. Датасет для обучения PVS[5]. Если не указан параметр в таблице алгоритм, то он равен значению по умолчанию.

1. Детерминированный без предобработки:

1.1. Этапы обработки:

1.1.1. Детерминированное прогнозирование (prediction = 0.42)

1.2. График корреляций прогнозов:



[5] J. Menegazzo and A. von Wangenheim, "Multi-Contextual and Multi-Aspect Analysis for Road Surface Type Classification Through Inertial Sensors and Deep Learning," 2020 X Brazilian Symposium on Computing Systems Engineering (SBESC), Florianopolis, 2020, pp. 1-8, doi: 10.1109/SBESC51047.2020.9277846.

2. Детерминированный с предобработкой:

2.1. Этапы обработки:

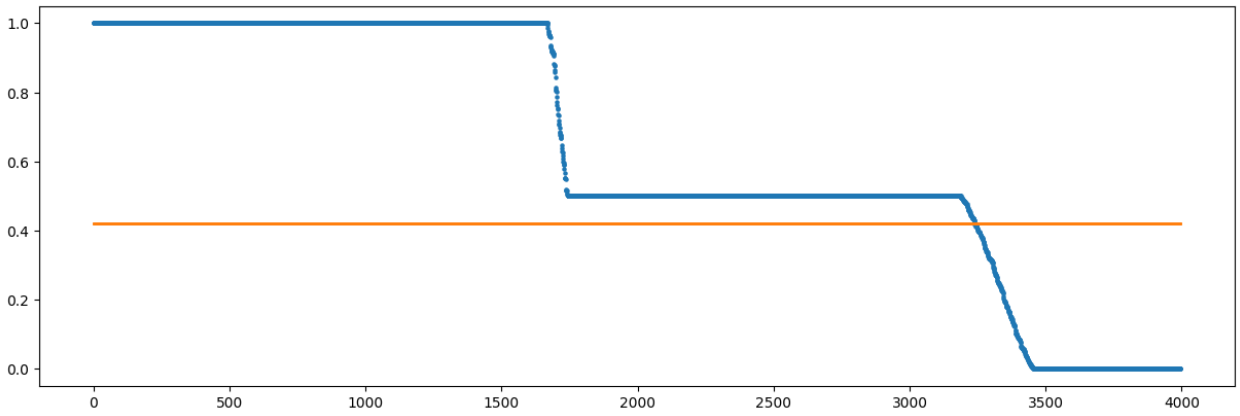
2.1.1. Очистка шумов

2.1.2. Получение фич

2.1.3. Выбор фич, k=24

2.1.4. Детерминированное прогнозирование (prediction = 0.42)

2.2. График корреляций прогнозов:



3. CART без параметра скорости

3.1. Этапы обработки:

3.1.1. Очистка шумов

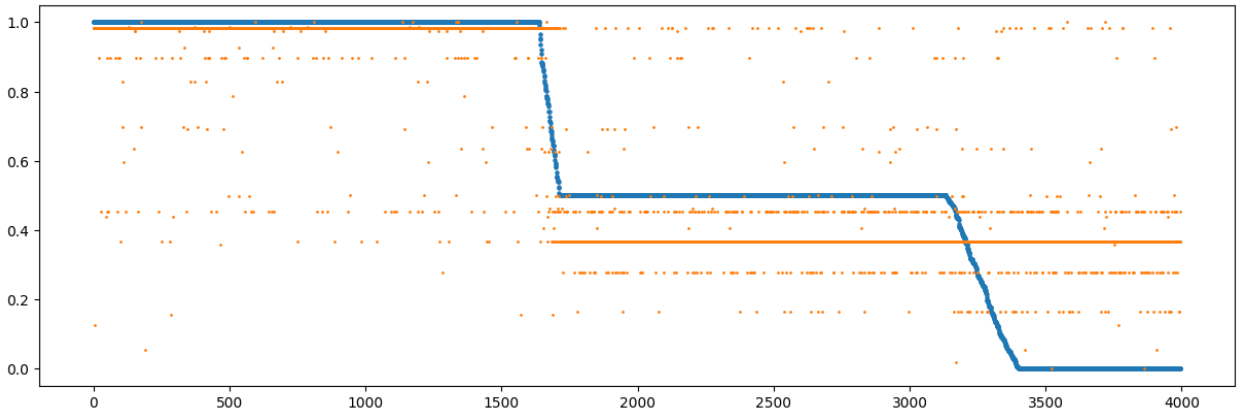
3.1.2. Получение фич

3.1.3. Выбор фич, k=24

3.1.4. Прогнозирование sklearn.tree.DecisionTreeRegressor

Имя параметра	Значение параметра
`criterion`	`squared_error`
`max_depth`	`5`
`min_samples_split`	`2`
`min_samples_leaf`	`1`

3.2. График корреляций прогнозов:



4. Random Forest без параметра скорости

4.1. Этапы обработки:

4.1.1. Очистка шумов

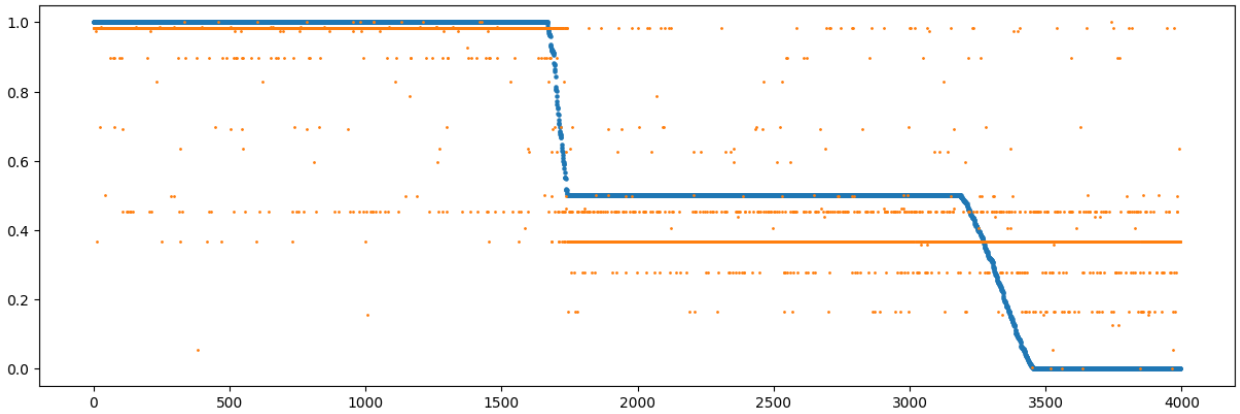
4.1.2. Получение фич

4.1.3. Выбор фич, k=24

4.1.4. Прогнозирование sklearn.ensemble.RandomForestRegressor

Имя параметра	Значение параметра
criterion	squared_error
n_estimators	100
max_depth	3
min_samples_split	2
min_samples_leaf	1

4.2. График корреляций прогнозов:



5. XGBoost без параметра скорости

5.1. Этапы обработки:

5.1.1. Очистка шумов

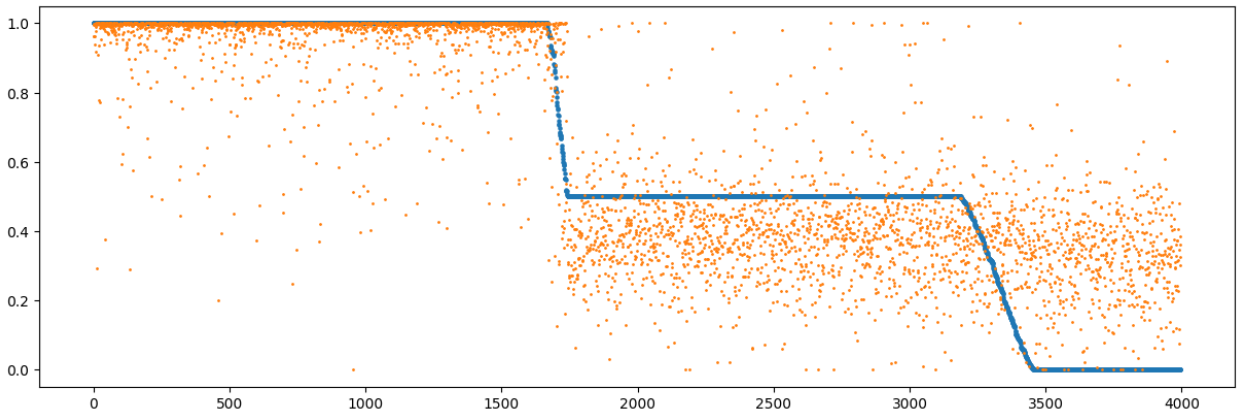
5.1.2. Получение фич

5.1.3. Выбор фич, k=24

5.1.4. Прогнозирование xgboost.XGBRegressor

Имя параметра	Значение параметра
objective	reg:squarederror
learning_rate	0.3
max_depth	6

5.2. График корреляций прогнозов:



6. XGBoost с параметром скорости

6.1. Этапы обработки:

6.1.1. Очистка шумов

6.1.2. Получение фич

6.1.3. Выбор фич, k=24

6.1.4. Прогнозирование xgboost.XGBRegressor

Имя параметра	Значение параметра
objective	reg:squarederror
learning_rate	0.3
max_depth	6

6.2. График корреляций прогнозов:

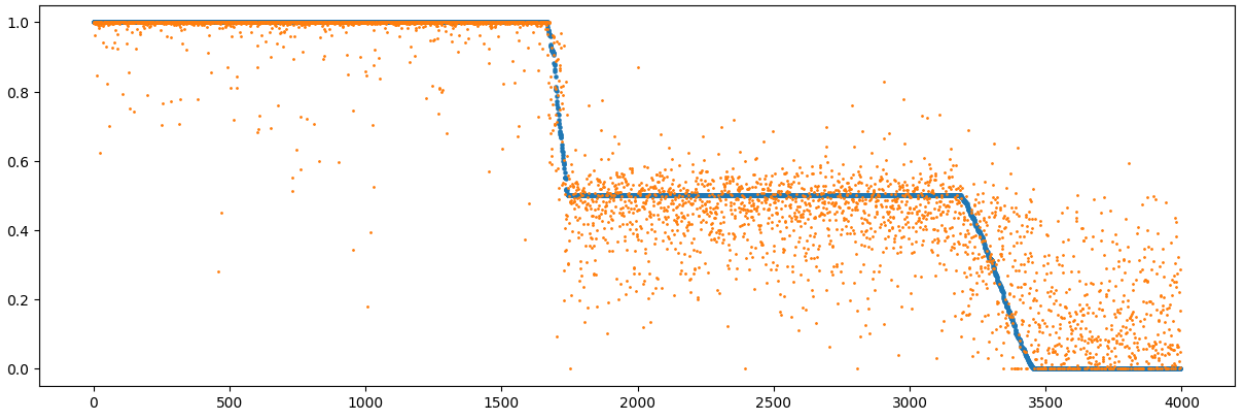


Таблица характеристик регрессии и времени исполнения

В таблице номер строки обозначает пункт в списке алгоритма выше.

	R2	Explained variance	MSE	MAE	RMSE	Время исполнения (с)
1	-0.34	0	0.173	0.347	0.416	45
2	-0.34	0	0.173	0.347	0.416	105
3	0.67	0.67	0.044	0.142	0.210	113
4	0.67	0.67	0.042	0.147	0.206	236
5	0.70	0.70	0.039	0.131	0.198	150
6	0.89	0.89	0.014	0.065	0.119	150