Федеральное государственное автономное образовательное учреждение
высшего образования
*«Национальный исследовательский университет
«Высшая школа экономики»*

Факультет компьютерных наук

КУРСОВОЙ ПРОЕКТ

Сервис интерпретации предлагаемых моделями решений с помощью расчета и визуализации значений SHAP

по направлению подготовки Прикладная математика и информатика
Образовательная программа «Финансовые технологии и анализ данных»

Работу выполнил:
Студент группы мФТиАД22
Скворцов Иван Александрович

Руководитель:
Масютин Алексей Александрович,
доцент

Москва, 2023

Аннотация

С распространением моделей машинного обучения и все большим их проникновением в нашу жизнь, все более стоит потребность в валидации принимаемых ими решений, а также в объяснении принципов их работы внешним заказчикам. Эти задачи решаются методами интерпретации предсказаний ML-моделей, таких как SHAP или LIME. Данная работа сосредоточена на разработке нового сервиса для обращения к ML-моделям Центра искусственного интеллекта НИУ ВШЭ с возможностью получения интерпретации отдельных предсказаний. В ходе работы были выбраны подходящие к задаче методы интерпретации, изучена существующая кодовая база и определены требования к новому сервису. С использованием современных подходов к разработке был разработан прототип сервиса на базе микросервисной архитектуры, а также намечены пути его дальнейшего развития.

Содержание

Введение

Модели машинного обучения широко применяются для решения целого спектра прикладных задач, в том числе в науке и бизнесе. Однако если высококвалифицированные специалисты, проектирующие пригодные к использованию модели, понимают особенности их работы, то конечным заказчикам не всегда очевидна адекватность выводов моделей и основания, на которых был получен тот или иной результат. Эта проблема возникает, поскольку при возрастающих требованиях к точности предсказаний моделей неизбежно падает их интерпретируемость, в связи с чем большинство современных решений в области машинного обучения представляют собой «черный ящик».

В связи с этим для презентации алгоритмов «во внешний мир», а также для защиты их работоспособности в контексте конкретных бизнес-задач используются подходы к объяснению предсказаний моделей машинного обучения. Они позволяют решить проблемы, возникающие из-за непрозрачности современных алгоритмов. В результате разработчики и заказчики получают возможность убедиться в отсутствии фундаментальных ошибок в работе моделей машинного обучения, обнаружить ошибки и неточности в обучающих данных, наглядно презентовать логику работы алгоритмов и так далее.

Более того, в условиях, когда машинному обучению отводится решение все более существенных с общественной точки зрения задач, таких как кредитный скоринг, определение состояния здоровья или пилотирование транспортных средств, объяснение принципов, на которых основываются предсказания, принимает еще и этическое значение. В конце концов, модели обучаются на данных, сгенерированных людьми, и содержат множество человеческих предубеждений и предрассудков. Наглядное объяснение предсказаний моделей может помочь обнаружить эти изъяны до того, как «объективный» алгоритм начнет принимать решения, обуславливающие жизнь реальных людей.

В этой работе не затрагиваются фундаментальные вопросы интерпретации моделей машинного обучения, касающиеся определения качества объяснения, нахождения компромисса между сложностью модели и простотой интерпретации, выбора оптимальной формы представления объяснений и так далее. Главной целью проекта была доработка ML-сервиса Центра искусственного интеллекта (ЦИИ) НИУ ВШЭ с добавлением возможности интерпретации предсказаний моделей, обученных на структурированных данных.

В ходе реализации было сделано следующее:

- Обзор существующих методов интерпретации моделей на структурированных данных;
- Изучение и критический обзор имеющейся кодовой базы ЦИИ;
- Рефакторинг бекэнда сервиса ML-моделей: переход от монолитной архитектуры к набору микросервисов;

Работа структурирована в соответствии с этими этапами. Помимо этого, приводится пример возможного взаимодействия между фронтендом и оркестраторов моделей машинного обучения, а также формулируются пути дальнейшего развития нового сервиса.

Методы интерпретации

Перед переходом к практической реализации сервиса ML-моделей рассмотрим потенциальные методы интерпретации моделей на структурированных данных, выводы которых могут использоваться в новом сервисе. Будем рассматривать два метода: Local Interpretable Model-agnostic Explanations (LIME) и Shapley Additive Explanations (SHAP), выбор которых обусловлен следующими соображениями:

- Во-первых, рассматривались методы с возможностью *локальной интерпретации*, т. е. такие методы, которые объясняют предсказания модели на конкретном наблюдении из выборки, а не общие закономерности, установленные моделью на данных. Это объясняется принципом работы самого сервиса – пользователь подает на вход конкретную запись и заинтересован в получении интерпретации применительно к ней, тогда как обучающая выборка имеет для него второстепенное значение.
- Во-вторых, оба рассматриваемых метода являются независимыми от конкретного класса моделей машинного обучения (SVM, логистические регрессии, решающие деревья, ансамбли, нейросети и т. п.) и решаемой задачи (регрессия или классификация). Это требование естественным образом следует из того факта, что сервис содержит модели различных классов, и никаких предпосылок к фокусированию на интерпретации лишь одного из них нет.
- Выбор методов носит рекомендательный характер: структура разработанного сервиса не накладывает ограничений на конкретный вид интерпретации, поскольку она определяется отдельно для каждой модели. Для интеграции иных методов интерпретации достаточно лишь, чтобы они имели визуальное представление, которое легко интегрируется в пользовательский интерфейс сервиса ML-моделей.

Local Interpretable Model-agnostic Explanations (LIME)

В основе LIME лежит идея о том, что сложные модели можно аппроксимировать простыми и интерпретируемыми (например, линейной регрессией) в окрестности одного наблюдения, так, как это делается при линейной аппроксимации сложных функций в математическом анализе. Подход был

предложен в 2016 году [1] и с тех пор стал одним из наиболее часто используемых в силу своей универсальности.

В рамках подхода модель представляется в виде «черного ящика», получающего на вход некоторые данные и дающего конкретное предсказание. На основе наблюдения, подлежащего интерпретации, генерируется случайная выборка его искаженных значений (например, в случае числовых данных берется выборка из случайного распределения с центром в точке наблюдения). Новый набор данных обучает простую модель, взвешенную по близости семплированных наблюдений к исходному.

Простой моделью, используемой для аппроксимации, может быть любая интерпретируемая модель (например, линейная регрессия или дерево решений). При этом предполагается, что обученная модель является хорошим приближением работы сложной модели локально, т. е. в окрестности наблюдения.

В общем виде задача, решаемая методом LIME, выглядит следующим образом:

$$LIME(x) = \arg\min_{g \in G} L\,(f, g, \pi_x) + \Omega(g),$$

где x – наблюдение, подлежащее интерпретации; f – исходная сложная модель; g – простая модель, аппроксимирующая «черный ящик»; π_x – размер окрестности x, который необходимо определить при семплировании; $\Omega(g)$ – сложность модели; L – некоторая функция потерь, определяемая в соответствии с задачей. В реальности оптимизация производится только по первой компоненте $L(f, g, \pi_x)$, тогда как учет «простоты» интерпретатора остается за пользователем. Также за человеком остается подход к формированию выборки: он отличается в зависимости от природы данных (табличные значения, тексты или изображения).

На Python существует реализация интерпретатора LIME, способная работать со всеми необходимыми типами моделей и входных данных [2].

Shapley Additive Explanations (SHAP)

SHAP (Shapley Additive exPlanations) — это теоретико-игровой подход к объяснению результатов любой модели машинного обучения. Он основан на т. н. значениях Шепли, которые являются результатом решения задачи оптимального распределения выигрыша между игроками в кооперативных играх. Сам подход был

описан в 1951 году [3], тогда как его приложение к проблеме интерпретации моделей машинного обучения появилось существенно позже [4].

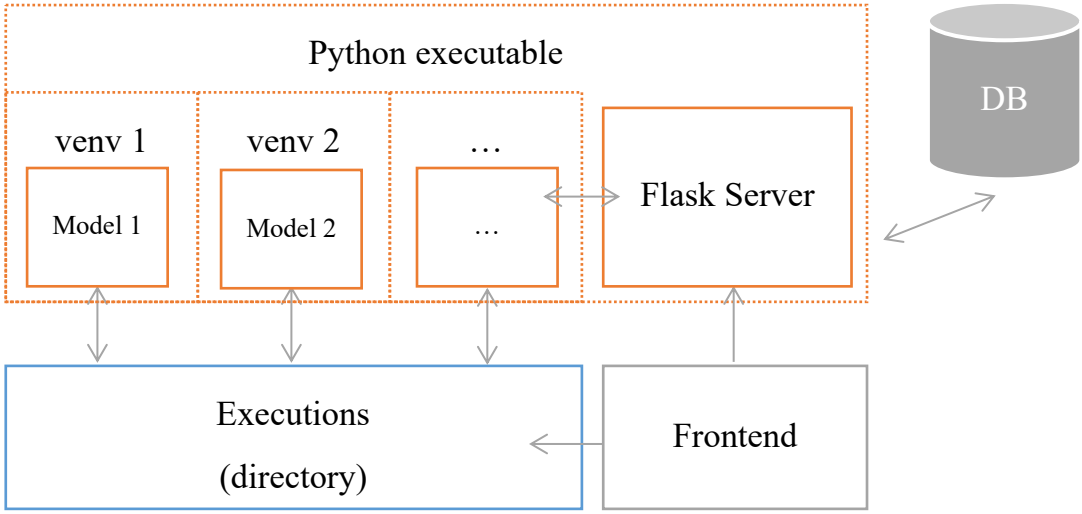
Данный метод позволяет получать вклад каждого отдельного признака в ответ модели. Как и в случае с LIME, данный метод подходит для любых типов данных, однако не требует подбора способа трансформации наблюдений для генерации случайной выборки. Однако принцип работы алгоритма основывается на переборе 2^n комбинаций признаков для оценки их индивидуальных вкладов, что трудно назвать вычислительно приемлемым. В настоящее время используются приблизительные методы расчета значений Шэпли [5].

Существующая реализация на Python [6] также предоставляет инструменты для наглядного представления SHAP-значений, что важно в контексте создания сервиса, ориентированного на неподготовленного пользователя. К сожалению, не все типы моделей в полной мере поддерживаются этой реализацией – покрытие библиотеки LIME в этом плане намного более полное. В частности, из-за высокой размерности TF-IDF трансформации не удалось получить SHAP-интерпретацию моделей на текстовых данных, которые присутствовали в исходной версии сервиса Центра искусственного интеллекта ВШЭ.

Сервис ML-моделей

В практическом плане предполагалось дополнить имеющийся сервис ML-моделей Центра искусственного интеллекта (ЦИИ) НИУ ВШЭ функционалом интерпретации. Этот сервис хранил ряд предобученных моделей машинного обучения и предоставлял простой API-интерфейс для получения выводов этих моделей на основе загружаемых данных. Архитектурно он представлял собой большой монолит на базе Python. Приблизительная схема сервиса приведена на рисунке 1.

Рис. 1: Схема исходного сервиса ML-моделей



Такая реализация сервиса имела существенные недостатки:

- *Низкая скорость работы.* При каждом запросе предсказания создавалось новое виртуальное окружение Python и считывался pickle-файл с моделью.
- *Избыточность.* Виртуальные окружения создавались прямо внутри Python скрипта, что усложняло код и приводило к нагромождению виртуальных окружений в рамках одного проекта.
- *Хаотичное управление зависимостями.* Не предоставлялось возможности использовать разные версии Python для запуска моделей.
- *Неуниверсальность.* В разных моделях использовались разные способы обращения к ним, и не предлагалось никакого универсального интерфейса получения предсказаний.

- *Сложность развертки.* Запуск сервиса был невозможен без ручного выполнения действий, которые в нормальных условиях не требуют участия человека и могут быть автоматизированы.
- *Сложность доработки.* Из-за строгой взаимозависимости всех компонентов системы любое дополнение требовало изменений в различных частях программы.

При этом дополнение сервиса новым функционалом означало бы еще большее усложнения структуры проекта и ухудшение и так сомнительных показателей качества масштабирования и поддержания проекта. *Поэтому было принято решение полностью переработать архитектуру и кодовую базу сервиса, что и легло в основу данной работы*.

Новый сервис моделей и интерпретации

В данном разделе будет описан новый сервис, отвечающий за получение ответов ML-моделей, а также имеющий функционал интерпретации этих ответов. Требования, предъявляемые к этому продукту, можно свести к следующим:

- Родство с legacy-сервисом, а именно: сохранение и перенос имеющихся моделей на структурированных данных и адаптация их под новые нужды; преемственность общей логики работы API; легкость интеграции с frontend.
- Единый подход к хранению и запуску моделей и унификация интерфейсов взаимодействия с ними. Подход должен быть универсальным и применяться к любым моделям машинного обучения, в том числе и на неструктурированных данных.
- Полное разведение кодовой базы отдельных моделей и самого сервиса, а также контейнерная виртуализация с заделом на дальнейшее масштабирование продукта.
- Простота развертывания на удаленных серверах, в первую очередь на unix-подобных системах.
- Возможность легко и быстро усложнять структуру системы, добавляя новые компоненты, заменяя или трансформируя существующие и т. д. Это особенно актуально с учетом активного развития моделей, создаваемых в Центре искусственного интеллекта, и увеличения нагрузки на сервис.

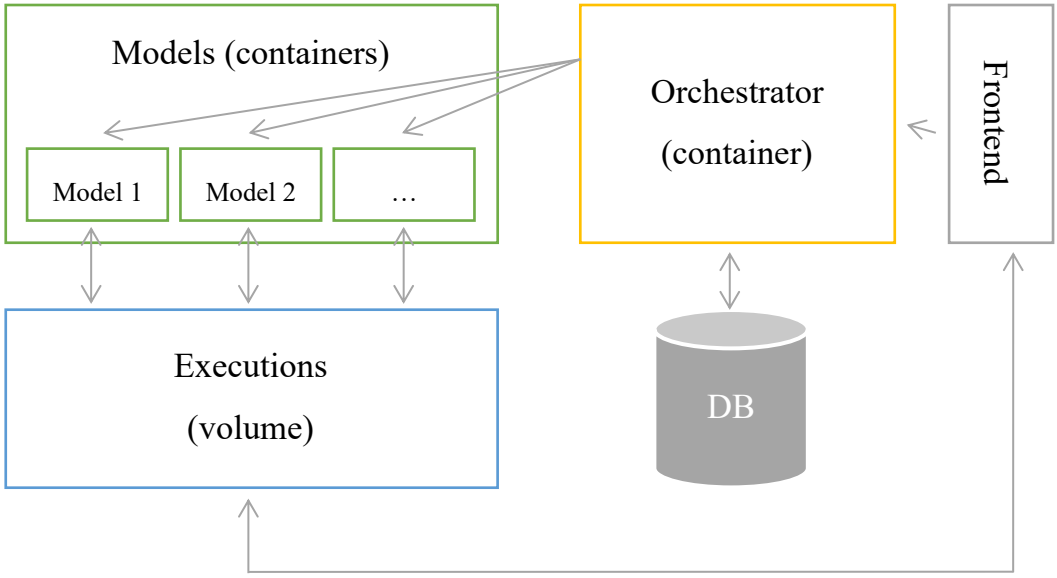
Очевидным выводом из этих требований была необходимость реализовать сервис в парадигме *микросервисной архитектуры*, которая уповает на создание распределенной системы, где каждый элемент отвечает за небольшую часть общего функционала и при необходимости может работать независимо от остальных элементов. В настоящее время переход от монолитной архитектуры к микросервисной получает все большее распространение в отрасли, поскольку он обеспечивает существенно большую стабильность и отказоустойчивость разрабатываемых продуктов.

В рамках этой работы микросервисы позволяют избавиться от слабых мест предыдущей реализации, описанной ранее. Еще одним преимуществом этого подхода является то, что он требует структурного понимания работы всей системы и ее отдельных компонентов, а значит в разы упрощается документирование сервиса и командная разработка.

Общая структура сервиса

Вкратце рассмотрим устройство нового сервиса, разработанного с учетом вышеперечисленных требований.

Рис. 2: Схема нового сервиса ML-моделей и интерпретации



На рисунке 2 приведена схема компонентов сервиса и их взаимодействия. Можно выделить четыре основные функциональные части проекта: *оркестратор*, отвечающий за управление имеющимися моделями и взаимодействие с БД; сама *база данных,* хранящая текущее состояние системы; *ML-модели*, потенциально имеющие абсолютно разные классы, но имеющие идентичный интерфейс; а также *хранилище данных* (в этом случае – директория executions в файловой системе сервера), в котором содержатся входные и выходные данные запусков моделей, а также интерпретации.

Модели

Модельная компонента нового сервиса во многом основывается на том, что было реализовано в исходном сервисе. Он содержал несколько моделей, работающих как со структурированными, так и с неструктурированными данными. В рамках этой работы под новый фреймворк были адаптированы только две модели на структурированных данных, однако принципы этой адаптации легко накладываются и на остальные модели.

В исходном виде каждая модель представляла собой Python-модуль, который содержал функции для загрузки предобученных моделей из pickle-файлов, а также

скрипт, инициализирующий подходящее виртуальное окружение и использующий функции модуля для получения предсказаний модели для входных данных. Для реализации новой архитектуры необходимо было переработать проект модели таким образом, чтобы:

- Модель изначально работала из своего виртуального окружения, а не создавала новое внутри текущего;
- Все зависимости устанавливались автоматически;
- Общение с моделью производилось по API с запросами на вывод предсказания и интерпретации;
- Внешние пользователи (в том числе оркестратор) могли получить необходимую информацию о модели и ее API из конфигурационного файла;
- Можно было легко и быстро (в идеальном случае --- одной командой) запустить модель на новой машине.

Первые два пункта обеспечиваются благодаря использованию фреймворка Poetry [7]. Poetry — это инструмент для управления зависимостями и проектами в Python. Он позволяет объявить библиотеки, от которых зависит проект, и автоматически управлять (устанавливать/обновлять) эти библиотеки. В основе работы с Poetry лежит т. н. lockfile, обеспечивающий легкое развертывание проекта на любых машинах. Более того, Poetry автоматически создает виртуальное окружение с соответствующей требованиям проекта версией Python. Пример конфигурации Poetry см. в приложении 2.

Третий пункт достигается за счет реализации простого API на базе библиотеки Flask [8], интерфейс которого будет описан далее.

Конфигурационный файл в формате YAML (пример см. в приложении 1), создаваемый для модели, позволяет получить информацию о ее названии, типе решаемой задачи, а также доступности метода интерпретации. Эта информация используется в том числе для популяции таблицы моделей в БД сервиса.

Наконец, легкое развертывание моделей обеспечивается контейнеризацией средствами Docker — open-source платформы для разработки и запуска приложений [9]. Docker позволяет отделять приложения от остальной инфраструктуры и в настоящее время по сути является стандартом отрасли.

Разработанный подход к «оборачиванию» моделей в API позволяет легко и быстро адаптировать существующие модели к требованиям нового сервиса, а также предоставляет понятный путь к разработке и добавлению новых моделей.

API моделей

`/infer/<exec_id:string>`

Получение выводов модели на входных данных соответствующего запуска. На бэкенде метод API вызывает функции для предобработки данных и предсказания, специфичные для каждой модели.

`/explain/<exec_id:string>`

Создание интерпретации предсказаний модели на входных данных соответствующего запуска. Конкретный метод интерпретации зависит от модели и реализовывается вне Flask модуля (например, наравне с функциями для получения предсказаний или загрузки пайплайнов ML-моделей).

Оркестратор

Оркестратор – это ядро нового сервиса, который предоставляет API для обращения к моделям, управляет запусками (executions) и обновляет информацию о состоянии системы в базе данных. Иными словами, он является промежуточным звеном между моделями и произвольным фронтендом.

В процессе разработки оркестратора использовались те же технологии, что и для оборачивания моделей в независимые модули: управление зависимостями при помощи Poetry, Flask для определения RESTful API и Docker для виртуализации. Также оркестратор – единственный элемент системы, который должен взаимодействовать со фронтендом.

API оркестратора

`/exec/init/`

Параметры:

- model_id – id модели, которая должна использоваться для запуска.

- exec_id – id запуска (по совместительству - название директории с данными в хранилище)

`/inference/<exec_id:string>`

Вызов метода infer модели, соответствующей данному запуску.

`/explanation/<exec_id:string>`

Вызов метода explain модели, соответствующей данному запуску.

Также были реализованы дополнительные методы для отладки: get_models и get_executions, позволяющие получить все записи в таблицах models и executions.

База данных

База данных, используемая в проекте для хранения информации о состоянии системы, реализована при помощи SQLite [10]. Выбор этой СУБД релевантен в контексте реализации MLP сервиса и соответствует базовым требованиям к БД, в дальнейшем же запланирован переход на более сложные фреймворки, такие как PostgreSQL [11].

База данных состоит из двух таблиц: models и executions. Models содержит информацию о всех доступных моделях и наполняется из конфигурационных YAML файлов. Executions хранит данные о «запусках», т. е. связках «модель – файл входных данных», которые используются для получения предсказаний и интерпретаций. Структуры данных сущностей описаны в таблицах 1 и 2.

Таблица 1: структура таблицы models

Поле	Тип данных	Атрибуты	Описание
id	int	primary_key=True autoincrement=True	Уникальный идентификатор модели.
task_type	string		Тип решаемой задачи (регрессия, классификация и т. п.)
task_class	string		Подтип решаемой задачи (напр., многоклассовая классификация)
name	string		Человекочитаемое название модели.

tech_name	string	«Техническое» название, или слаг.
data_type	string	Тип входных данных.
description	string	Произвольное описание модели.
has_explanation	bool	Флаг, имеет ли модель метод explain (интерпретация)
port	int	Порт, на котором модель слушает запросы от внешних потребителей.

Таблица 2: структура таблицы executions

Поле	Тип данных	Атрибуты	Описание
id	string	primary_key=True	Уникальный идентификатор запуска. Создается на стороне фронтенда.
model_id	int	ForeignKey("models.id")	ID модели, соответствующей запуску. Является внешним ключом из таблицы models.
state	string		Состояние запуска (инициализирован, выполнен, выполнен с ошибкой)
user	string		Имя пользователя, создавшего запрос на запуск.
datetime	timestamp	nullable=False server_default=func.now()	Время инициализации запуска.
log	string		Лог запуска
type	string		Тип запуска (inference, explanation или both)

***

Все элементы системы объединяются в единое целое при помощи Docker-compose – средства запуска мульти-контейнерных Docker-приложений. В конечном счете это позволяет запустить всю систему совместно при помощи одной команды в командной строке, без необходимости проверять наличие всех необходимых

зависимостей (в т. ч. нужных версий Python). В наиболее простом виде, запуск сервиса производится следующим образом:

```
docker compose up
```

При этом усложнение и расширение сервиса никак не ограничивается – Docker-compose может состоять из произвольного количества элементов.

Порты моделей открыты только внутри системы (т. е. только для оркестратора), тогда как к оркестратору можно обращаться извне. Конфигурационный файл Docker-compose для приложения с оркестратором и двумя моделями приведен в приложении 3.

Пример работы сервиса

Предоставим пример работы сервиса с предположением о наличии фронтенда, позволяющего выбрать интересующую модель и загрузить файл с данными.

1) *Создание и инициализация execution из фронтенда.*
   В хранилище (директория executions) создается директория с названием, которое затем будет использоваться как id запуска. Размещается файл с входными данными input.
   Запрос к API оркестратора:
   /exec/init с соответствующими параметрами: model_id и execution_id (= название директории)
   - Создается новая запись в таблице executions, сохраняется связка между запуском и необходимой для него моделью
2) *Получение предсказаний модели.*
   Запрос к API оркестратора:
   inference/<execution_id>
   - Оркестратор получает информацию о модели, соответствующей данному запуску, после чего обращается к API модели. Модель возвращает выходные данные в директорию хранилища с актуальным execution_id.
3) *Получение интерпретации модели.*
   Запрос к API оркестратора:
   explanation/<execution_id>
   - Оркестратор получает информацию о модели, соответствующей данному запуску, после чего обращается к API модели. Модель возвращает интерпретацию в директорию хранилища с актуальным execution_id.

Дальнейшие шаги

Несмотря на существенное качественное развитие по сравнению с первоначальной имплементацией сервиса ML-модели, данная реализация обладает несколькими существенными недостатками. В этом разделе мы наметим основные пути развития нового сервиса, а также приведем возможные решения наиболее острых проблем.

Отсутствие валидации действий фронтенда на стороне оркестратора

В текущей конфигурации предполагается полное доверие действиям, ожидаемым со стороны фронтенда: созданию директории запуска и добавлению файла с данными в верном формате. Оркестратор же при инициализации запуска лишь добавляет информацию о нем в базу данных, тогда как в угоду надежности всех узлов системы необходимо проверять как сам факт наличия данных, так и их корректность. Таким образом, требуются доработки как на стороне оркестратора (поиск директории, базовая проверка на валидность input-файла), так и на стороне отдельных моделей (проверка соответствия входных данных требуемому формату, дополнительные проверки качества данных).

Заметим, что для реализации этих изменений не требуется существенной переработки сервиса – они хорошо ложатся на уже созданные паттерны взаимодействия между микросервисами. Это еще раз доказывает адекватность разработанного продукта технической и бизнес-задаче.

Отсутствие выделенного хранилища

Как было описано ранее, в качестве хранилища входных и выходных данных используется файловая система локального или удаленного сервера. Единственное усложнение следует лишь из создания томов Docker и их использования разными компонентами системы. При этом в отрасли принято считать, что файлы, необходимые для работы ПО, не должны храниться на той же машине, что и код (или хотя бы находиться вне изолированной среды выполнения этого кода). Это необходимо, чтобы обеспечить сохранность информации при возможных технических неполадках.

Чтобы преодолеть эти ограничения, можно использовать объектные хранилища, такие как Minio [12] или S3 [13]. Они позволяют хранить данные различного типа и предоставляют универсальный интерфейс для их загрузки и

выгрузки. В контексте этого сервиса в хранилище было бы необходимо иметь входные и выходные данные, а также (в идеальном случае) файлы, хранящие архивы предобученных моделей – упомянутые ранее pickle-файлы.

Простота базы данных

Выбор SQLite в качестве СУБД для хранения состояний системы обусловлен простотой инициализации и адекватен в случае небольшой нагрузки на базу данных и небольшого количества хранящихся данных. При масштабировании сервиса будет необходимо вывести базу данных за пределы «ядра» сервиса в силу тех же причин, что свидетельствуют и в пользу создания отдельного объектного хранилища (см. выше). Потенциальным решением может быть СУБД с открытым исходным кодом PostgreSQL – одно из самых популярных решений, особенно для небольших проектов.

Более того, доработке может подлежать и сама структура сущностей и таблиц в БД. Например, можно усложнить понятие «запуска» и разделить его на два подтипа – inference и explanation.

Ограниченный набор методов API и их ответов

Для качественного взаимодействия с бэкендом, фронтенду может понадобиться большее количество методов обращения к оркестратору – например, для получения информации об отдельных запусках или моделях. Также требуется расширить перечень возможных ответов сервера и покрыть их документацией, чтобы любые ошибки на бэкенде могли правильно интерпретироваться на стороне клиента.

***

Таким образом, реализованный сервер хоть и соответствует минимальным требованиям, сформулированным на начальных стадиях разработки, все еще требует ряда доработок для обеспечения качественного и безотказного функционирования. На данный момент лишь заметим, что структура сервиса построена таким образом, чтобы любая из этих доработок требовала минимальных временных вложений. При этом сами эти доработки могут стать основой будущих работ.

Заключение

Данная работа была направлена на существенную переработку сервиса ML-моделей Центра искусственного интеллекта НИУ ВШЭ, а также на расширение его функциональности: реализацию возможности получить интерпретацию ответов модели. В ходе работы были рассмотрены существующие подходы к интерпретации моделей машинного обучения, после чего был выбран наиболее релевантный поставленной задаче способ – метод SHAP, основанный на практическом применении теории игр и значений Шапли.

Разработанный сервис существенно улучшает процесс взаимодействия с моделями. Он прост в развертке, полностью изолирует работу моделей и оркестратора, предоставляет единый подход к оборачиванию моделей в простой API с универсальным интерфейсом. При этом обеспечена преемственность со старым сервисом: имеющийся фронтенд легко адаптировать к работе с новым API, предобученные модели легко переиспользовать, сохранена структура хранилища с отдельными запусками (executions) как основными логическими единицами сервиса.

В ходе разработки использовались современные state-of-the-art подходы к разработке и развертыванию приложений. Каждый элемент системы запускается внутри Docker-контейнера, совместное же их взаимодействие обеспечивается при помощи Docker compose. API как моделей, так и оркестратора построены на принципах REST и реализованы при помощи библиотеки Flask.

Разумеется, у реализованного сервиса имеются существенные ограничения - он не полностью соответствует требованиям, которые обычно применяются к подобным продуктам в реальных условиях. Однако значимость проделанной работы состоит в устранении фундаментальных проблем предыдущего сервиса и определении дальнейших шагов к совершенствованию приложения. В дальнейшем будет проделана работа по расширению перечня доступных методов API оркестратора, дополнению сервиса выделенным хранилищем на базе s3 API, а также развертыванию независимой СУБД PostgreSQL.

Примечание: репозиторий с разработанным сервисом интерпретации доступен по адресу https://github.com/ivanskv2000/new-hse-ml-platform

Список литературы

1. Ribeiro M.T., Singh S., Guestrin C. “Why Should I Trust You?”: Explaining the Predictions of Any Classifier. arXiv, 2016.

2. Ribeiro M.T. Lime. https://github.com/marcotcr/lime.

3. Notes on the N-Person Game - II: The Value of an N-Person Game. RAND Corporation, 1951.

4. Lundberg S., Lee S.-I. A Unified Approach to Interpreting Model Predictions. arXiv, 2017.

5. Aas K., Jullum M., Løland A. Explaining individual predictions when features are dependent: More accurate approximations to Shapley values. arXiv, 2020.

6. Lundberg S. SHAP. https://github.com/slundberg/shap.

7. Poetry: Python packaging and dependency management made easy [Electronic resource] // GitHub. URL: https://github.com/python-poetry/poetry.

8. Flask [Electronic resource]. URL: https://flask.palletsprojects.com/en/2.3.x/.

9. Merkel D. Docker: lightweight Linux containers for consistent development and deployment // Linux J. 2014. Vol. 2014, № 239. P. 2:2.

10. Hipp R.D. SQLite [Electronic resource]. URL: https://www.sqlite.org.

11. Stonebraker M., Kemnitz G. The POSTGRES next generation database management system // Commun. ACM. 1991. Vol. 34, № 10. P. 78–92.

12. Minio [Electronic resource]. URL: https://github.com/minio.

13. Amazon S3 API Reference [Electronic resource]. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html.

Приложение

1. Конфигурационный YAML файл модели

```yaml
---
# ML Model Configuration & Metadata

name: <Имя модели>
tech_name: <Техническое имя (слаг) модели>
description: <Описание>

task_type: <Тип решаемой задачи>
task_class: <Уточнение типа задачи>

data_type: <Тип входных данных>
has_explanation: <Реализован ли метод explain>
```

2. Конфигурационный файл Poetry

```toml
[tool.poetry]
name = "orchestrator"
version = "0.1.0"
description = ""
authors = ["Ivan Skvortsov <iaskvortsov@edu.hse.ru>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.9"
Flask = "^2.3.2"
flask-restx = "^1.1.0"
Flask-SQLAlchemy = "^3.0.3"
SQLAlchemy-Utils = "^0.41.1"
PyYAML = "^6.0"
coolname = "^2.2.0"
watchdog = "^3.0.0"
requests = "^2.31.0"
```

3. Файл Docker-compose финальной совокупности сервисов

```yaml
version: '3.8'

services:
  orchestrator:
    container_name: orchestrator
    restart: always
```

```
    build: ./orchestrator
    volumes:
      - ./orchestrator/:/app
      - ./${MODELS_DIR}/:/ml_models
      - ./${EXEC_DIR}:/executions
    environment:
      - PORT=${ORCHESTRATOR_PORT}
    ports:
      - ${ORCHESTRATOR_PORT}:${ORCHESTRATOR_PORT}
    depends_on:
      - nlp_nl2ml
      - nlp_sentiment_tfidf_lr
    expose:
      - ${ORCHESTRATOR_PORT}
  nlp_nl2ml:
    container_name: nlp_nl2ml
    restart: always
    build: ./${MODELS_DIR}/nlp_nl2ml
    volumes:
      - ./${MODELS_DIR}/nlp_nl2ml/:/model
      - ./${EXEC_DIR}:/executions
    environment:
      - EXEC_DIR=${EXEC_DIR}
    ports:
      - "5430"
    expose:
      - "5430"
  nlp_sentiment_tfidf_lr:
    container_name: nlp_sentiment_tfidf_lr
    restart: always
    build: ./${MODELS_DIR}/nlp_sentiment_tfidf_lr
    volumes:
      - ./${MODELS_DIR}/nlp_sentiment_tfidf_lr/:/model
      - ./${EXEC_DIR}:/executions
    environment:
      - EXEC_DIR=${EXEC_DIR}
    ports:
      - "5430"
    expose:
      - "5430"
```