

ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: _____ Торговая система для криpto-бирж _____

(промежуточный, этап 2)

Выполнили:

Студент группы БПМИ209 _____ Л.В.Прокопчук
Подпись _____ И.О.Фамилия

Студент группы БПМИ209 _____ Л.И.Рыбаков
Подпись _____ И.О.Фамилия

Студент группы БПМИ209 _____ И.Ю.Бондырев
Подпись _____ И.О.Фамилия

17.02.2022

Дата

Принял:

Руководитель проекта _____ Казаков Евгений Александрович
Имя, Отчество, Фамилия

разработчик

Должность, ученое звание

Facebook inc.

Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 17.02 2022 10
Оценка (по 10-ти бальной шкале) _____ Подпись _____

Москва 2022

Аннотация

Уважаемые члены комиссии!

Вашему вниманию представляется программный проект по теме "Система для высокочастотного трейдинга на криптовалютных биржах".

В наше время криптовалюта все сильнее укрепляется как средство платежа во всем мире. Большие объемы торговли и высокая волатильность неизбежно приводит к многочисленным возможностям заработать на изменении курса и, как следствие, большому количеству желающих воспользоваться этими возможностями.

Целью нашей работы является создание автоматизированной трейдинг системы с положительной прибылью. Мы провели исследование существующих стратегий и реализовали наши вариации некоторых из них.

Промежуточным результатом нашей работы можно назватьирующую программу, способную по сигналу размещать заказы на бирже и даже в некоторых случаях закрывать позицию в плюс, однако из-за повышенной волатильности криптовалют в последнее время и еще нескольких причин, в среднем наша система уходит в минус.

Содержание

1 Введение	3
1.1 Актуальность проблемы	3
1.2 Цели и задачи	4
1.2.1 Цель	4
1.2.2 Задачи	4
1.3 Статьи	4
1.4 Документация бирж	4
1.5 Документация библиотек	4
1.6 Аналоги	4
1.7 Описание функциональных требований к программному проекту	5
1.7.1 Коннектор к бирже	5
1.7.2 Сбор данных	5
1.7.3 Измерение скорости соединения	5
1.7.4 Машинное обучение	6
1.8 Описание нефункциональных требований к программному проекту	6
1.8.1 Безопасность	6
1.8.2 Отказоустойчивость	6
1.8.3 Скорость	6
1.8.4 Переиспользование кода	6
1.8.5 Масштабируемость	6
2 Коннекторы	6
2.1 BinanceConnector	6
2.2 DydxConnector	7
3 Market Making	8
3.1 Что это такое	8
3.2 Как это работает	9
3.3 Реализация	9
3.4 Результаты	10
3.5 Выводы	10
4 Индикаторы	10
4.1 <code>fill_target_values</code>	10
4.2 <code>fill_features_values</code>	10

5 Задача классификации	11
5.1 CatBoost	11
5.2 Данные	11
5.2.1 Датасет	11
5.2.2 Парсинг	11
5.3 Использование CatBoost	12
5.3.1 Fit-Predict	12
5.3.2 Автоподбор гипер-параметров	12
5.4 FPR/FNR	13
5.5 Важность индикаторов	13
5.6 Распределение индикаторов	14
5.7 Результат	16
6 Арбитражная стратегия	16
6.1 Описание стратегии	16
6.2 Выбор биржи и инструмента	16
6.3 Обнаружение скачков с помощью скользящего окна	16
6.4 Анализ исторических данных, реагируем на все сигналы	17
6.4.1 Class Profit Calcualtor	17
6.4.2 Сбор сигналов	17
6.4.3 Получение трейдов и расчет прибыли	18
6.5 Анализ исторических данных, попытка отсеять плохие сигналы	19
6.5.1 Сбор фичей	19
6.5.2 Узнаем, был ли сигнал теоретически прибыльным	19
6.5.3 Пытаемся отличить плохие сигналы от хороших	20
7 Трейдер	22
7.1 Использованные технологии	23
7.1.1 Асинхронная функция	23
7.1.2 <code>asyncio.Event</code>	23
7.2 Устройство трейдера	24
7.2.1 <code>listen_binance</code>	24
7.2.2 <code>account_listener</code>	24
8 Дополнительные результаты	25
8.1 Визуализация	25
8.2 Контролирующий бот	25
8.3 CI/CD, тесты, линтер	25
8.4 Смарт контракты	25

1 Введение

1.1 Актуальность проблемы

Сегодня, кого не спроси, все знают, что такое биткоин, или, по крайней мере, говорят, что знают. Разговоры о криптовалютах и их производных в мире не утихают уже несколько лет, но, к сожалению, большинство из них очень поверхностные и не доходят даже до обсуждения принципа работы блокчейна в общих словах. Несмотря на кажущуюся сложность устройства, 2 самых больших блокчейна (Bitcoin и Ethereum) критически неправляются с нагрузкой, возложенной на них желающими воспользоваться криптовалютой. Из-за того, что сеть Эфириума может обрабатывать лишь 15 транзакций в секунду, комиссия, которую нужно заплатить, чтобы транзакция была одной из этих 15, доходит до \$70, что делает любой токен на блокчейне непригодным для использования в качестве обменной валюты. Чтобы снизить нагрузку на майннет¹, были разработаны и все еще разрабатываются несколько альтернативных решений, которые одним словом называются Layer-2 решения. Это надстройки над блокчейном, которые увеличивают пропускную способность и скорость в

¹Основная сеть блокчейна, на которой криптовалюта имеет реальную стоимость. Есть также сети для тестирования разработок. На них валюта можно получить по запросу от специальных адресов

ущерб децентрализованности. Мы считаем, что пока не придумали более изящного способа достичь тех же результатов, которые дают L2 решения, данная технология будет развиваться, а актуальности нашей темы будет расти.

1.2 Цели и задачи

1.2.1 Цель

Написать трэйдинг систему, которая сможет стабильно выходить в плюс.

1.2.2 Задачи

- Проведение исследований по стратегиям трэйдинга.
- Проверка работоспособность стратегий.
- Создание инфраструктуры, позволяющей взаимодействовать с биржей автоматизированно.
- Сбор данных и обучение модели.
- Написание программы, совершающей сделки.
- Обзор и сравнительный анализ источников и аналогов

К сожалению, выбранная нами тема мало освещается в источниках любого вида: никто не захочет делиться прибыльной стратегией. Многое нам приходилось и придется делать с нуля.

1.3 Статьи

Тем не менее, существуют статьи, описывающие некоторые возможные подходы к написанию алгоритмов HFT. Например, есть ресурс [?], на котором описывается стратегия маркет-мейкинга, аналог которой мы попытались реализовать. Но материалы такого рода, находящиеся в открытом доступе, с течением времени теряют свою актуальность: если большое количество участников рынка придерживается одной схемы действий, то вскоре она перестает приносить прибыль. По этой причине мы старались не ориентироваться на подобные источники.

1.4 Документация бирж

Основным же источником информации для нас служила документация API [?] [?] бирж, к которым мы подключались. С помощью нее был написан коннектор, инкапсулирующий процесс подключения и взаимодействия с биржей, произведен сбор необходимой информации: список сделок за последний месяц, состояние о счете и т.п.

1.5 Документация библиотек

Для машинного обучения мы использовали CatBoost [?] — это библиотека от Яндекса для градиентного бустинга, надстройки над решающими деревьями. КэтБуст для нас лучшее решение, потому что это самая быстрая библиотека для классификации среди аналогов и проста в использовании.

1.6 Аналоги

На крипто валютном рынке существует множество торговых ботов, но информации об их характеристиках и принципах работы практически нет. Мы можем судить об их доходности, лишь по каким-то сомнительным заявлениям или косвенным признакам. В открытом доступе в основном находятся боты, которые предоставляют лишь интерфейс взаимодействия с биржей [?] [?]: “ручная” покупка и продажа токенов, выставление лимитных ордеров и т.п.. Такие решения не представляют для нас никакого интереса.

Так достойных аналогов в свободном доступе нет, функциональные и нефункциональные требования нам пришлось придумывать самим. К счастью, они достаточно интуитивны.

1.7 Описание функциональных требований к программному проекту

1.7.1 Коннектор к бирже

В программе должны быть коннекторы к биржам. Это класс, в конструктор которого подаются приватные ключи кошельков. После этого можно работать с биржей: смотреть информацию о счете, валютаю, отправлять и отменять ордера. Надо реализовать функционал, который предоставляет апи, чтобы можно было его использовать в трейдинг-стратегиях.

Примеры использования функций класса, который способен взаимодействовать с биржами по API:

- Отправка ордеров

```
connector.send_order(  
    symbol=ETH_USD, side=BUY, price=1, quantity=0.1  
)
```

- Получение текущих позиций

```
connector.get_our_positions(  
    opened=True, symbol=ETH_USD  
)
```

- Информация о конкретном рынке

```
connector.get_symbol_info(market=ETH_USD)
```

1.7.2 Сбор данных

Нужно обеспечить удобный механизм сбора исторических данных с бирж, на которых будет тестироваться и обучаться система. Удобно реализовать такие функции в коннекторе, ведь у него уже есть соединение с биржей.

Пример функции для сбора данных:

- Получение трейдов за определенный промежуток времени

```
connector.get_historical_trades(  
    market=BTC_USD,  
    begin="2021-12-12T09:00:00",  
    end="2021-12-12T12:00:00"  
)
```

Также в коннекторе может быть функционал, позволяющий собирать данные в прямом эфире и сохранять их в архив, если эти исторические данные не предоставляет биржа.

1.7.3 Измерение скорости соединения

Так как счет идет на миллисекунды, мы всегда должны иметь четкое представления, какое время у нас займет отправка и получения пакета данных. Для этого должен быть предусмотрен отдельный модуль, который будет замерять скорость соединения с различными сервисами. Он тоже основывается на коннекторе, так как он уже умеет соединяться с биржей.

Примеры использования функций класса для измерения скорости коннектора:

- Измерение скорости

```
speed_measure = SpeedMeasure(connector)  
speed_measure.get_connector_funcs_exec_times(  
    market=ETH_USD,  
    side=BUY,  
    iters_num=10,  
    filename="connector_funcs_exec_times.json",  
)
```

- Измерение задержки до биржи

```
speed_measure.get_orders_processing_delays(
    market=ETH_USD,
    side=BUY,
    orders_num=10,
    filename="orders_processing_delays.json",
)
```

1.7.4 Машинное обучение

Индикаторов, по которым можно строить прогнозы, очень много, поэтому ручными методами не получится подобрать правильное соотношение весов. Здесь нужно машинное обучение, которое принимает на вход пред обработанные данные сделок, а на выход выдает модель, которую можно использовать в трейдинг-стратегии. Важно, чтобы модель была устойчива к тому, что в датасет добавляется или убирается индикаторы. Примеры использования

1.8 Описание нефункциональных требований к программному проекту

1.8.1 Безопасность

Финансы — чувствительная тема, поэтому наша программа не должна допускать утечек данных о кошельках и приватных ключах. Нужно обеспечить безопасное хранение.

1.8.2 Отказоустойчивость

Во время трейдинга торговая система получает сотни обновлений от разных бирж, их обрабатывает, строит прогнозы и торгует. Нужно сделать так, чтобы система была готова к большим нагрузкам, и поведение было однозначно определено. Еще нужно проработать быстрое отключение торговой системы от торгов, если ее поведение станет неадекватным, и можно было бы быстро закрыть открытые заявки.

1.8.3 Скорость

В трейдинге важна каждая миллисекунда, поэтому цель — минимизировать время обработки, отправки и принятия данных.

1.8.4 Переиспользование кода

Нужно выстроить архитектуру проекта так, чтобы можно было быстро и легко тестировать свои гипотезы, поэтому код, который есть в проекте, должен быть написан так, чтобы его можно было легко понять и переиспользовать в других местах.

1.8.5 Масштабируемость

Система должна быть расширяема на несколько бирж и потенциально работать с большим количеством предсказательных моделей.

2 Коннекторы

Чтобы программа могла торговать, ей нужно подключиться к бирже. У выбранных нами бирж есть python-клиенты (python-binance и dydx-v3-python), позволяющие реализовать соединение. Итак, в нашем проекте есть 2 класса-коннектора, BinanceConnector и DydxConnector, с достаточно похожим функционалом.

2.1 BinanceConnector

[BinanceConnector](#)

Определение 1. Инструмент – пара торгующихся на бирже валют

Определение 2. Стакан – таблица лимитных заявок на покупку и на продажу

В конструкторе коннектор получает полученные из личного кабинета binance публичный и приватный ключи API для идентификации пользователя и список интересующих инструментов. Библиотека python-binance предоставляет класс синхронного клиента, Client. Методы этого класса используются в следующих функциях коннектора:

- `get_all_balances` – получение информации о балансе аккаунта.
- `get_cached_symbol_info` – получение информации о маркете (например tick-size цены и количества)
- `get_cached_order_book` – получение состояния стакана (для реализации существует отдельный класс, кэширующий приходящие обновления стакана)
- `send_limit_order/send_ioc_order` – создание ордера. Больше функций по отправке ордера не реализовано, потому что коннектор бинанса в последствии использовался только для мониторинга данных.
- `cancel_order` – отмена ордера по id, которое мы задали при создании.
- `get_*historical*_trades` – сбор исторических данных по проведенным сделкам.
- и т. д.

Также в python-binance есть классы AsyncClient и BinanceSocketManager для оформления подписки на совершенные сделки и на обновления стакана. Ключевая функция – `_subscribe_exchange_data`, которая подписывается на совершенные сделки и на обновления стакана для каждого указанного в конструкторе маркета и начинает ждать и обрабатывать приходящие апдейты. Функция асинхронная, потому что пока получаем обновления, мы хотим иметь возможность вовремя отправить ордер, или наоборот его отменить.

Для того, чтобы было удобно "слушать" информацию через коннектор, реализован концепт функции-listener.

Обозначение 3. Листенер – функция, которая вызывается из коннектора при получении им обновлений. Листенеры "слушают" обновления.

- `add_*_listener` – добавление кастомного листенера в список. При получении коннектором соответствующего обновления, все листенеры из этого списка будут вызваны с апдейтом в аргументе.
- `_call_*_listeners` – вспомогательная функция. Вызывается при получении коннектором соответствующего обновления. Вызывает все добавленные листенеры с апдейтом в аргументе функции.

Функции `start` и `_async_start` используются для активации коннектора, подписки на интересующие потоки и начала получения обновлений.

2.2 DydxConnector

DydxConnector

Коннектор для dydx очень похож на BinanceConnector, но более приспособлен к торговле, так как именно через него мы размещали ордера.

В коннекторе для dydx, как и binance, создается элемент класса `Client` с несколькими аргументами. Для удобства задания полей коннектора есть вспомогательный датакласс `Network` и список элементов этого класса с заданными полями для сетей `mainnet` и `ropsten`. Так, при создании коннектора, в аргументах нужно всего лишь указать название сети. Для тестирования используется сеть `ropsten`, для реальных торгов - `mainnet`.

В целом, интерфейс коннектора для dydx очень похож на интерфейс коннектора для binance. Тут тоже есть функции, вызывающие синхронные методы клиента:

- `get_symbol_info` – получение информации о маркете
- `get_order_book` – получение состояния стакана
- `get_our_positions/get_our_orders` – получение информации о имеющихся позициях и выставленных нами ордерах

- `send_*_order` – создание ордера. DydxConnector лучше приспособлен к торговле, поэтому реализовано 4 вида функции по созданию ордера. `send_limit_order` - лимитная заявка; `send_trailing_stop_order` – stop-loss ордер, который переставляется ближе к рыночной цене, если она идет в благоприятную для нас сторону; `send_take_profit_order` - лимитная заявка, использующаяся для фиксирования прибыли; `send_market_order` - ордер по рыночной цене.
- `cancel_order/cancel_all_orders` – отмена ордера по id, заданному нами при создании/отмена всех наших ордеров.
- `get_historical_trades` – сбор исторических данных по проведенным сделкам.
- и т. д.

Для подписки и прослушивания обновлений используется python-модуль `websockets` и, как и в `BinanceConnector`, функции-listener.

- `add_orderbook/trade/account_listener` - добавление листенера на обновления стакана/сделок/аккаунта.
- `add_orderbook/trade/account_subscription` - добавление запроса на подписку на обновления стакана/сделок/аккаунта в список запросов на подписку.
- `_call_orderbook/trade/account_listener` - вспомогательная функция. Вызывается при получении коннектором соответствующего обновления. Вызывает все добавленные листенеры с апдэйтом в аргументе функции.

Таким образом, если, например, программе нужно прослушивать обновления стакана для маркета `MARKET_ETH_USD`, то перед запуском коннектора функцией `start` или `async_start` нужно добавить соответствующие листенеры и подписку.

```
#! \bin\python3
...
add_orderbook_listener(orderbook_listener)
add_orderbook_subscription(MARKET_ETH_USD)
...
```

Есть аналогичная с `BinanceConnector` асинхронная функция `subscribe_exchange_data`. В ней сначала оформляются подписки на интересующие нас обновления путем отправки запросов из заполненного ранее списка запросов на подписку. Затем функция начинает получать обновления и вызывать соответствующие листенеры.

Если `dydx` фиксирует слишком большую активность от одного клиента, то клиент получает трехсекундный бан. Во время бана отправление и отмена ордеров невозможна, что может привести к потере реальных денег. Для избежания таких ситуаций был реализован декоратор `safe_execute`, который в случае поимки исключения при исполнении функции будет пытаться исполнить ее до 50 раз, пока функция не завершится корректно.

3 Market Making

3.1 Что это такое

Когда человек приходит на биржу, он хочет купить актив по рыночной цене. Если биржа *A* продает биткоины по 11k\$, а биржа *B* по 10k\$, то выгоднее покупать биткоины у *B*. Чтобы *A* не терять клиентов, она пользуется услугами маркет-мейкеров.

Маркет-мейкеры решают проблему дисбаланса цен между биржами. Они могут за вознаграждение от биржи *A* продать у них биткоины и выровнять курс до 10k\$, и тогда всем остальным будет снова выгодно торговать на бирже *A*. Аналогично работает, когда на бирже *A* курс ниже относительно других бирж: маркет-мейкеры закупятся биткоинами. В стабильное время маркет-мейкеры занимаются поддержанием маленького спреда, то есть делают так, что цены покупки и продажи отличаются как можно меньше.

На децентрализованных биржах все еще интереснее. Так как там биткоин не привязан к бирже, то стратегия выше выгодна для маркет-мейкеров, потому что они могут на бирже A продать биткоины по $11k\$$, и купить на B по $10k\$$, заработав разницу в ценах, пока они не выровняются.

Однако межбиржевые операции очень дорогие и долгие, поэтому чаще всего маркет-мейкеры зарабатывают на резких инертных скачках рынка и вознаграждения от биржи.

Для второго нужно очень много денег, так что попробуем заработать на первом.

3.2 Как это работает

Заработать можно в предположении краткосрочно высокоскорочного инерционного рынка.

Определение 4. Пробитием будем называть ту заявку, которую мы не успели отменить, и по ней у нас открылась позиция.

Алгоритм 5. *Market-Making orders*

1. Выставляем лимитные заявки в обе стороны на $\pm\Delta_1$ от индекс-цены.
2. Когда индекс-цена изменяется, переставляем заявки на ту же самую $\pm\Delta_1$, но уже от новой цены.
3. Если переставиться успели, и нас не пробили, то переходим к шагу 2
4. Неумалая общности нас пробили на покупку по цене p . Отменяем все заявки
5. Выставляем заявку на продажу на Δ_2 от цены, по которой нас пробили на покупку
6. Когда заявка исполнилась, возвращаемся к шагу 1

В итоге на шаге 6 мы заработка $\Delta_2 \cdot p$.

3.3 Реализация

Стратегия в репозитории

Программа работает в трех потоках, которые нужны для:

1. Выставление ордеров
 - (a) В этом потоке мы получаем новый трейды.
 - (b) Если цена нового трейда не такая, как у прошлого, и цена ордера, который надо выставить не такая, как у нас сейчас стоит, то мы отменяем текущие ордера и выставляем новые.
 - (c) Перестановка ордеров переходит не через сначала отмену, старых, а потом выставления новых ордеров, а непосредственно через переставление позиций аргументом `cancel_id` в функции `create_order`. Это позволило сократить количество запросов в два раза, а значит ошибка `Too many requests` будет встречаться реже.
2. Получение обновлений ордербука
 - (a) В предыдущем потоке во время выставления и отмены ордеров новые обновления по вебсокету не приходят, но обновления ордербука нам нужны, потому что по нему мы определяем цену ордеров.
 - (b) Поэтому получения ордербука вынесено в отдельный поток, чтобы всегда иметь свежий стакан.
3. Проверки положение ордеров
 - (a) Бывает так, что новые трейды не приходят, но окно спреда ордербука меняется.
 - (b) Нам важно наше окно трейдов держать строго на \pm какой-то спред вокруг бидсов и асков.
 - (c) Поэтому мы раз в какое-то время проверяем этот баланс, и если происходит дисбаланс, то мы обновляем наши ордера.

3.4 Результаты

Мы торговали на бирже DyDx. Она входит в топ самых популярных децентрализованных бирж. Количество сделок в месяц в ней около 30 тысяч и по биткоинку, и по эфиру. На самом деле это очень мало: в среднем 0.3 сделки в секунду, то есть одна сделка в 3-4 секунды. Соответственно рынок совсем не высоко-инерционный, поэтому после того, как нас пробили в одну сторону, до пробития во вторую, может пройти несколько десятков минут. Это превышает наш таймаут, потому что если мы слишком долго будем держать открытую позицию, то повышается вероятность, что рынок пойдет в обратную сторону и мы потеряем еще больше денег.

Еще часто бывали случаи, когда рынок отскакивает в нужную нам сторону, но на величину меньше комиссии, соответственно, мы теряем деньги.

3.5 Выводы

Если бы у нас была нулевая или близка к нулевой комиссия, то мы бы были в плюсе. Но с той, что у нас есть, в минусе.

4 Индикаторы

После маркет-мейкинга мы решили построить модель машинного обучения для предсказания направления рынка. Для построения модели мы использовали индикаторы, которые зависят от трэйдов в окне. Класс `Indicators` содержит функции, заполняющие значения фичей и столбца таргета: `fill_features_values` и `fill_target_values`.

4.1 fill_target_values

[Функция в репозитории](#)

Функция принимает словарь для заполнения значений, два окна трэйдов и параметры ордеров `stop_profit` и `stop_loss`. Функция считает, исполнились бы эти ордера и выставляет соответствующее значение в словарь для заполнения.

4.2 fill_features_values

[Функция в репозитории](#)

Обозначение 6. Окно длины t – окно, в котором есть все трэйды, случившиеся не раньше чем за t секунд до какого-то момента. Обозначаем как $window$.

Обозначение 7. $window[i]$ – i -ый элементы в окне.

Обозначение 8. $window[i].price$ и $window[i].volume$ – цена и объем i -й сделки в окне соответственно.

Функция принимает словарь для заполнения, окно трэйдов, и два списка вариантов числовых параметров фичей n и t . За n всегда будем обозначать количество сделок в окне. Функция заполняет словарь значениями фичей, которые представлены ниже:

1. `seconds_since_midnight` – количество секунд с начала дня.
2. `seconds_since_n_trades_ago` – количество секунд, прошедших с первого трэйда в окне из n последних трэйдов.

Псевдо-формула: $(window.end_time - window[-n].time).to_seconds()$

3. `WI_exp_moving_average` – экспоненциальное среднее в окне длины t .

Псевдо-формула: $\alpha (window[-1].price + (1-\alpha) window[-2].price + \dots + (1-\alpha)^{n-1} window[-n].price)$

4. `WI_weighted_moving_average` – взвешанное среднее в окне длины t .

Псевдо-формула:
$$\frac{\sum_{i=0}^n window[i].price \cdot window[i].volume}{\sum_{i=0}^n window[i].size}$$

5. WI_trade_amount – количество сделок в окне длины t .

Псевдо-формула: n

6. WI_trade_volume – суммарный объем сделок в окне длины t .

Псевдо-формула: $\sum_{i=0}^n \text{window}[i].volume$

7. WI_open_close_diff – частное между ценой закрытия и ценой открытия в окне длины t .

Псевдо-формула: $\text{window}[-1].price / \text{window}[0].price$

8. WI_stochastic_oscillator – стохастический осцилятор.

Псевдо-формула: $\frac{\text{CUR}-\text{MIN}}{\text{MAX}-\text{MIN}}$, где CUR - цена последней сделки, а MIN и MAX - наименьшая и наибольшая цена сделки соответственно в окне длины t .

Вторая фича зависит от значения n и от направления трэйдов в окне, поэтому для каждой комбинации направления и параметра n в таблице будет свой столбец.

Последние 5 фичей зависят от t и от направления трэйдов в окне, поэтому для каждой комбинации направления, параметра t и фичи в таблице будет свой столбец.

В последствии мы решили еще дополнительно делить все скользящие средние на среднее арифметическое цен всех сделок в окне.

Псевдо-формула: $\text{WI_moving_average} \cdot \frac{n}{\sum_{i=0}^{n-1} \text{window}[i].price}$

Это сделано для того, чтобы при глобальном изменении стоимости валюты модель продолжала работать.

5 Задача классификации

5.1 CatBoost

Самый простой среди мощных и самый мощный среди простых – CatBoost [?]. Эта библиотека, которая решает задачу классификации. Она основана на градиентном бустинге, надстройкой над решающими деревьями. Как оно работает внутри – тема другой работы, здесь мы рассмотрим ее применение.

5.2 Данные

5.2.1 Датасет

По индикатором, которые мы выбрали, нужно построить датасет. Мы его устроили так:

1. Посчитали значения индикаторов по каждой из сторон за последние n/c , $2n/c$, ..., n секунд.
2. Распарсили так, что по столбцам данные за промежутки $[0, n/c]$, $[n/c, 2n/c]$, ..., $[(c - 1)n/c, n]$ секунд.
3. Параметры выбрали такие: $n = 60$, $c = 10$.
4. В качестве вектора ответов у нас единица, если рынок отклонился более, чем на Δ вверх, -1, если вниз, и 0 иначе. Δ экспериментально выбрали как $5 * \text{commision}$, $\text{commision} = 0.02\%$.

5.2.2 Парсинг

Код:

1. Парсер.
2. Очередь на минимумы/максимумы.
3. Индикаторы.

Парсер устроен так:

1. У нас есть две очереди на минимумы/максимумы. Мы написали свою реализацию BuySellQueue. Часть операций, которые поддерживает эта очередь:
 - (a) pop_front
 - (b) push_back
 - (c) get_side_queue_max_price
 - (d) get_side_queue_min_price
2. В каждый момент времени в первой очереди у нас все трейды за последний 60 секунд, а во второй за последующие 30. Эти параметры можно настраивать.
3. После чтения очередного трейда мы обновляем очереди и с какой-то вероятностью пересчитываем индикаторы. Делаем это не всегда, потому что количество трейдов огромное, и построить датасет по всем сделкам невозможно из-за ограничений по памяти.

5.3 Использование CatBoost

5.3.1 Fit-Predict

```
params = {
    'iterations': 300,
    'l2_leaf_reg': int(best['l2_leaf_reg']),
    'learning_rate': best['learning_rate'],
    'custom_loss': [metrics.Accuracy()],
    'eval_metric': metrics.Accuracy(),
    'random_seed': 42,
    'logging_level': 'Silent',
    'loss_function': 'Logloss',
}
train_pool = Pool(X_train, y_train)
validate_pool = Pool(X_validation, y_validation)
model = CatBoostClassifier(**params)
model.fit(
    train_pool,
    eval_set=validate_pool,
    plot=True,
)
predictions = model.predict(X_test)
predictions = predictions.reshape(predictions.shape[0], 1)
predictions_probs = model.predict_proba(X_test)
```

5.3.2 Автоподбор гипер-параметров

```
def hyperopt_objective(params):
    model = CatBoostClassifier(
        l2_leaf_reg=int(params['l2_leaf_reg']),
        learning_rate=params['learning_rate'],
        iterations=300,
        eval_metric=metrics.Accuracy(),
        random_seed=126,
        verbose=False,
        loss_function=metrics.Logloss(),
    )
    cv_data = cv(
        Pool(X, y),
        model.get_params(),
        fold_count=5,
```

```

        shuffle=True,
        logging_level='Silent',
    )
best_accuracy = np.max(cv_data['test-Accuracy-mean'])
return 1 - best_accuracy

params_space = {
    'l2_leaf_reg': hyperopt.hp.qloguniform('l2_leaf_reg', 0, 2, 1),
    'learning_rate': hyperopt.hp.uniform('learning_rate', 1e-3, 5e-1),
}
trials = hyperopt.Trials()
best = hyperopt.fmin(
    hyperopt_objective,
    space=params_space,
    algo=hyperopt.tpe.suggest,
    max_evals=50,
    trials=trials,
    rstate=RandomState(42)
)

```

5.4 FPR/FNR

Определение 9. **False Positive/Negative Ratio** – отношения ложно-положительных/ложно-отрицательных предсказаний ко всем отрицательным/положительным.

Обозначение 10. FPR/FNR

Мы хотим оптимизировать лучше FPR, чем FNR, потому что в первом случае мы теряем деньги, а во втором не получаем.

Параметр Δ подобран так, что

$$FPR = \frac{\text{false positive}}{\text{total negative num}} = 0.010005175090564086$$

$$FnR = \frac{\text{false negative}}{\text{total positive num}} = 0.8827444956477215$$

Эксперимент показал, что такие значения достигаются при плече принятия решений $\omega = 34\%$. То есть если модель предсказывает ± 1 с такой вероятностью или больше, то мы считаем, что ее прогноз верный. Причем мы не хотим оказаться в ситуации, когда модель выдает $P(1) = 45\%$, $P(-1) = 45.1\%$, порог в 34% пройден, значит надо выбирать -1. Но мы хотим выбрать -1 только тогда, когда $P(-1) \geq \omega \cap P(-1) > (P(1) + \varepsilon) \cap P(-1) > (P(0) + \varepsilon)$. Эксперимент выдал, что $\varepsilon = 1\%$ оптимальнее всего.

5.5 Важность индикаторов

После того, как мы обучили модель, можно посмотреть самые важные индикаторы, которые выбрал CatBoost. Сделать это можно командой

```

feature_importances = model.get_feature_importance(train_pool)
feature_names = X_train.columns
for score, name in sorted(zip(feature_importances, feature_names), reverse=True):
    print('{}: {}'.format(name, score))

```

Топ-15 индикаторов оказались:

1. open-close-diff-SELL-60-sec
2. stochastic-oscillator-BUY-600-sec
3. open-close-diff-SELL-600-sec

4. open-close-diff-BUY-60-sec
5. stochastic-oscillator-SELL-600-sec
6. open-close-diff-BUY-600-sec
7. open-close-diff-BUY-30-sec
8. open-close-diff-SELL-30-sec
9. seconds-since-1000-trades-ago-BUY
10. seconds-since-midnight
11. seconds-since-10-trades-ago-BUY
12. seconds-since-100-trades-ago-BUY
13. seconds-since-50-trades-ago-BUY
14. trade-amount-BUY-60-sec
15. stochastic-oscillator-BUY-60-sec

5.6 Распределение индикаторов

Чтобы понять, насколько хорошо модель может обучиться, надо посмотреть на распределение индикаторов по разным таргетам. Хочется, чтобы как можно большая часть распределение разных таргетов не пересекалась, потому что тогда алгоритм решающих деревьев будет справляться лучше всего.

Посмотрим на распределение всех фичей, и поближе на seconds-since-50-trades-ago-BUY и open-close-diff-SELL-30-sec

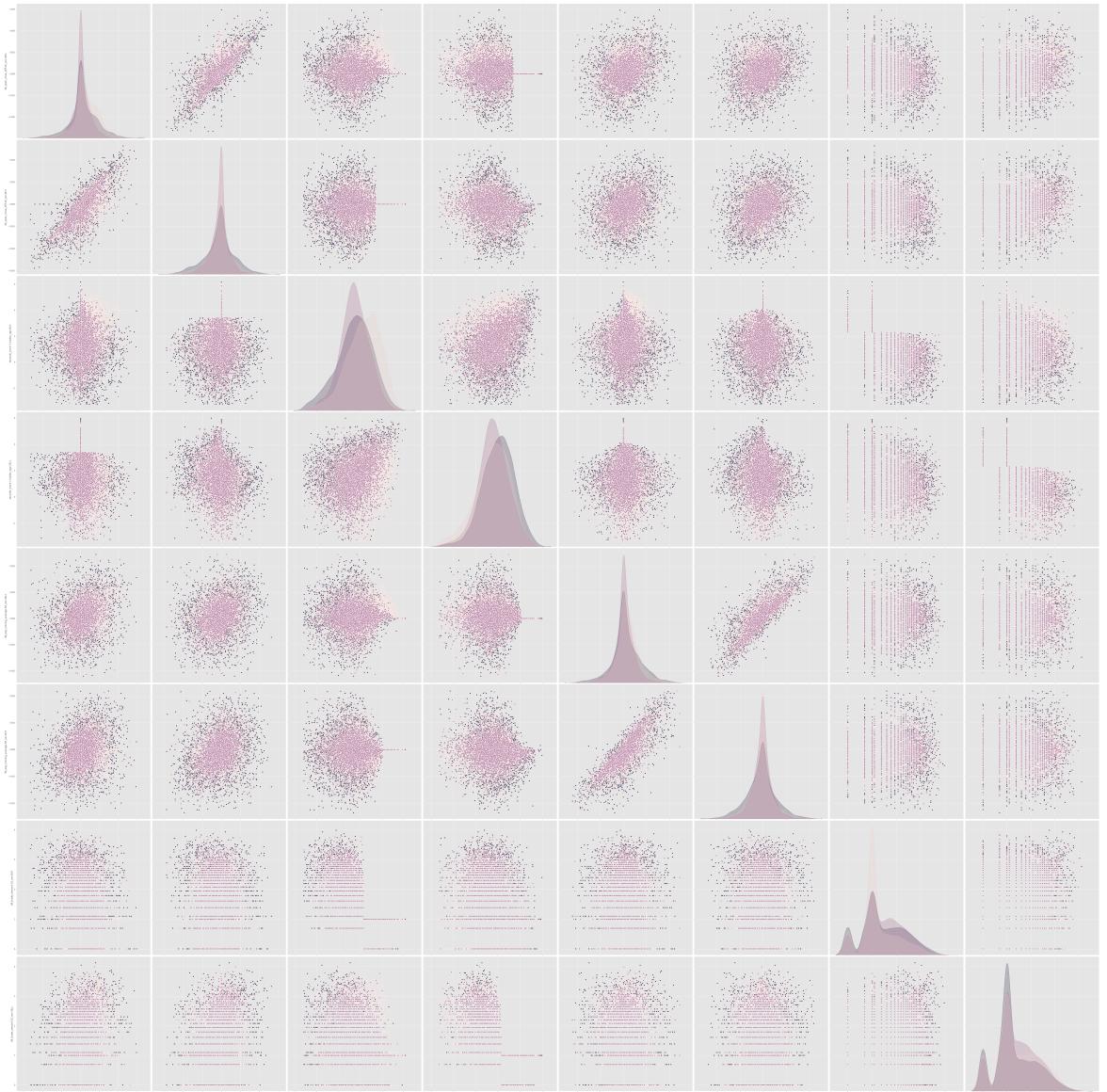


Рис. 1: Распределение фичей

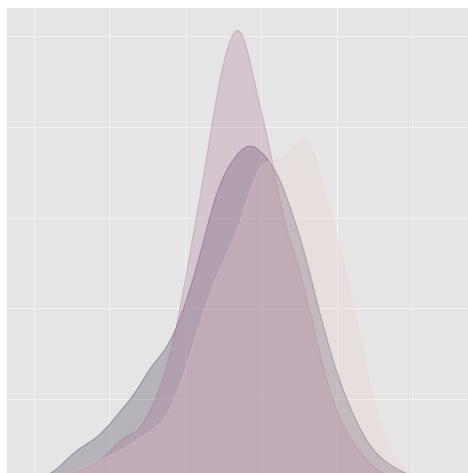


Рис. 2: seconds-since-50-trades-ago-BUY

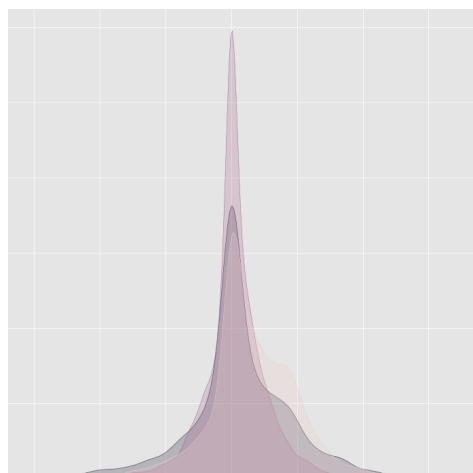


Рис. 3: open-close-diff-SELL-30-sec

Видим, что оба распределения такие, что почти невозможно отличить один таргет от другого, только на окно немножко отличить таргет -1 , который бежевого цвета. Поэтому ждать, что модель у нас получится хорошая не приходится.

5.7 Результат

При таких параметрах получается, что моделька в 9.78% случаев с вероятностью 69.63% правильно предсказывает скачок рынка. То есть из всех скачков она ловит только 9.78%. И среди них с вероятностью 69.63% угадывает, куда пойдет рынок.

Однако если учитывать комиссию биржи, то мы будем играть в минус.

6 Арбитражная стратегия

В нашем проекте мы, конечно, не могли не попробовать одну из самых распространённых и классических стратегий высокого частотного трейдинга – арбитраж. Идея в этой стратегии все просто: мы смотрим на цену на одной бирже (сигнальной) и если она резко пошла вверх или вниз, то предпринимаем соответствующие действия на нашей бирже.

6.1 Описание стратегии

Если мы засекли быстрое и сильное изменение цены на сигнальной бирже, то выставляем рыночный ордер на нашей бирже в направлении изменения цены. Следом отправляем два ордера, которые будут фиксировать прибыль и убыток. Таким образом, если мы угадали с направлением движения цены, то у нас появляется шанс заработать. Ключевые параметры стратегии:

- **Profit threshold** – от этого параметра зависит, насколько далеко от цены исполнения рыночного ордера мы будем выставлять ордер для фиксации прибыли.
- **Loss threshold** – аналогично предыдущему, но для фиксации убытков.
- **Signal threshold** – показывает, какое изменение в цене мы считаем "сильным", если на сигнальной бирже оно было больше этого параметра, то можем фиксировать сигнал
- **Window size** – промежуток времени, за который мы фиксируем скачок. Если цена выросла на нужный **Signal threshold** за **Window size**, то тогда фиксируем сигнал, если цена росла дольше, чем **Window size**, то мы не считаем это за сигнал.
- **Sec to wait** – сколько мы держим рыночный ордер. Если по фиксирующим ордерам не прошла сделка и через **Sec to wait** секунд мы не зафиксировали ни прибыль, ни убыток, то закрываем нашу позицию новым рыночным ордером.

6.2 Выбор биржи и инструмента

В мире сейчас очень много криптовалютных бирж, но самой крупной уже долгие годы остается Binance, поэтому мы решили наблюдать за ценой на ней, ведь там совершаются десятки сделок в секунду. Инструментов тоже существует бесчисленное множество, мы остановились на бессрочном фьючерсе биткоина, который торгуется в долларах. Такой выбор был продиктован следующими причинами: биткоин до сих пор остается самой популярной криптовалютой, цена на сам биткоин очень волатильна, а на его фьючерсы и подавно, это дает нам возможность ловить большие скачки цены, на бирже dydx в основном представлены бессрочные фьючерсы и нам показалось логичным совершать сделки на основании информации о похожем инструменте.

6.3 Обнаружение скачков с помощью скользящего окна

Одна из ключевых задач в арбитражной стратегии – обнаружение быстрого и сильного изменения цены. Мы считаем, что цена изменилась быстро, если изменения произошли менее, чем за одну секунду (значит, **Window size** = 1000 миллисекунд) и сильно, если скачок цены был больше, чем на две комиссии на Binance (значит, **Signal threshold** = $1 + 2 * \text{Binance Commission} + \text{eps}$). Чтобы обнаруживать такие события, был создан класс **SlidingWindow** [?], реализующий алгоритм поиска минимума в скользящем окне [?], адаптированного под наши нужды. Работает это примерно следующим образом:

- Мы создаем экземпляр класса SlidingWindow, с указанием размера окна в миллисекундах

```
slide = SlidingWindow(window_size)
```

- При поступлении нового трейда от вебсокета Binance/при анализе исторических данных, мы парсим данные и обновляем окно, добавляя туда новую цену и timestamp свежего трейда, метод push_back возвращает True, если минимум или максимум был изменен и False иначе:

```
result = slide.push_back(price, timestamp)
```

- Далее мы смотрим, если отношение минимума цены к максимуму больше или равно нашего порога, то тогда считаем, что мы поймали скачок, который дальше можно как-то анализировать:

```
if result:
    max_in_window = self.slide.get_max()
    max_timestamp = self.slide.get_max_timestamp()
    min_in_window = self.slide.get_min()
    min_timestamp = self.slide.get_min_timestamp()
    if max_in_window / min_in_window >= (
        1 + self.signal_threshold):
        if max_timestamp > min_timestamp:
            return "BUY"
        elif max_timestamp < min_timestamp:
            return "SELL"
```

Этот класс будет использоваться для поиска скачков на исторических данных и для их обнаружения в реальном времени в рабочем прототипе трейдера

6.4 Анализ исторических данных, реагируем на все сигналы

Чтобы понять, возможно ли вообще заработать, используя эту стратегию, был проведен анализ исторических данных. Данные с Binance мы брали из публичного архива [?], а данные с dydx собирали простеньким скриптом [?], впрочем, и в коннекторе есть для этого отдельная функция [?].

6.4.1 Class ProfitCalculator

Для анализа исторических данных и подсчета прибыли был создан класс с говорящим названием ProfitCalculator, который позволяет нам понять, сколько мы теоретически можем заработать, используя арбитражную стратегию. При создании экземпляра этого класса ему обязательно передаются имена файлов, которые содержат исторические трейды с сигнальной биржи и трейды биржи, с которой мы торгуем. Можно также установить ключевые параметры стратегии, но они есть по умолчанию. Выглядит это так:

```
PC = ProfitCalculator(signal_filename, predict_filename, **params)
```

6.4.2 Сбор сигналов

У класса ProfitCalculator есть метод get_signals, который позволяет нам получить сигналы на основе трейдов, которые лежали в файле с именем signal_filename. Мы можем сдампить сигналы, воспользовавшись соответствующими методами класса. И, конечно, визуализировать их тоже можно. То есть стандартный сценарий использования ProfitCalculator с целью получения сигналов такой:

```
PC.get_signals()
PC.dump_signals(signals_filename)
PC.show_signals()
```

Вот пример работы метода show_signals:

Signals, side = BUY



Рис. 1: BUY signal example

6.4.3 Получение трейдов и расчет прибыли

После того как мы получили сигналы, можно воспользоваться методом `get_trades`, который, имитируя торговлю по арбитражной стратегии на исторических данных из файла `predict_filename`, сохраняет трейды. Далее можно сдампить трейды в файл с помощью метода `dump_trades` и получить график, на котором отмечены наши сделки помошью `show_trades`.

Signal trades | Amount of trades : 42 | Total profit : -2.052829

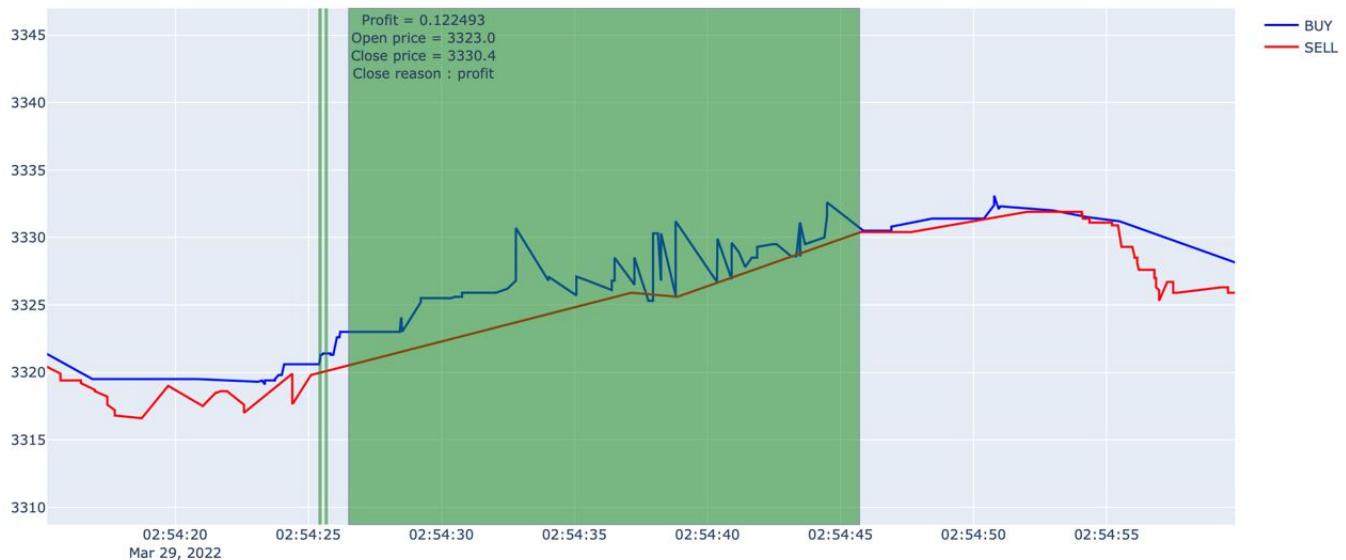


Рис. 2: BUY trade example

Теперь, когда у нас есть трейды, мы можем посчитать, на сколько процентов увеличится наш портфель. Делается это с помощью метода `get_profit`, который возвращает нам общий профит со всех циклов откры-

тия и закрытия позиций и список профитов для каждого из них по отдельности. Вот таблица суммарных профитов за несколько месяцев:

Dec	Jan	Feb	Mar
5.21	6.45	-6.27	-17.03

Эти значения были получены при фиксированных ключевых параметрах на весь месяц. Понятно, что можно с ними немного поиграться с целью увеличения профита, но это не дает больших результатов. Сколько бы мы не экспериментировали с ними, картина остается примерно такой же: в декабре/январе мы что-то зарабатываем, а дальше все очень плохо.

6.5 Анализ исторических данных, попытка отсеять плохие сигналы

Как видно из таблицы выше, ситуация скорее печальная, чем радостная. Есть несколько аномальных месяцев, которые принесли бы нам прибыль, но большинство из них все-таки были убыточными. Мы теряем много денег, когда реагируем на плохие сигналы и зарабатываем недостаточно, когда реагируем на хорошие, но выставляем плохие трешхолды. Есть два очевидных пути улучшения нашей стратегии: научиться не реагировать на плохие сигналы, либо уметь подбирать ключевые параметры стратегии для каждого сигнала по отдельности. Поговорим здесь о первом варианте (спойлер: даже он не работает).

6.5.1 Сбор фичей

Нам нужно выделить какие-то фичи, по которым мы будем предсказывать, будет ли сигнал прибыльным или нет. Их должно быть удобно собирать, они должны отражать фундаментальные характеристики нашего скачка. Мы выделили следующие признаки:

- `Average quantity` – средняя цена объема сделки в окне.
- `Max quantity` – максимальная цена объема сделки в окне.
- `Stdev price` – стандартное отклонение цены в окне.
- `Stdev quantity` – стандартное отклонение объема в окне.
- `Trades frequency` – частота трейдов в окне.
- `Jump value` – величина прыжка: во сколько раз изменилась цена.
- `Jump time length` – длина прыжка: сколько миллисекунд потребовалось, чтобы достичь `Jump value`.

Все эти признаки легко собирать с помощью стандартного питоновского пакета `statistics`, добавив еще одно скользящее окно, отслеживающее объем трейдов, вместо их цены. Теперь метод `get_signals` будет не только находить сигналы, но и собирать фичи к каждому из них.

6.5.2 Узнаем, был ли сигнал теоретически прибыльным

Для этого мы будем идти на 20 секунд вперед от времени сделки по рыночному ордеру, который мы отправили после получения сигнала, и если отношение цены хотя бы одного из трейдов за эти 20 секунд к цене рыночного была $> 1 + \text{dydx_commission}$, то это значит, что мы могли заработать, отреагировав на этот сигнал.

Еще по дороге будем собирать наилучшие ключевые параметры стратегии для данного сигнала: какой максимальный `Profit threshold` мы могли выставить, чтобы заработать как можно больше, какой должен был быть `Profit threshold`, чтобы не закрыться по нему, зафиксировав прибыль и т.п. Делается это просто, надо всего лишь найти минимальную/максимальную цену на окне в 20 секунд, которые мы анализируем и посчитать отношение цен.

Average quantity	Max quantity	...	Has profit
100	1000	...	1
...	0
...

6.5.3 Пытаемся отличить плохие сигналы от хороших

Теперь у нас есть матрица вида

И мы можем попробовать поприменять к этим данным различные модели классификации, чтобы научиться отличать успешные сигналы. Но для начала посмотрим на совместное распределение фичей. Может удастся увидеть какую-то структуру в данных.

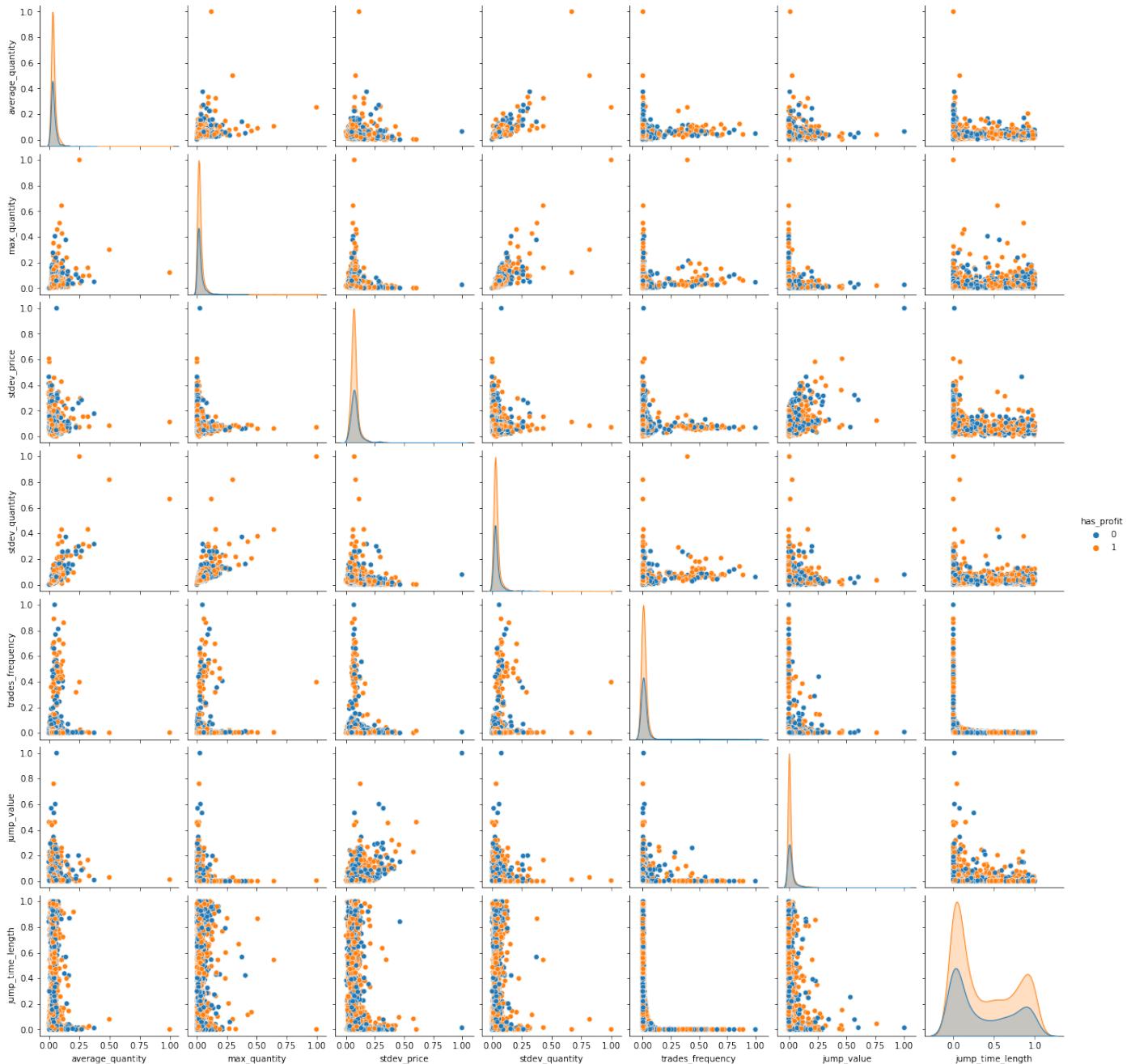


Рис. 3: Feature's distribution

Видно, что классы очень плохо отделяются, все перемешано, шансов на хорошую классификацию очень мало. Но мы попробуем (но ничего не получиться).

- **Решающие деревья**

Пользуемся `DecisionTreeClassifier` из `sklearn`. Для хорошей интерпретируемости ограничим глубину дерева одним уровнем. Это даст нам понимание, какой самый значимый признак выделил классификатор.

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=1)
model.fit(X_train, Y_train_has_profit)
model.score(X_test, Y_test_has_profit)
```

Результаты не очень хорошие: средняя точность составляет всего 69%. Посмотрим на визуализацию нашего небольшого дерева:

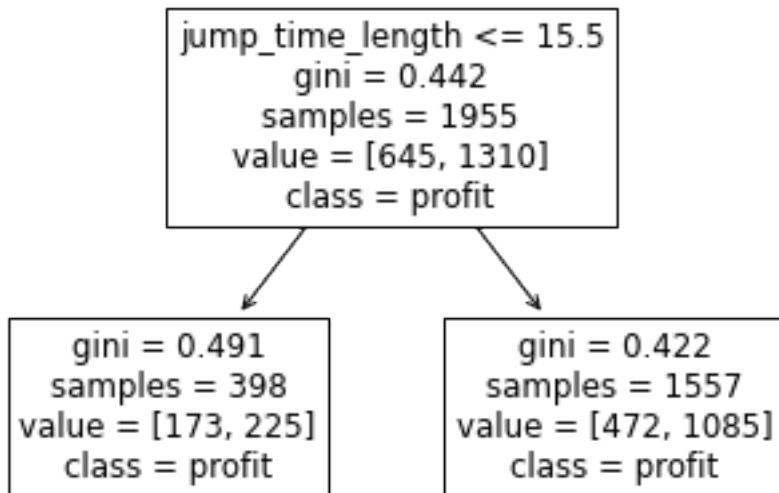


Рис. 4: Decision Tree

Видно, что gini-коэффициент в каждой вершине довольно близок к 0.5, это означает, что данный признак плохо разделяет классы. В каждой вершине находится много сигналов обоих типов. Общее число успешных сигналов во всем нашем датасете составляет где-то 66%. Если мы будем игнорировать все сигналы, `Jump time length` которых меньше 100мс, то можем повысить долю успешных сигналов до 68%:

```
>>> print(f"Было: {df_total[df_total.has_profit == 1].shape[0] / df_total.shape[0]}")
Было: 0.6617752326413744
>>> df_total = df_total[df_total.jump_time_length > 100]
>>> print(f"Стало: {df_total[df_total.has_profit == 1].shape[0] / df_total.shape[0]}")
Стало: 0.6815804764671702
```

Можно еще раз повторить процедуру с `DecisionTreeClassifier`, но уже с отфильтрованным датасетом, получим такую картинку:

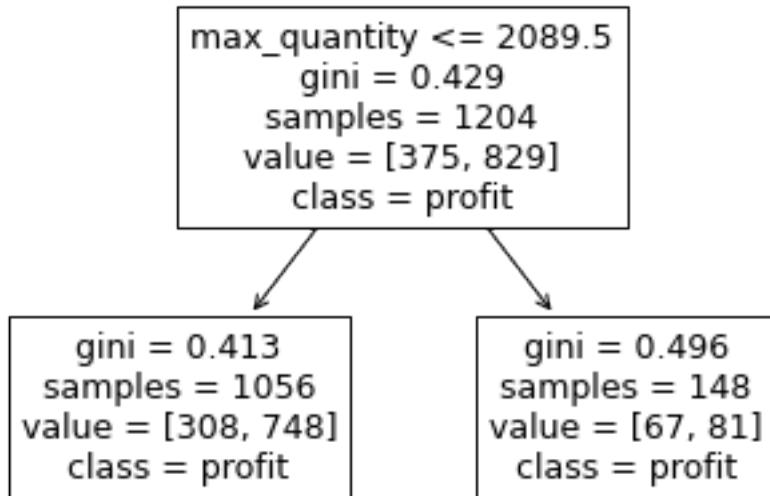


Рис. 5: Decision Tree

Ну тоже, мягко говоря, не очень получается, но давайте опять отфильтруем датасет по этому признаку и посмотрим, возросла ли доля хороших сигналов:

```

>>> print(f"Было: {df_total[df_total.has_profit == 1].shape[0]
        / df_total.shape[0]}")
Было: 0.6815804764671702
>>> df_total = df_total[df_total.max_quantity <= 2100]
>>> print(f"Стало: {df_total[df_total.has_profit == 1].shape[0]
        / df_total.shape[0]}")
Стало: 0.6950732356857523

```

Ну опять как-то не радостно. Можем теперь посчитать профит на разных месяцах, отсекая сигналы, у которых `Jump time length < 100` и `Max quantity > 2100`. Получим следующие профиты:

Dec	Jan	Feb	Mar
5.21	6.45	-6.27	-17.03

Ситуация практически не изменилась, а мы ведь так старались!

- **Другие модели**

Мы также попробовали другие модели бинарной классификации: логистическая регрессия, решающие деревья с градиентным бустингом, даже какие-то совсем базовые нейросети. Но ничего не дает впечатляющего результата, который бы позволял нам зарабатывать больше.

Что касается попытки предсказывать ключевые параметры стратегии, то она была обречена на провал, ведь если задуматься, одной из характеристик модели, которая бы предсказывала параметры, как раз является умение отличить плохой сигнал от хорошего, а мы это сами не смогли сделать. Попробовали разные виды линейных регрессий с регуляризацией, различные регрессоры, основанные на решающих деревьях, но все это не принесло никаких хороших результатов. Очень жаль!

7 Трейдер

[Трейдер в репозитории](#)

Определение 11. Трейдер – программа, которая на основе готовой стратегии, совершает сделки.

7.1 Использованные технологии

7.1.1 Асинхронная функция

Определение 12. Асинхронная функция – функция, которая не блокирует основной поток программы.

Наш проект написан на питоне, а он одно-поточный. Поэтому нужно было пользоваться всеми возможностями асинхронности. Для этого в питоне есть очень простой интерфейс:

```
async def slow_calculation():
    return ...

async def hard_calculation():
    return ...

def main():
    loop = asyncio.get_event_loop()
    loop.create_task(
        slow_calculation(), name="slow_calculation"
    )
    loop.create_task(
        hard_calculation(), name="hard_calculation"
    )
    loop.run_forever()
```

Здесь мы создали две задачи: `slow_calculation` и `hard_calculation`, которые будут выполняться асинхронно, то есть не мешать друг другу своим вычислениями.

7.1.2 `asyncio.Event`

В трейдере, помимо асинхронных функций, используется класс `asyncio.Event`, позволяющий ждать задачам, пока событие не произойдет в другой задаче и элемент класса не будет разблокирован.

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

В данном примере функция `waiter` не завершится до тех пор, пока функция `main` не разблокирует событие строчкой `event.set()`, потому что `waiter` находится в ожидании после строчки `await event.wait()`.

С помощью этого класса можно синхронизировать работу алгоритма торговли с ответами от биржи.

7.2 Устройство трейдера

Используемые параметры:

- `trailing_percent` – процент, на который цена stop-loss ордера отстает от рыночной при отправке trailing-stop ордера.
- `profit_threshold` – процент, на который цена лимитки, выставленной по take-profit ордеру отличается от рыночной.
- `sec_to_wait` – количество секунд, через которое цикл торговли считается завершенным, если ни take-profit, ни trailing-stop ордера не были исполнены.

Трейдер состоит из двух асинхронных задач и вспомогательных функций.

Из ключевых вспомогательных функций нужно отметить функцию `reset`, которая переводит все поля класса в состояние для ожидания скачка на бинансе. Она вызывается после успешного завершения цикла торговли, или после отмены ордеров и закрытия позиции в случае, когда какой либо ордер не смог исполниться.

Основные асинхронные задачи:

- `listen_binance`
- `account_listener`

Пройдемся по каждой

7.2.1 `listen_binance`

[Функция в репозитории](#)

Определение 13. Слушать [что-то] – значит подключиться по веб-сокету к какому-то серверу и получать от него обновления.

Эта функция реализует логику торговли. В этом трейдере по техническим причинам не получилось использовать функционал коннектора для бинанса, поэтому подписываться на обновления пришлось вручную. Функция держит окно трейдов на одном направлении фиксированной длины, и если перепад цены превысил порог, то запускается алгоритм торговли:

1. Отправляется маркет ордер. Функция входит в режим ожидания с помощью `asyncio.Event` и ждет, пока с биржи придет ответ об исполнении ордера. Если с биржи приходит сообщение об отмене ордера, то вызывается функция `reset` и трейдер возвращается в состояние ожидания скачка цены на бинансе.
2. Когда `account_listener` сообщил об успешном выполнении маркет ордера и сохранил цену по которой он исполнился, трейдер выставляет `take-profit` и `trailing-stop` ордера с заданными при инициализации класса параметрами и на протяжении `sec_to_wait` секунд ожидает исполнения хотя бы одного из них.
 - Если за `sec_to_wait` секунд ордера не закрылись, то они отменяются. Далее закрывается позиция в которую мы зашли маркетом, и когда с биржи приходит сообщение о закрытии позиции, цикл торговли считается завершенным.
 - Если какой-то из ордеров исполнился, то мы в любом случае отменяем оставшийся. Если исполнился лимит, то мы в плюсе, если трейлинг стоп, то, скорее всего, в минусе. После отмены цикл считается завершенным.

7.2.2 `account_listener`

[Функция в репозитории](#)

Эта задача слушает изменения в наших ордерах и позициях, обновляет цену, по которой исполнился маркет, выставляет флаги и разблокирует нужные события, на которых спит функция `listen_binance`, тем самым обеспечивая её синхронизацию с биржей.

8 Дополнительные результаты

Помимо задач, которые нам поставил руководитель, мы сделали еще:

8.1 Визуализация

Перед тем, как работать с данными, надо понять, как они устроены: попарное распределение классов, плотность каждого из индикаторов. Мы все это сделали. Теперь стало гораздо проще подбирать параметры для модели машинного обучения.

8.2 Контролирующий бот

Мы сделали телеграм бота, на которого можно по паролю подписаться и получать обновления состояния аккаунта на бирже. Это полезно, когда ты запускаешь торговую стратегию, куда-то отходишь, но при этом всегда можешь контролировать, что с ней происходит, через телеграм.

8.3 CI/CD, тесты, линтер

Любая ошибка в торговой системе может потерять наши деньги, поэтому важно, чтобы весь код всегда был рабочим. Для этого мы все, что смогли, обложили тестами с использованием утилиты PyTest [?]. Теперь при каждом pull реквесте у нас запускается тестирующая система, и если какие-то тесты не проходят, то мы запрещаем дальнейший пуш. Реализовали мы это через GitHub Actions [?].

Еще нам хочется, чтобы весь код был консистентным. Для этого мы используем линтер Black [?] и статический анализатор PyLint [?]. В совокупности эти утилиты поддерживают консистентность нашего кода и могут еще до запуска теста, выдать синтаксические ошибки.

8.4 Смарт контракты

С помощью смарт контрактов можно занимать у других людей миллиарды, трейдить на них, и потом возвращать криптовалюту обратно. Звучит заманчиво. Мы написали смарт контракты на Solidity [?], и залили их в сеть эфира через Brownie [?].