Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему **«Forex торговий бот мовою Rust»**

Виконав: студент 3-го року навчання,

Спеціальності
121 «Інженерія Програмного Забезпечення»

Студента Маслова Нікіти

Керівник Бабич Т.А.

магістр комп’ютерних наук, асистент

«6» червня 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2021 року

Завдання для курсової студенту

- Тема роботи — «Forex торговий бот мовою Rust», керівник роботи Бабич Трохим Анатолійович, магістр комп’ютерних наук, асистент
- Строк подання студентом роботи — 3 червня 2022
- План роботи
    - Анотація
    - Вступ
    - Розділ 1. Дослідження та аналіз предметної області
        - Валютні пари, графіки, свічки, таймфрейми та тіки
        - Торговельна платформа та API для автоматизованої торгівлі
        - Торговий брокер
        - Тестування на історії, демо рахунок та справжня торгівля
        - Оптимізація стратегії
    - Розділ 2. Проектування та розробка системи
        - Складові системи та використані технології
        - Ядро стратегії
        - Виконавець стратегії
        - API ринкових даних
        - Синхронізація ринкових даних

        - Кешування ринкових даних
        - Відображення результатів тестування на історії
        - Оптимізація стратегії
        - Розміщення на сервері та планувальник задач
- Висновки
- Список використаних джерел

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2021			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтеня 2021 – 2 листопада 2021			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2021			
4.	Написання розділів роботи	2 листопада 2021 – 01 березня 2022			

5.	Проміжний контроль виконання роботи	01 лютого 2022			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2022 – 29 березня 2022			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2022			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2022			
	Розділ 3 (проектно-рекомендаційна частина)	29 березня 2022			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	01 квітня 2022 – 06 травня 2022			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	17 травня 2022			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2022 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець курсової роботи Маслов Нікіта

ЗМІСТ

Анотація

У цій роботі розглянеться, як створити повноцінну автоматизовану систему торгівлі на Forex: як облаштувати універсальну інфраструкту для тестування стратегій на історичних даних, як оптимізувати стратегію для найкращих фінансових результатів, як запрограмувати виконавця стратегій для торгівлі в реальному часі, як краще налаштувати торгову систему на сервері.

Вступ

Все почалося с того, що з’явилася інсайдерська, дуже ефективна, але складна торгова стратегія. Приблизні фінансові результати за рік, які отримують мої знайомі трейдери з цією стратегією — це 200—300% від балансу торгового облікового запису на початку року. Але торгувати цю стратегію вручну дуже важко, бо треба постійно відслідковувати дуже багато параметрів та робити нетривіальні розрахунки. Тому було вирішено створити автоматизовану систему торгівлі, яка буде все робити самостійно, постійно себе підлаштовувати під волатильний ринок та доповідати про свої кроки та результати.

Можна було б всю систему прив’язати до однієї конкретної стратегії, але було вирішено додатково створити універсальну екосистему для автоматизованої торгівлі, яку можна використовувати з будь-якими іншими стратегіями.

Розділ 1. Дослідження та аналіз предметної області

Валютні пари, графіки, свічки, таймфрейми та тіки

Щоб було легше розуміти торгівлю на Forex, важливо спочатку познайомитися з базовими сутностями, які повинен знати кожен трейдер.

Валютні пари

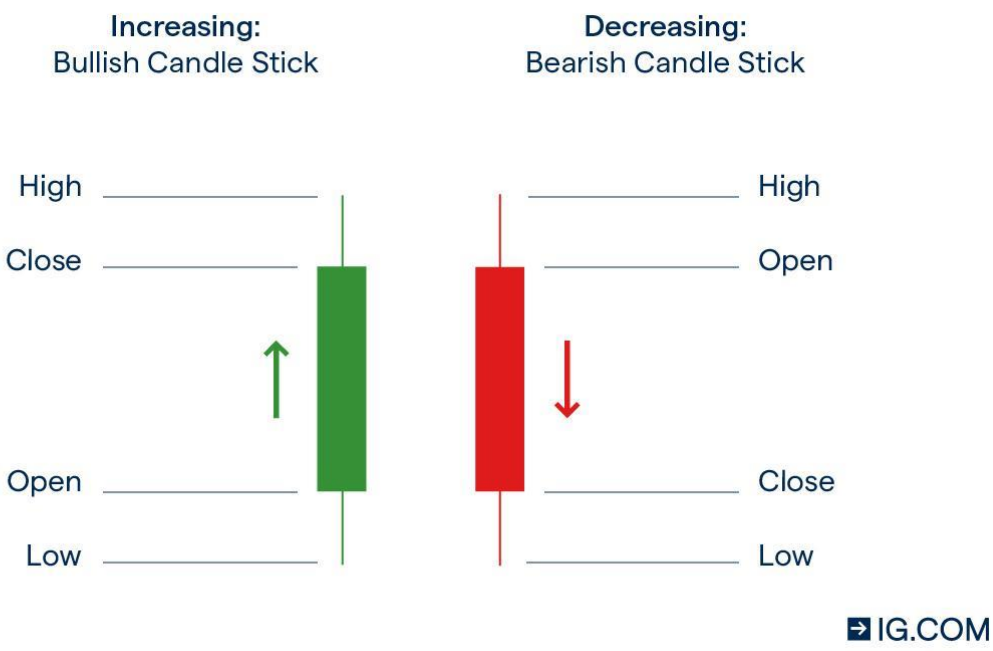
Валютні пари на Forex — це співвідношення двох національних валют на ринку Forex. Будь-яка валютна пара складається з базової валюти,

що стоїть на першому місці, і валюти котирування. Базова валюта — це валюта, яка продається або купується за другу валюту в парі. Валютні пари позначаються назвами валют, які в них входять.

Наприклад, у валютній парі EUR/USD європейська валюта виступає товаром, а американський долар — грошима. Значення валютної пари на даний момент часу вказує, скільки треба віддати доларів США за один євро. Так само розглядаються й всі інші пари.

Графіки

Основний об’єкт, з яким постійно контактують трейдери під час торгівлі — це графік змін цін валютних пар. Один з видів графіків — це графік, який складається з послідовного ланцюжка свічок певного таймфрейму. Цей графік ще можна коротко назвати графіком японських свічок.



Свічки

Свічка (або японська свічка) — це візуальний технічний індикатор, який складається з тіла та паличок, які ще називають тінями. Тіло — це прямокутник, який відображає різницю між ціною в момент відкриття свічки та ціною в момент її закриття.

Якщо свічка має **зелений колір**, це означає, що ціна відкриття (open) знаходиться на лінії нижньої сторони прямокутника, а ціна закриття (close) — на лінії верхньої сторони сторони. Тобто за час формування свічки ціна виросла. Зелені свічки ще називають бичачими.

Якщо свічка має **червоний колір**, це навпаки означає, що ціна відкриття знаходиться на лінії верхньої сторони прямокутника, а ціна закриття — на лінії нижньої сторони прямокутника. Червона свічка вказує на те, що за час її формування ціна впала. Червоні свічки ще називають ведмежими.

Палички, які йдуть вгору та вниз від тіла свічки, називаються тінями свічки. Ці тіні відображають максимальну (high) та мінімальну (low) ціни, які з'являлися за весь період формування свічки. Верхній край верхньої палички — це максимальна ціна, нижній край нижньої палички — мінімальна ціна.

Час, за який формується одна свічка, називається **таймфреймом**. Є багато різних таймфреймів: 1 год., 30 хв., 15 хв., 5 хв., 1 хв. і т.д.

Трейдери можуть працювати з різними таймфреймами в залежності від своєї стратегії.

Тіки

Окрім свічок, є ще тіки. Тіки відображають постійну мінімальну зміну ціни на ринку. Для трейдера тік завжди виглядає як дві ціни: **ask** і **bid**. Ask — це ціна, за якою трейдер купує у продавця, тому вона вище. А bid — це ціна, за якою трейдер продає, тобто та сума, яку запитує покупець, і вона завжди нижча.

Різниця в ціні між ask та bid називається **spread**. Для трейдера spread — це ті торгові витрати, які він сплачує при кожному закритті угоди. Тобто якщо трейдер купляє якусь кількість валюти та одразу її продає, то це не нейтральна угода для нього. Трейдер у цьому випадку втрачає невеликі кошти, а саме значення spread, різниці між ціною купівлі та продажу.

На графіку торгової платформи можна слідкувати за постійно змінюваними тіками у вигляді різких невеликих стрибків ціни вгору чи вниз.

Торговельна платформа та API для автоматизованої торгівлі

Найпопулярніша торговельна платформа — **Metatrader.** Для написання торгових скриптів у них є своя мова програмування MQL5, яка схожа на C++. Її часто використовують для написання невеликих скриптів-помічників, які вбудовуються в саму програму Metatrader та допомагають ручній торгівлі. Але поставлена задача — реалізувати складну повністю автоматизовану систему, тому MQL5 буде накладати більше обмежень, ніж допомогати. До того ж, вчити нову мову програмування для однієї задачі — нераціонально.

Окрім вбудованих скриптів мовою MQL5, Metatrader створили торгове API, яке допомагає автоматизувати всі процеси торгівлі та яке можна використовувати потенційно з будь-якою мовою програмування. Це саме те, що треба, щоб створити повноцінного торгового бота.

Торговий брокер

Щоб торгувати на Forex, потрібен посередник, якого називають брокер. Було вибрано gerchik.co, бо отримано багато позитивних відгуків від інших знайомих трейдерів, які довго працюють з цим брокером. Через брокера буде можливість створювати торговельні облікові записи, поповнювати рахунки, виводити кошти і т.п.

Окрема можливість, яка є у брокера Gerchik&Co, це рейтинг TIMA-рахунків. TIMA-рахунок (або ПАММ-рахунок) — це сервіс, за допомогою якого інвестори можуть заробляти, не торгуючи. Інвестори вкладають кошти в рахунки трейдерів, трейдери збільшують ці кошти, і прибуток ділиться між інвесторами та трейдерами автоматично за домовленою схемою. Gerchik&Co публікує рейтинг своїх найефективніших TIMA-рахунків, щоб залучати інвесторів та мотивувати трейдерів ретельно працювати над своїми стратегіями. Цей рейтинг — це можливість реалізувати ефективну стратегії та обзавестися колом інвесторів, які будуть збільшувати спільні прибутки.

Тестування на історії, демо рахунок та справжня торгівля

Щоб створити ефективного бота, треба розділити всю систему на 3 частини: тестування на історії, тестування на демо рахунку та торгівля за справжні кошти в реальному часі.

Тестування на історії — це можливість перевірити ефективність своєї стратегії на основі історичних даних торгової платформи.

Кожна торгова платформа зберігає певний часовий об’єм свічок різних таймфреймів. Чим менше таймфрейм, тим більше даних доводиться зберігати і тим менше період, за який ці дані доступні. Наприклад, торгова платформа Metatrader зберігає лише 3 місяці хвилинних свічок, але декілька років годинних. Це все через те, що за за один і той самий період кількість хвилинних та часових свічок буде сильно відрізнятися.

Щоб використати історичні дані для тестування своєї стратегії, треба написати програму, яка буде ітеруватися по цих даних від початку до кінця, імітуючи торгівлю в реальному часі.

При створенні алгоритму тестування на історії важливо все налаштувати так, щоб середовище тестування було максимально наближено до середовища реальної торгівлі. Один з моментів, які треба врахувати, це постійні торгові витрати при закритті кожної угоди, які сплачує трейдер. Як вже було сказано раніше, ці витрати існують через spread, який постійно є та змінюється при торгівлі в реальному часі. Значення spread недоступне в історичних даних, тому торгові збитки від spread при тестуванні доводится вписувати в логіку функцій, які імітують виставлення ордерів.

При переході через 24:00 spread часто зростає дуже сильно. Тому вночі краще вимикати торгівлю, бо торгові витрати надто високі при укладанні торгових угод. Треба визначити максимально допустимий для себе spread та вмикати торгівлю ближче до ранку, коли значення spread увійде в допустимий діапазон.

Етап тестування на історії дуже важливий, щоб відточити стратегію, перевірити гіпотези та подивитися на потенційні результати.

Коли стратегія перевірена на історичних даних, можна переходити до **тестування на демо рахунку в реальному часі.**

Демо рахунок відрізняється від справжнього тільки тим, що на демо рахунку трейдер торгує паперовими грошима. Тобто можно в будь-який момент часу написати баланс 10 тис. доларів та ними торгувати, проте вивести ці гроші на свій справжній рахунок неможливо.

Торгівля на демо рахунку відбувається в реальному часі й майже нічим не відрізняється від справжньої торгівлі. Єдина відмінність в тому, що втрати та прибутки трейдера на демо рахунку не впливають на його справжній гаманець.

Етап тестування на демо рахунку важливий, щоб подивитися на роботу стратегії в реальних умовах, але водночас вберегти себе від втрати справжніх коштів, якщо щось піде не так.

Справжня торгівля в реальному часі — це те, для чого створюється вся ця система. Це кінцевий етап, де йде заробіток та втрата справжніх коштів з карманів трейдера. Це та сама реальна реальність, де 80—90% трейдерів втрачають свої гроші. Цей момент важливо розуміти, коли трейдер збирається виходити на ринок. Бо ринок — це місце, яке не щадить нікого. Тут все строго й жорстко. Тому перед тим,

як виходити в люди, критично важливо ретельно пройти всі попередні етапи тестування та все добре перевірити.

Оптимізація стратегії

Майже у всіх стратегій є певні параметри, які ці стратегії використовують для прийняття рішень. Ефективність стратегії напряму залежить від значень цих параметрів. Можна було б просто прикинути оптимальні значення, але це нераціональний підхід. На щастя, можна використовувати математичні алгоритми для оптимізації параметрів.

Зрештою, тестування на історії зводиться до деякої функції, яка може приймати параметри стратегії та повертати значення ефективності стратегії з цими параметрами. Для цієї функції можна легко використати математичний алгоритм мінімізації. Щоб отримати найбільше значення ефективності, треба просто ставити знак “-” біля результату, який повертає функція. Але алгоритм мінімізації треба ще обрати.

Є різні типи алгоритмів мінімізації: з градієнтом, з Hessian, без деривативів і т.д. При виборі алгоритму ніхто не може наперед точно сказати, який алгоритм буде найкращим. Треба звузити поле алгоритмів за рахунок знань особливостей своєї функції та просто тестувати на практиці.

Для обраної для цього проєкту складної стратегії дуже важко порахувати якісь деривативи, бо ядро стратегії нетривіальне, не зовсім лінійне і буде постійно змінюватися в процесі внесення покращень. Тому для цього випадку можна розглядати алгоритми мінімізації без деривативів.

2 головних таких алгоритми, про які наголошують бібліотеки оптимізації, — це Powell та Nelder-Mead. Вони були створені приблизно в один і той самий час і на практиці мають близькі результати.

Багато трейдерів, які також займаються автоматизацією торгівлі, часто згадували саме Powell у розмові про оптимізацію. Тому, зрештою, було вибрано саме його.

Алгоритму Powell треба передати початкові значення параметрів та обмеження діапазону пошуку значень для кожного параметру. Для ініціалізації алгоритму початкові значення параметрів стратегії можно підібрати за рахунок власних знань предметної області. Теж саме стосується й діапазонів пошуку значень. На виході функції мінімізації отримується локальний мінімум функції і можливість використати вирахувані параметри, щоб покращити ефективність стратегії.

Розділ 2. Проектування та розробка системи

Складові системи та використані технології

Для автоматизованої системи торгівлі критичним моментом є надійність, бо кожна помилка в коді потенційно може призвести до втрати справжніх коштів. Складність та об'єм обраної для цього проєкту стратегії дуже великі, тому запуск стратегії на історичних даних — це потенційно дорога операція. Важливо, щоб процес оптимізації стратегії не займав надто багато часу, бо тоді буде ставати все складніше проводити тестування та перевірку гіпотез.

Враховуючи всі ці моменти, врешті-решт, було обрано мову Rust. Ця мова відома за рахунок трьох своїх якостей: надійність, швидкість та продуктивність. Надійність гарантується за рахунок статичної та строгої типізації та розумного компілятора, який додатково відловлює помилки з lifetime та race conditions. Швидкість забезпечується за допомогою тієї самої компіляції, яка збирає початкову програму в нативний машинний код з можливостями максимальної оптимізації. Продуктивність здобувається за рахунок багатого синтаксису та високорівневих zero-cost абстракцій.

Єдина частина, де поки що використовується Python, — це сам алгоритм оптимізації. Тут поки є прив'язка до бібліотеки scipy з її функцією minimize. Потенційна альтернатива на Rust — це argmin, але там поки що немає імплементації алгоритму Powell. Важливо додати, що при оптимізації найбільше часу займає саме виклик функції, яка оптимізується. Тому якщо функція буде виконуватися швидко за допомогою Rust, то Python не буде гальмувати процес.

Дизайн та архітектура

Перша версія системи була написана в стилі ООП. Але, врешті-решт, було помічено багато особливостей цієї парадигми, які на практиці не дуже сподобалися:

- дуже багато boilerplate коду;
- важко перевикористовувати дані в різних контекстах, бо дані тісно пов'язані з методами;
- скрізь графи об'єктів, все на все посилається і не завжди зрозуміло, хто має виконувати дію;
- object–relational impedance mismatch;
- рекурсивні посилання і схожі ситуації з посиланнями призводять до нетривіальних лайфтаймів. У більш низькорівневих мовах через це доводиться використовувати різні смарт-поінтери, типу weak reference, з якими поступово код перетворюється в пекло.

Через всі ці моменти було вирішено використати мікс Data-Oriented + functional + clean & hexagonal архітектуру.

Всі підсистеми та їх частини покриті юніт та інтеграційними тестами, щоб ще більше покращити надійність.

Підсистеми

Бібліотека автоматизованої торгівлі складається з таких підсистем:

1. Ядро стратегії.
2. Виконавець стратегії.
3. API ринкових даних.
4. Синхронізація історичних даних.
5. Кешування історичних даних.
6. Торгове API.
7. Відображення результатів тестування.
8. Оптимізація стратегії.

```rust
pub fn run_iteration(
    tick: &BasicTick,
    candle: Option<&BasicCandle>,
    signals: StrategySignals,
    stores: &mut StepBacktestingStores,
    params: &impl StrategyParams,
) -> Result<()> { todo!() }
```

Ядро стратегії

Ядро стратегії — це сама логіка стратегії: коли купляти, коли продавати і т.д. Точка входу для запуску ядра стратегії — це функція run_iteration. Ця функція приймає 5 сутностей:

- поточний тік (tick). Передається обов'язково. Нова ітерація не запускається доти, доки не буде отримано новий тік;
- поточна свічка (candle). Передається необов'язково, бо нова свічка з'являється набагато рідше, ніж новий тік. Без свічки логіка стратегії виконується, а без тіка вже ні, бо просто недостатньо даних.
- сигнали для додаткового керування ходом виконання стратегії (signals). Можуть змінюватися в процесі виконання програми. Одним з таких сигналів може бути сигнал заборони торгівлі, який передається вночі з 24:00 до того часу, поки не зменшиться spread;
- сховища додаткових даних, з якими працює стратегія (stores). Логіка стратегії може мати багато різних сутностей, які вона використовує по ходу своєї роботи. Поточні значення цих сутностей знаходяться в сховищах, які можуть зберігати дані в пам'яті комп'ютера або в базах даних;
- параметри стратегії (params). Це ті числові значення, які використовує стратегія і які треба оптимізувати для кращих фінансових результатів.

Виконавець стратегії

Виконавець стратегії — це керуючий код, який запускає ітерації ядра стратегії. Його задача — поєднання різних підсистем в одну працюючу програму.

Логіка виконавця стратегії може різнитися в залежності від конкретної стратегії та її режиму виконання. Наприклад, для тестування на історичних даних виконавець стратегії спочатку отримує всі історичні дані, а потім поступово згодовує їх функції run_iteration обраної стратегії. А коли пишеться логіка виконавця стратегії для торгівлі в реальному часі, історичні дані не запитуються. Натомість періодично відправляються запити до metaapi на отримання поточних тіка та свічки.

При торгівлі в реальному часі в логіку виконавця стратегії додається правило перед вихідними серіалізувати весь поточний стан стратегії та повністю завершувати виконання програми. Справа в тому, що на вихідних ринок не працює. Можна було б на період вихідних написати логіку, яка б просто ходила в циклі та нічого не робила. Але якщо програма постійно працює, то немає простої можливості вносити правки та покращення. Єдине, що можна зробити, це примусово зупинити виконання всієї програми. Але у випадку торгового бота — це небезпечна операція: торговий обліковий запис може опиниться в стані, який призведе до втрати коштів, якщо вручну все не проконтролювати. Тому краще скористатися часом, коли боту нема що робити та просто безпечно та контрольовано його зупинити. Тоді на вихідних можна вносити правки до коду програми, а після вихідних знову запускати бота з новою конфігурацією.

Також при торгівлі в реальному часі у виконавця стратегії з’являється додаткова задача відправляти повідомлення про аварійне зупинення програми розробникам. Це важливо, щоб вчасно відреагувати на проблему, проконтролювати стан торгового облікового запису та не допустити втрати коштів.

```rust
pub trait MarketDataApi {
    fn get_current_tick(&self, symbol: &str) -> Result<BasicTick>;

    fn get_current_candle(&self, symbol: &str, timeframe: Timeframe) -> Result<BasicCandle>;

    fn get_historical_candles(
        &self,
        symbol: &str,
        timeframe: Timeframe,
        end_time: DateTime<Utc>,
        duration: Duration,
    ) -> Result<Vec<Option<BasicCandle>>>;

    fn get_historical_ticks(
        &self,
        symbol: &str,
        timeframe: Timeframe,
        end_time: DateTime<Utc>,
        duration: Duration,
    ) -> Result<Vec<Option<BasicTick>>>;
}
```

API ринкових даних

Не хотілося б всього бота тісно зв’язувати з REST metaapi. Тому було створено інтерфейс MarketDataApi, який не знає нічого про реальну імплементацію. Його методи повертають структури, які цікавлять бізнес логіку, а не ті, які повертає REST metaapi.

Методи get_current_tick та get_current_candle використовуються в торгівлі в реальному часі, щоб періодично отримувати нові поточні тік та свічку. Методи get_historical_candles та get_historical_ticks використовуються при тестуванні стретегії на історії та повертають відповідно свічки та тіки за вибраний проміжок часу.

При тестуванні на історії немає можливості отримати справжні історичні тіки, бо небагато таких API, які їх зберігають. А ті, які зберігають, можуть повернути тільки невелику кількість. Тому тіки треба імітувати. Для цього використовуються свічки з хвилинним таймфреймом. Цієї частоти буде цілком достатньо, щоб зберігати наближеність до умов реальної торгівлі.

get_historical_ticks всередині працює так, що він насправді також отримує свічки, як і get_historical_candles. Проте, на відміну від get_historical_candles get_historical_ticks повертає масив тіків, а не

свічок. Для цього свічки просто трансформуються в тіки. Тік зберігає в собі 3 значення: ask, bid та час. Свічка складається з 5 значень: open, close, high, low та час. Щоб трансформувати свічку у тік, в ask та bid записується значення close хвилинної свічки. Але в цьому випадку немає різниці між ask та bid, тобто немає spread, який завжди присутній у справжній торгівлі та зменшує прибутки трейдера. Тому, як вже було сказано раніше, торгові витрати від spread при тестуванні на історії треба просто записувати в логіку функцій, які імітують купівлю та продаж.

Синхронізація історичних даних

API історичних даних вимагає дату останньої сутності та кількість сутностей, які треба отримати. На жаль, це API не гарантує цілісності даних. Тобто з отриманих свічок та тіків можуть бути пропуски. Наприклад, треба отримати 60 хвилинних свічок до 12:00. У цьому випадку є надія, що перша свічка буде 11:00. Проте через пропуски перша свічка може бути й 10:35 або будь-який інший час раніше 11:00. Тобто через те, що API повинно повернути визначену кількість сутностей, але деякі сутності можуть бути пропущені, повертаються зайві дані.

Тобто є дві проблеми: пропуски та зайві дані. Коли налаштовується виконавець стратегії, треба зробити так, щоб тіки та свічки послідовно та синхронізовано передавалися ітерації ядра стратегії. Якщо отримані історичні дані десинхронізовані, то в один момент стратегія може почати працювати з неіснуючими ситуаціями. Наприклад, йде робота з годинним таймфреймом для свічок. Поточний тік — 12.05.2022 11:00. У цей час очікується поява нової свічки з часом 12.05.2022 10:00. Проте через десинхронізацію даних замість свічки з таким часом отримується свічка за 14.05.2022 16:00. І виходить так, що стратегія далі завжди буде отримувати несумісні тіки та свічки, і всі результати ефективності, які будуть отримані, будуть стосуватися даних казкового королівства. І, звичайно, всі оптимізації, які будуть зроблені на таких фейкових даних, не вплинуть на отримання кращих результатів у справжній торгівлі.

Щоб вирішити проблеми пропусків, було написано систему синхронізації, яка вставляє в незаповнені проміжки між свічками

значення None. Наприклад, зараз йде робота з хвилинними свічками. Metaapi повернуло свічки [12:00, 12:03, 12:04, 12:06]. Після відпрацювання алгоритму синхронізації на виході отримуються [12:00, None, None, 12:03, 12:04, None, 12:06]. За допомогою значень None на основі знання таймфрейму можна тепер чітко визначити час будь-якої свічки масиву за її індексом. Наприклад, якщо є інформація, що свічка з індексом 0 має час 12:00, то можна чітко розуміти, що свічка з індексом 5 буде мати час 12:05, навіть, якщо її значення None.

Коли пропуски заповнені, залишається проблема зайвих свічок або тіків. Через пропуски, які були, перший тік та перша свічка, які повертає metaapi, можуть мати різний час. Щоб не було багато даних на екрані, для прикладу представляються тіки у вигляді 30-хвилинних свічок замість хвилинних. А для свічок вибереться часовий таймфрейм. Наприклад, у metaapi запитуються 4 годинних свічки до 15:00. Відповідно, щоб отримати сумісні дані, запитуються також 8 30-хвилинних свічок до 15:00. В ідеальному варіанті хотілось би мимати такі свічки та тіки:

- годинні: [12:00, 13:00, 14:00, 15:00];
- 30-хвилинні: [12:30, 13:00, 13:30, 14:00, 14:30, 15:00, 15:30, 16:00].

Але по факту можна отримати такі:

- годинні: [11:00, 12:00, 14:00, 15:00];
- 30-хвилинні: [11:00, 11:30, 12:00, 13:30, 14:00, 14:30, 15:30, 16:00].

Після заповнення проміжків отримуються такі масиви даних:

- годинні: [11:00, 12:00, None, 14:00, 15:00];
- 30-хвилинні: [11:00, 11:30, 12:00, None, None, 13:30, 14:00, 14:30, None, 15:30, 16:00].

Можна побачити, що свічок стало 5, а тіків 11. Не проблема, якщо буде йти робота з трохи більшої кількістю даних, проте важливо, щоб тіки та свічки з самого початку та до кінця були синхронізованими. Для стану, в якому знаходяться дані, достатньо виконати операцію перетину часових значень свічок та тіків. Зрештою, отрумуються такі масиви даних:

- годинні: [11:00, 12:00, None, 14:00, 15:00];

- 30-хвилинні: [11:30, 12:00, None, None, 13:30, 14:00, 14:30, None, 15:30, 16:00].

На виході 5 свічок та 10 тіків. Це трохи більше ніж 4 та 8 відповідно, але це зовсім нічого не псує. Головне, що всі дані синхронізовані та всі часові значення елементів масиву передбачувані.

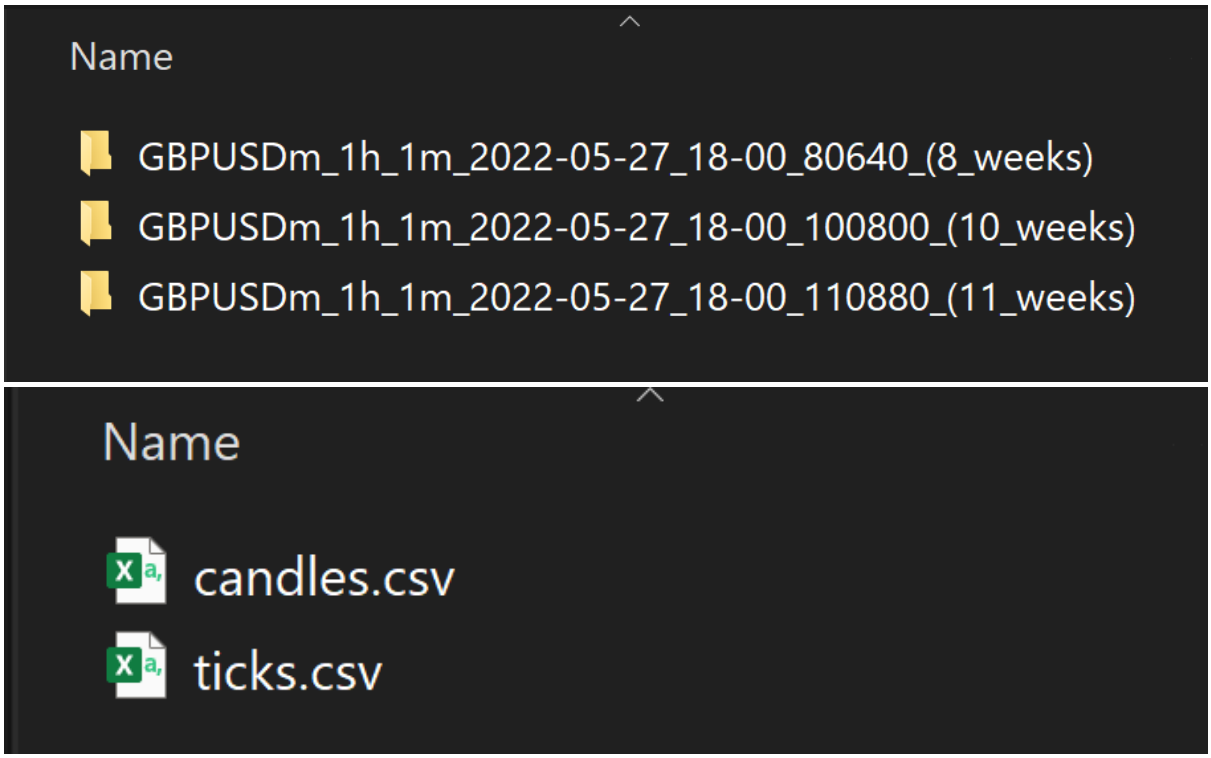
Залишилося спробувати запустити основний цикл виконавця стратегії та переконатися, що нічого не зламається. Як вже було показано раніше, функція ітерації ядра стратегії отримує пару (Tick, Option(Candle)). Це визначення каже про те, що ітерація обов'язково приймає тік та необов'язково приймає свічку. Тепер подивимось покроково, які аргументи будуть фактично передані при синхронізованому стані історичних даних:

1. (11:30, None).
2. (12:00, 11:00).
3. Можна було б передати (None, None), але тіка немає, тому ця ітерація пропускається.
4. Можна було б передати (None, 12:00), але тіка немає, тому ця ітерація пропускається.
5. (13:30, None).
6. (14:00, None). Ми не маємо свічки за 13:00, але хоча б тік є, тому ітерація зможе відпрацювати.
7. (14:30, None).
8. Можна було б передати (None, 14:00), але тіка немає, тому ця ітерація пропускається.
9. (15:30, None).
10. (16:00, 15:00).

Можна побачити, що після синхронізації з'явилася можливість керувати процесом ітерації даних дуже чітко та передбачувано. Всі дані, які повернуло metaapi, були максимально повно використані. У прикладі вище було навмисно зроблено дуже багато пропусків, тільки щоб показати процес синхронізації та всі особливості ітерування даних такого формату. У справжніх історичних даних, які повертає metaapi, пропусків дуже мало, а таймфрейми, типу годинного, частіше всього взагалі не мають ніяких пропусків.

Кешування історичних даних

HTTP запит на отримання великої кількості історичних даних — дорога операція, тому для оптимізації використовується кешування. Свічки та тіки в рамках одного пакету даних для тестування стратегії зберігаються в csv файли всередині спільної папки:



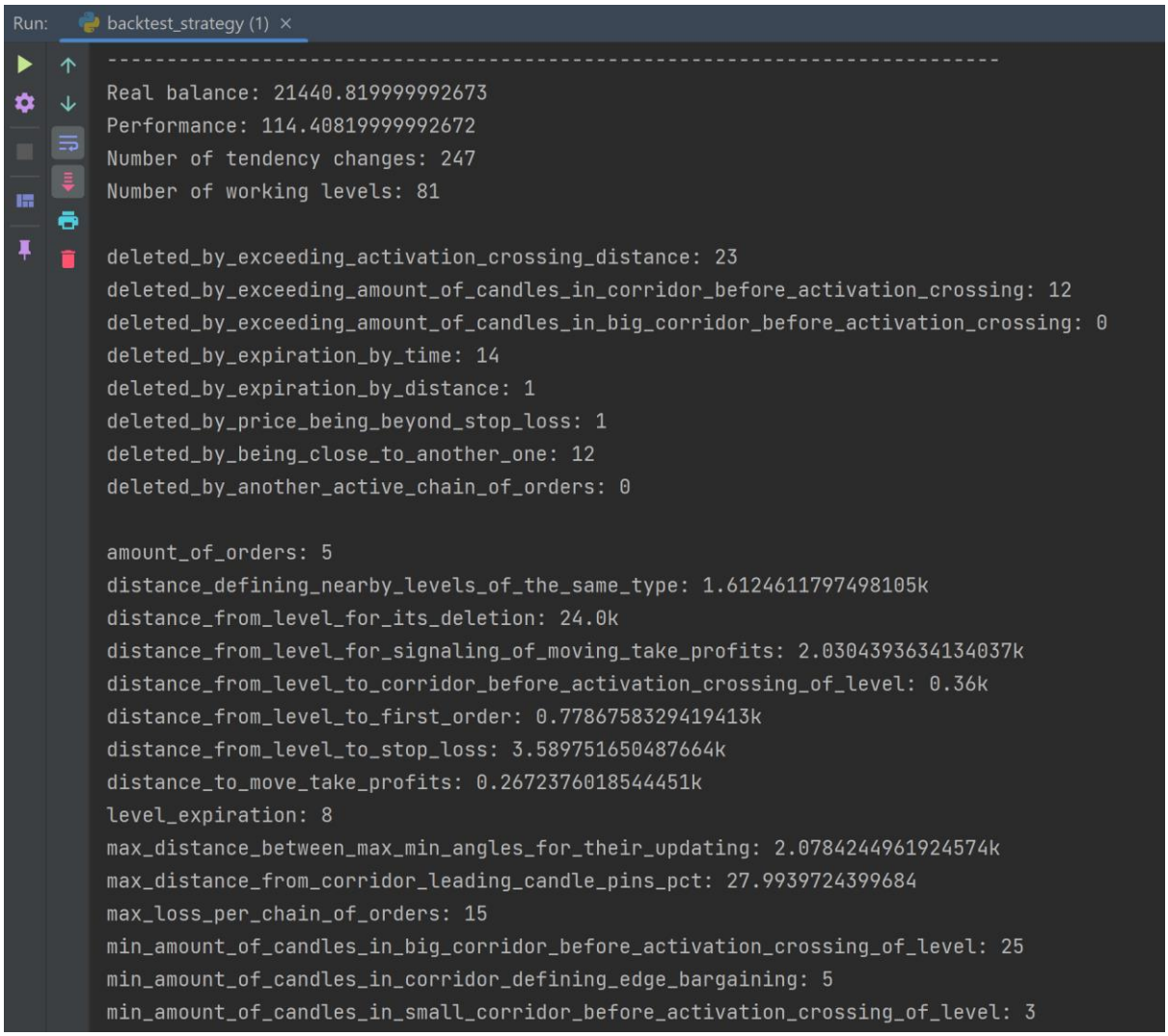
За назвою папки можна легко зрозуміти, який пакет даних у ній зберігається.

Торгове API

Торгове API являє собою інтерфейс з методами, які використовуються ядром стратегії для виставлення ордерів. Для тестування на історії реалізація цього API тільки імітує справжнє виставлення ордерів, а для справжньої торгівлі запит пересилається metaapi для укладення справжніх торгових угод на ринку Forex.

Фейкова реалізація торгового API для тестування на історії містить логіку, яка враховує торгові витрати від spread. Через те, що немає можливості отримати справжні історичні значення spread, використовується середнє значення.

Відображення результатів тестування на історії

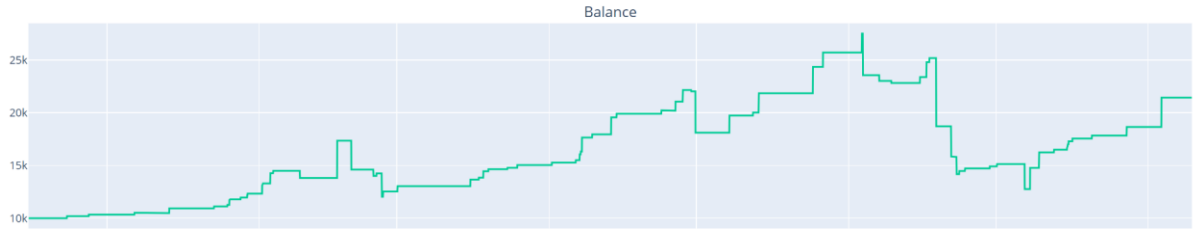


Перше, що хотілось би побачити в якості результатів тестування стратегії на історії, це новий баланс та ефективність. Наприклад, на цьому скріншоті стратегія заробила 11 тис. доларів за 9 місяців: з 10 тис. доларів балансу за замовчуванням стратегія зробила 21 тис. доларів.

Поряд з новим балансом та ефективністю можна відображати інші дані, які цікаві для аналізу результату стратегії. Наприклад, на скріншоті вище додатково показуються значення параметрів, з якими була запущена поточна версія стратегії та аналітичні дані щодо кількості видалених сутностей, які використовує стратегія.

Аналіз графіків результатів

Окрім результатів у текстовому форматі, хотілося б ще бачити візуальний формат.



На скріншоті вище відображається зміна балансу облікового запису в процесі ітерування по історичних даних. Цей графік може допомогти визначити моменти, коли втрачаються гроші, і в подальшому зробити зневадження та внести правки в логіку стратегії для покращення фінансових результатів.

Окрім графіку балансу, можна створювати будь-які інші графіки, які будуть цікаві для аналізу результатів. Наприклад, для поточної стратегії є певні контрольні точки, пов’язані з виставленням ордерів. У ядрі стратегії реалізовано алгоритм, який по ходу ітерації стратегії збирає необхідні дані, а по завершенню формує інтерактивний графік свічок з історичних даних. На цьому графіку контрольні точки позначені горизонтальними лініями різних кольорів. Графік можна збільшувати, зменшувати, скролити, дивитися значення свічок та слідкувати за рівнями, на яких виставлені контрольні точки.

Оптимізація стратегії

Як вже було сказано раніше, для оптимізації своєї стратегії було використано алгоритм Powell. Його робота схожа на таку послідовність кроків:

1. Взяти перший параметр функції та змінювати його в указаному діапазоні до моменту, поки кращі значення перестануть з'являтися.
2. Повторити ті самі дії для кожного параметру функції.
3. Повторювати повні ітерації по всіх параметрах, допоки результати поточної ітерації залишаються краще, ніж результати попередньої ітерації

У реалізації алгоритму Powell у бібліотеці scipy було помічено один цікавий момент, який можна покращити: по ходу оптимізації значень одного параметру можуть проскакувати результати, які будуть трошки краще, ніж той, який вибере врешті Powell. Тому було вирішено створити невелику обгортку навколо Powell, щоб перехоплювати найкращі результати. Працює вона так: алгоритм Powell запускається для кожного параметра окремо. Всі результати, які продукує алгоритм по ходу оптимізації одного параметру, перехоплюються та записуються в окремий список. Після завершення процесу оптимізації параметру кращим результатом вважається найкращий результат зі списку, а не той який повернув Powell. Часто це дозволяє перехопити той насправді кращий результат, з яким стикався алгоритм в процесі своєї роботи. Далі береться цей найкращий результат та передається на вхідні параметри для процесу оптимізації другого параметру. І так йде все до останнього параметру. Далі алгоритм запускається мінімум ще раз для всього набору параметрів. Якщо результат другої повної ітерації краще, ніж результат першої повної ітерації, то повна ітерація запускається знову. І далі алгоритм буде повторно запускатися, поки поточний результат є краще попереднього.

На практиці результат цієї невеликої модифікації показав кращі результати, ніж стандартна версія Powell бібліотеки scipy.

Розміщення на сервері та планувальник задач

Один з варіантів — це запустити бота на локальній машині. Але в цьому є деякі очевидні проблеми: постійний фоновий процес; проблеми з інтернетом; випадкове втрата живлення. Тому краще розмістити бота на окремому сервері, де всі ці проблеми давно вирішені.

Програма автоматично повністю завершує своє виконання перед вихідними, щоб була можливість вносити правки в код, поки ринок не працює. Але хтось повинен запустити робота знову після вихідних. Робити це вручну — незручно, тому краще в планувальнику задач операційної системи створити завдання автоматично запускати програму у визначений час.

Висновки

У цій роботі було детально розглянуто весь процес та особливості створення автоматизованої системи трейдингу. Можна сказати, що процес покращення торгової інфраструктури та логіки стратегій — нескінченний. Завжди є ще щось, над чим можна працювати.

У найближчий час планується додатково реалізувати періодичний автоматичний запуск оптимізації стратегії. Ринок змінюється, тому при справжній торгівлі в реальному часі періодично треба отримувати більш свіжі історичні дані, запускати на них оптимізацію стратегії та оновлювати параметри стратегії. Все це можно періодично робити вручну, але навіщо, якщо можна автоматизувати. Алгоритм буде автоматично запускатися щомісяця, отримуватиме історичні дані, наприклад, за останні 3 місяці, потім на цих даних запускатиме оптимізацію стратегії та параметри найкращого результату записуватиме у файл, який використовує стратегія при торгівлі в реальному часі.

Список використаних джерел

- 36-годинний курс з алгоритмічного трейдингу на Python: https://www.udemy.com/course/algorithmic-trading-with-python-and-machine-learning/.

- Види алгоритмів оптимізації та їх особливості: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html
- Інструкції по застосуванню API ринкових даних та торгового API: https://metaapi.cloud/docs/client/