МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

ИЗ РАЧУНАРСТВА И ИНФОРМАТИКЕ

Примена вештачке интелигенције за анализу финансијских вести и употреба исте у инвестиционим стратегијама

Аутор:
Димитрије Марић, IVб

Ментор:
проф. Милош Арсић

Београд, мај 2025.

Садржај

Резиме

У овом раду представљен је систем за аутоматску обраду, анализу сентимента и класификацију финансијских вести. Циљ је био да се помогне у доношењу инвестиционих одлука повезивањем релевантних чланака са компанијама и израчунавањем sentiment-а садржаја. Кључне компоненте обухватају модул за повлачење вести са WEB-а путем MarketAux API-а [1] (news_fetcher), парсирање вести (news_parser), обраду и одређивање сентимента (news_processing) и помоћне алате за повезивање са MarketAux API-јем, AstraDB [2], OpenAI моделима [3] и Newspaper3k [4]. Резултати тестирања показују високу поузданост у издвајању релевантних вести и конзистентност сентимента, уз могућност даљег побољшања оптимизацијом параметара.

Увод

У савременом финансијском екосистему, динамика тржишта и велика количина доступних информација чине доношење инвестиционих одлука све захтевнијим. Финансијске вести стижу непрекидно и у огромном обиму, чинећи ручну обраду спором, скупоценом и подложној људској грешци. Истовремено, савремени пробој у области вештачке интелигенције и обраде природног језика омогућио је развој алата који текст претварају у векторске репрезентације и анализирају сентимент садржаја са високим степеном поузданости. Релевантни радови у овој области обухватају коришћење трансформер модела за анализу сентимента, као и претрагу векторских база података за проналажење сличности између два текста. Међутим, већина постојећих решења или захтева компромис између брзине и прецизности, или није оптимизовано за интеграцију у трговачким системима у реалном времену. [5] [6] [7]

Циљ овог рада је представити систем за аутоматско прикупљање, обраду и анализу сентимента финансијских вести, са фокусом на везивање релевантних чланака уз компанијске ентитете и извештавање о позитивним или негативним трендовима који могу утицати на инвестиционе стратегије. Систем је реализован као комбинација C++ модула: news_fetcher за преузимање наслова и метаподатака преко MarketAux API-ја [1] [8], news_parser за парсирање одговора добијеног од news_fetcher-а и news_processing за векторизацију текстова и класификацију сентимента; као и Python скрипти: marketaux_fetcher.py (интерфејс према MarketAux API-ју [1] [8]), article_scraper.py (преузимање чланака у целости), company_matcher.py (повезивање вести са компанијама помоћу векторске базе података у AstraDB-у [2]) и sentiment_analyzer.py (коришћење HuggingFace трансформера [8] за финалну процену сентимента). Поред тога, систем користи паралелну обраду која у потпуности искоришћава могућности савременог хардвера и механизме за синхронизацију ресурса, што омогућава високе перформансе и поуздано логовање рада сваке компоненте.

У следећим поглављима биће детаљно описани сви коришћени алати и методе, захтеви и дизајн система, начин имплементације појединачних компоненти, као и резултати тестирања на стварним скупова података. Рад ће се завршити анализом

постигнутих перформанси, дискусијом о могућим унапређењима и закључцима који указују на даље правце истраживања и примене оваквог система у инвестиционом контексту.

Теоријски оквир

У овом поглављу представљују се основни појмови и методе на којима се темељи систем за аутоматску обраду и анализу сентимента финансијских вести. Пре уласка у детаље конкретних компоненти, објашњавамо три кључна концепта: векторску репрезентацију текста, анализу сентимента и паралелну обраду података. Они чине теоријску основу за избор алгоритама и технологија које су имплементиране у нашем систему. У даљим поднасловима биће приказано како се свака од ових метода примењује у оквиру пројекта, које алате користимо и којим механизмима постижемо жељену прецизност и перформансе.

Векторска репрезентација текста

Векторска репрезентација текста (embedding) представља технику у природној обради језика којом се текст или његов сегмент претвара у густ, континуиран вектор у високо-димензионалном простору. Идеја произлази из дистрибутивне семантике и омогућава да семантички односи између појмова буду квантификовани као растојања или углови између вектора.

Процес генерисања векторског embedding-а обично обухвата следеће кораке:

1. Токенизација и предобрада: Текст се разбија на мање јединице (токене), уклања се пунктуација и извршава нормализација.
2. Инференција embedding модела: Припремљени токенизовани улаз пролази кроз претходно тренирани модел, који за сваки улазни токен или целокупан текст враћа нумерички вектор фиксне дужине.
3. Композиција вектора: Уколико модел генерише векторе за сваки токен, онда се они комбинују у један вектор који представља цео текст.

Типичне димензије embedding вектора крећу се од неколико стотина до више хиљада. Повећање броја димензија обично побољшава способност модела да представи сложене семантичке односе и разликује фине нијансе значења, али истовремено пасте и потреба за рачунарским ресурсима, меморијска заокупљеност и ризик од

уношења шума (curse of dimensionality). Избор оптималног броја димензија представља компромис између прецизности репрезентације и перформанси система.

Главна предност векторских репрезентација лежи у могућности да се текстуални подаци чувају и претражују као вектори у базама података. Када сваки текст има свој вектор, сличност између два текста може се лако израчунати и омогућава:

- Прецизну претрагу: Враћање текстова чији су вектори „блиски“ вектору упита, без ослањања на тачну подударност кључних речи.
- Кластеризацију и визуализацију: Груписање сродних докумената у кластере и њихово приказивање у смањеном димензионалном простору.
- Семантичку анализу: Проналажење парафраза, препорука и повезивање садржаја по значењу, а не само по површним кључним речима.

Ова својства чине embedding критичним алатом за савремене апликације као што су паметне претраге, системи препорука, анализе сентимента и други облици семантичке обраде текста. [3] [5] [6]

Анализа сентимента

Анализа сентимента представља метод у природној обради језика којим се аутоматски одређује емоционални тон текста, односно да ли је садржај позитиван, негативан или неутралан. Циљ је претварање квалитативног у квантитативни показатељ, што омогућава статистичку обраду и праћење динамике расположења у великим скуповима текстова.

Процес анализе сентимента обухвата следеће кораке:

1. Токенизација и предобрада: Текст се разбија на токене, уклањају се неизражавајући карактери, врши нормализација и, по потреби, лематизација.
2. Инференција модела: Припремљени токенизовани улаз пролази кроз претходно истренирани модел, који за улазни текст враћа скуп класификационих резултата или једану кумулативну оцену.

3. Сабирање оцена: На нивоу текста или објекта сабирају се појединачни резултати и израчунава коначни sentiment score.

Типични модели за анализу сентимента крећу се од једноставних лексичких речника до вишеслојних неуронских мрежа и трансформера са димензијама embedding-а од неколико стотина до неколико хиљада димензија. Повећање броја слојева и димензија обично побољшава фину осетљивост на контекст и нијансе емоција, али истовремено захтева веће рачунарске ресурсе и може увести шум услед прекомерне сложености (curse of dimensionality). Избор одговарајућег нивоа сложености представља компромис између прецизности и перформанси система.

Главне примене и предности анализе сентимента обухватају:

- Прецизна класификација: Омогућава категорисање великих скупова текстова по емоционалном тону без ручне обраде.
- Праћење у реалном времену: Подршка у праћењу брзе промене јавног расположења на друштвеним мрежама и у медијима.
- Доносивост одлука: Интеграција sentiment score-а као једног од кључних показатеља у маркетиншким, политичким и финансијским стратегијама.

Ова својства чине анализу сентимента критичним алатом за савремене апликације као што су мониторинг бренда, системи упозорења на кризне ситуације и предиктивна аналитика у пословним процесима. [7] [9]

Паралелна обрада и синхронизација

Паралелна обрада и синхронизација представљају технике које омогућавају да се више независних задатака унутар једне апликације извршава истовремено на више процесорских језгара, чиме се смањује укупно време обраде и у потпуности искоришћавају могућности савременог хардвера. Кључна идеја је декомпозиција посла на мање јединице (нит или процес) и координација њиховог приступа заједничким ресурсима како би се избегли проблеми као што су условне трке (race condition) и закључавања (deadlock).

Процес имплементације обухвата следеће кораке:

- Декомпозиција задатака: Главни ток обраде дели се на независне потзадатке који се могу паралелно извршавати.
- Управљање нитима: Креирање и коришћење пулова нити (thread pools) или динамичко покретање нити у складу са бројем доступних процесорских језгара.
- Синхронизациони механизми: Применом узајамног искључивања (mutex), семафора и баријера контролише се приступ заједничким структурама података, док атомске операције омогућавају безбедне измене појединачних вредности без класичног закључавања.
- Балансирање оптерећења: Динамичко распоређивање потзадака и минимизација времена проведеног у закључавању (lock contention) осигуравају равномерно коришћење ресурса и смањују оверхед.

Приликом пројектовања паралелних система потребно је узети у обзир и следеће компромисе: повећано време преласка између контекста (context switching), додатни трошкови управљања нитима и сложеност координације, што може ограничити остварене добитке. Оптималан избор броја нити у односу на број физичких и виртуелних језгара представља најважнији фактор за линеарно скалирање перформанси.

Главне предности паралелне обраде и синхронизације су:

- Повећана пропусност: Истовремено руковање више задатака скраћује укупно време обраде великих скупова података.
- Смањење кашњења: Подзадаци се одмах обрађују по пријему, без очекивања на претходне операције.
- Искоришћење мулти-језгарних архитектура: Максимално коришћење физичких и хипертреадинг језгара, као и SIMD јединица.
- Конзистентност у реалном времену: Систем одржава стабилне перформансе чак и под високим оптерећењем.

Ове технике чине основу развоја високо перформантних апликација и кључне су за системе који захтевају брзу и поуздану обраду великих количина података у реалном времену. [9] [10]

Захтеви и дизајн система

У овом поглављу анализирају се све потребне спецификације и концепти који стоје иза имплементације система за аутоматску обраду и анализу финансијских вести. Прецизно се утврђују функционалности које систем мора да обезбеди: преузимање и парсирање вести, векторизацију текста, класификацију сентимента и повезивање са компанијским профилима, као и нефункционални аспекти као што су перформансе, скалабилност, поузданост и одрживост система. Тачно описани захтеви служе као темељ за пројектовање и развој, обезбеђујући да крајње решење испуни очекивања корисника и техничке стандарде.

Након дефинисања захтева, представићемо дизајн система, која је дизајнирана као скуп међусобно повезаних модула. Ово поглавље пружа јасан увид у начин организовања главних елемената, укључујући комуникацију преко API-ја, чување и претрагу векторских података и механизме за обраду података у реалном времену, у циљу разумевања начина претварања захтева у конкретна техничка решења.

Функционални и нефункционални захтеви

Дефинисање захтева представља кључну фазу у развоју софтверског система јер поставља темеље за све наредне кораке [11]. У следећим одломцима наводе се основни функционални захтеви, који описују шта систем мора да ради, као и нефункционални захтеви, који дефинишу услове под којима те функције треба да буду реализоване.

Функционални захтеви:

1. Преузимање финансијских вести: Систем мора да обезбеди аутоматско повлачење наслова и метаподатака у реалном времену.
2. Парсирање добијеног одговора: Преузети одговор у JSON формату треба прецизно распарсировати и издвојити кључне информације попут наслова, URL-а и датума објаве.

3. Преузимање пуних текстова чланака: На основу добијеног URL-а, систем покреће скрипту за екстракцију пуних текстова.
4. Векторизација текста: Комбиновани текст (наслов и тело чланка) треба послати у модел embedding-а и примити вектор фиксне дужине за семантичку репрезентацију.
5. Анализа сентимента: На основу текста чланака систем мора да генерише sentiment score.
6. Повезивање са компанијским профилима: Користећи векторску претрагу у векторској бази података, систем треба да идентификује и врати најрелевантнију компанију повезану са садржајем вести.
7. Логовање и праћење рада: Сваки корак процеса треба забележити у лог фајлу ради касније анализе и отклањања грешака.

Нефункционални захтеви:

1. Перформансе: Систем мора да обради сваки нови чланак у року од неколико секунди, уз минимално кашњење и просечну пропусност од најмање 60 уноса по минути.
2. Скалабилност: Решење треба да омогућава хоризонтално и вертикално повећање капацитета како би подржало раст броја извора вести и повећано оптерећење.
3. Поузданост: Мора бити обезбеђена непрекидна радна доступност и механизми поновног покретања у случају грешке без губитка података.
4. Одрживост и лакоћа одржавања: Код мора бити организован у модуле са јасном документацијом која омогућава брзе измене и надоградње.
5. Безбедност: Сви API позиви морају бити аутентификовани и заштићени, а осетљиви подаци се морају чувати у сигурном оквиру.

Дизајн система

Дизајн система осмишљена је као скуп јасно одвојених и међусобно повезаних компоненти које заједно обављају цео процес: од прикупљања финансијских вести до извештаја о сентименту. Овај модуларни приступ омогућава лако одржавање,

повећање функционалности и оптимализацију појединачних делова без утицаја на остатак система.

Први слој чини C++ модул news_fetcher, који комуницира са MarketAux API-јем [1] и преузима наслове, кратке описе и метаподатке у JSON формату. Следи слој news_parser, који од добијених одговора екстрактује кључне податке: наслов, URL, датум објаве.

На следећем нивоу, Python скрипта article_scraper.py преузима пуне текстове чланака преко Newspaper3k библиотеке [4]. Текст се затим прослеђује C++ модулу news_processing, где се комбиновани садржај, наслов и текст, шаље на векторизацију. Скрипта company_matcher.py користи нови embedding вектор [3] за претрагу колекције вектора компанија у AstraDB-у [2] [8] и идентификује оне чији су вектори семантички најближи новом вектору. Истовремено, текст пролази кроз sentiment_analyzer.py, који користи трансформерски модел [9] за одређивање sentiment score-а.

Комуникација између C++ и Python компоненти остварује се преко процесних позива и размене података у меморијским редовима [12]. Сваки корак се бележи у лог фајловима, што омогућава праћење перформанси и брзо решавање потенцијалних грешака.

За обраду више вести у реалном времену систем користи мултитреадинг [8], при чему се независни задаци: повлачење вести, парсирање JSON одговора, scrap-овање текстова, векторизација текстова, претрага векторске базе података и анализа сентимента, распоређују на више процесорских језгара. Конфигурационе параметре: брзину polling-а, API кључеве и параметре за AstraDB дефинише фајл settings.cfg, што омогућава подешавања без измена у самом коду.

Овакав дизајн омогућава систему да у потпуности искористи савремени хардвер, као и да буде лако проширен и прилагођен новим захтевима и изворима података.

Опис имплементације

У овом поглављу дајемо детаљан преглед конкретних корака и алата који су коришћени при израђивању система за аутоматску обраду и анализу сентимента финансијских вести. Након што смо у претходним деловима дефинисали захтеве и представили архитектуру, овде показујемо како су ти концепти преточени у јединствену и функционалну апликацију. Опис почиње избором и подешавањем окружења, затим наставља презентацијом помоћних алата и библиотека, а потом систематски пролази кроз имплементацију сваког модула и скрипте.

Прво ћемо објаснити конфигурацију развојног окружења, укључујући коришћење CMake [13] алата за компиловање C++ дела пројекта, као и Python окружење и главне пакете (као што су requests [14], newspaper3k [4] и transformers [9]). Навешћемо и структуру фајлова и улогу конфигурационог фајла settings.cfg, који омогућава подешавање појединих параметара без потребе за изменама у изворном коду.

Након тога, прелазимо на опис појединачних компоненти. На почетку је C++ модул news_fetcher који преузима метаподатке преко MarketAux API-ja [1], затим модул news_parser који обрађује JSON одговоре JSON, а потом news_processing који обавља векторизацију и анализу сентимента. У закључном делу ове секције пратићемо и рад Python скрипти: marketaux_fetcher.py, article_scraper.py, company_matcher.py и sentiment_analyzer.py, које се позивају из главног кода и извршавају парцијалне задатке као што су екстракција текста и семантичка претрага.

Конфигурација и помоћни алати

Ради лакше конфигурације систем, сва подешавања се учитавају из конфигурационог фајла settings.cfg позивом функције loadSettings унутар модула main.cpp, а резултујуће вредности се смештају у глобални објекат cfg. Основне опције конфигурације обухватају:

- lookbackSeconds и delaySeconds: временски интервали за временски прозор упита и паузу између следећих упита, како би се контролисало број захтева ка API-ју у оквиру модула news_fetcher.cpp.
- debug: промењива која контролишу покретање debug модула за детаљније праћење рада система.
- logToFile: промењива која одређује да ли ће се излаз чувати као log фајл унутар logs фолдера.
- useGPU: промењива која одређује да ли се у Python скриптама користи GPU или CPU.
- pythonInterpreter: путања до извршног Python интерпретатора који ће се користити за покретање свих помоћних скрипти.
- marketAuxBaseApi: основни URL за MarketAux REST API, који садржи filter_entities, language и api_token параметре [1].
- astraDBApplicationToken: апликациони токен за приступ DataStax Astra бази података путем Data API Client-a [2] [8].
- astraDBApiEndpoint: крајња тачка за DataStax Astra DB услугу, која омогућава векторску претрагу колекције компанија [2] [8].
- openAIApi: OpenAI API кључ који се користи за позиве ка OpenAI услугама у скриптама [3].

Систем користи следеће спољашње сервисе и алате као помоћне компоненте за прикупљање и чување података у реалном времену:

- MarketAux API REST: услуга за преузимање финансијских вести на основу параметара као што су језик, кључне речи и временски прозор, конфигурисана преко baseApi-a унутар settings.cfg фајла и опција published_after/published_before унутар модула news_fetcher.cpp [1].
- DataStax Astra DB: векторска база података која чува профиле компанија и њихову векторску репрезентацију за семантичку претрагу, приступ се остварује преко AstraDB Data API клијента уз помоћ скрипте company_matcher.py у оквиру модула company_matcher.cpp [2] [8].
- Polygon.io API: сервис за преузимање историјских и цена акција у реалном времену, које се касније користе за проналажење најбољих параметар за инвестициону стратегију [15].

У фајлу globals.cpp дефинишу се сви кључни објекти и структуре доступне преко целе апликације:

- Config cfg – конфигурација учитана из settings.cfg фајла.
- queue<NewsItem> newsQueue и queue<CompanyStatus> companyStatusQueue – редови за обраду вести и статуса компанија.
- set<string> seenUrls – скуп обрађених URL-ова за избегавање дупликата.
- InvestmentStrategy strategy, InvestmentList investmentList и unordered_map<string, vector<InvestmentNode*>> stockInvestmentsMap, double balance – објекти за дефинисање и извршавање инвестиционе стратегије.
- mutex newsQueueMutex и mutex companyStatusQueueMutex – закључавања за синхронизацију приступа глобалним редовима.
- ofstream logFile – уписивање логова.

Свака компонента система је синхронизована помоћу thread [16] објеката и mutex [17] закључавања за заштиту глобалних промењиви. Овим приступом обезбеђује се паралелна обрада, поуздано логовање и флексибилно прилагођавање понашања система према потребама корисника.

Модул news_fetcher

Модул news_fetcher.cpp реализује циклично прикупљање метаподатака финансијских вести преко MarketAux API-ja [1] и њихово прослеђивање на даљу обраду. По покретању, функција newsPolling у оквиру модула news_fether.cpp се извршава у засебној нити, која се покреће унутар main.cpp модула и која непрекидно ради током рада програма.

Унутар newsPolling функције:

1. Иницијално, вредност промењиве previousTime се добија позивом функције getUTCTimeOffset(cfg.lookbackSeconds), која је дефинисана унутар utils.cpp модула и за дато време X у секундама враћа време пре X секунди у ISO 8601 формату, што поставља доњи праг временског прозора првог упита. (Прилог 1)

2. Потом се креће са прикупљањем вести. У свакој итерацији се узима тренутно време currentTime, које се добија позивом функције getUTCTimeOffset(0), након чега се гради URL. (Прилог 2)
3. Позивом функције fetchNewsFromAPI(apiUrl) покреће се Python скрипта marketaux_fetcher.py која уз помоћ функције _popen [18] враћа JSON одговор са листом вести. У случају грешке, добија се изузетак који се у оквиру catch блока исписује у виду грешке. (Прилог 3)
4. Након успешног пријема одговора, резултујући JSON стринг се предаје функцији extractNewsFromResponse, унутар модула news_parser.cpp, у новој нити чиме се обезбеђује да главна нит не чека обраду већ наставља са следећим упитом. (Прилог 4)
5. Након тога, вредност previousTime се ажурира на currentTime, а програм чека cfg.delaySeconds секунди, пре покретања наредног циклуса. (Прилог 5)

Главне предности дизајна модула news_fetcher:

- Непрекидно и асинхроно преузимање: Главна функција newsPolling ради у посебној нити, што омогућава да програм истовремено прикупља нове вести и обрађује већ пристигле без блокирања главног тока извршавања.
- Неблокирајућа обрада одговора: По успешном добијању JSON стринга, покреће се нова нит која позива extractNewsFromResponse, што дозвољава да newsPolling одмах пређе на следећи упит без чекања на парсирање.
- Једноставна интеграција са конфигурацијама: Све кључне вредности се учитавају из settings.cfg преко loadSettings функције, што омогућава лако мењање понашања модула без измена у самом коду.

Модул news_parser

Модул news_parser.cpp обрађује сирову JSON листу вести враћену од стране MarketAux API-ја [1] и допуњује сваку ставку целокупним текстом чланка, након чега га додаје у newsQueue ради даље обраде. Рад модула покреће се у засебној нити из модула news_fetcher.cpp, тако да главна нит не чека на парсирање и екстракцију текста. Овим модулом се постиже поуздано издвајање и припрема текстуалних

садржаја за следећу фазу анализе.
Унутар extractNewsFromResponse функције:

1. Иницијално, позиција за претрагу у JSON стрингу се поставља на нулту вредност. Потом се уз помоћ функције response.find(...) лоцирају поља „title”, „url” и „published_at” за сваки чланак. Уколико нисмо лоцирали „title” поље, то знаћи да унутар JSON стринга немамо више не обрађених чланака. (Прилог 6)
2. Затим се за сваки URL проверава да ли се налази у сету seenUrls. Ако се URL већ налази унутар seenUrls сета, он је већ обрађен, и исти се прескаче. (Прилог 7)
3. За сваку нову ставку покреће се getArticleText унутар try блока да би се преко Python скрипте добио пун текст чланка. Унутар try блока, након успешног враћања текста, чланак се убацује у глобални ред уз mutex заштиту и URL се додаје у seenUrls. (Прилог 8)

Главне предности дизајна модула news_parser:

- Неблокирајуће парсирање: Обрада сваког одговора врши се у посебној нити, што омогућава newsFetcher-у да настави прикупљање нових вести без чекања.
- Изолација грешака: Појединачне неуспеле екстракције нису фаталне за целокупан систем. Грешке се логују, а парсер наставља да ради.
- Заузеће ресурса по потреби: Темпирање обраде и међусобно искључивање при приступу глобалним промењивима и структурама података обезбеђују поуздан и стабилан рад чак и при високим оптерећењима.

Модул news_processing

Модул news_processing.cpp обрађује већ парсиране и обогаћене чланке из newsQueue, идентификује најрелевантнију компанију поменуту у чланку, врши анализу sentiment-a и резултате убацује у companyStatusQueue за даљу обраду. Рад модула покреће се у засебној нити из главног фајла, тако да прикупљање, парсирање и обрада текстава теку паралелно.
Унутар processNewsArticles функције:

1. У бесконачној петљи се прво проверава да ли постоји не обрађени чланак унутар newsQueue-а уз mutex заштиту. Ако newQueue није празан, односно постоји не обрађени чланак, то обелезавамо помоћу hasNews промењиве и крећемо са обрадом. (Прилог 9)
2. Потом се обрада врши унутар try блока, ради хварања изузетака и њихове обраде, и у оквиру истог имамо два корака. Прво се позива фукнција detectCompanyInNews, која помоћу наслова и текста вести, враћа tuple који садржи companyName, stockSymbol и similarity. Потом се позива analyzeSentiment функција, која нам враћа sentimentScore. Сви резулати се ћувају у структури CompanyStatus и додају у companyStatusQueue. На крају се исписују информације о успешној обради, а унутар catch блока се логују грешке. (Прилог 10)

Главне предности дизајна модула news_processing:

- Неблокирајућа обрада: Обрада се одвија у засебној нити уз кратке и сигурне приступе глобалним промењивима, што омогућава осталим модулима да раде неометано.
- Јасна раздвојеност одговорности: Функције detectCompanyInNews, analyzeSentiment и processNewsArticles су логички издвојене, што олакшава тестирање, одржавање и евентуално проширење функционалности.
- Отпорност на грешке: Сваки појединачни позив Python скрипти је унутар try/catch блока, тако да неуспех у детекцији компаније у једном чланку не прекида обраду .

Помоћне скрипте у Python-у

У оквиру реализације система за анализу финансијских вести коришћене су четири главне Python скрипте, које C++ модули позивају преко системских позива [12]. Свака скрипта обезбеђује одређену функционалност и ради у сарадњи са конфигурационим параметрима из фајла settings.cfg. У наставку следи опис сваке скрипте, њене улоге у систему и кључна имплементациона решења.

1. Скрипта marketaux_fetcher.py:

- Примена: преузимање одговора са MarketAux API-ја [1] ради даље обраде од стране модула news_fetcher.cpp.
- Систем рада: скрипта користи библиотеку requests [14] за слање HTTP GET захтева на URL, који садржи базни API позив и временске параметре, из cfg фајла. Добијени JSON одговор се исписује на стандрадни излаз ради даље обраде [12].
- Грешке: у случају лошег позива скрипте, скрипта исписује упутство за коришћење и излази са кодом -2. У свим осталим ситуацијама, ако се током извршавања појави било каква грешка, скрипта исписује детаљан опис грешке и такође завршава рад са кодом -1.

2. Скрипта article_scraper.py:

- Примена: преузимање целокупних текстова чланака са дате URL адресе ради даље обраде од стране модула news_parser.cpp.
- Систем рада: скрипта уз помоћ библиотеке Newspaper3k [4] у функцији fetch_article(url) креира Article објекат, након чега започиње преузимање текста (article.download()) и парсирање (article.parse()), након чега екстрактовани текст исписује на стандардни излаз ради даље обраде [12].
- Грешке: у случају лошег позива скрипте , скрипта исписује упутство за коришћење и излази са кодом -2. У свим осталим ситуацијама, ако се током извршавања појави било каква грешка, скрипта исписује детаљан опис грешке и такође завршава рад са кодом -1.

3. Скрипта company_matcher.py:

- Примена: на основу наслова и текста чланка проналази најсличнију компанију ради даље обраде од стране модула company_matcher.
- Систем рада: након пријема аргумената title и text, функција match_companies позива embed_text функцију која генерише embedding позивом OpenAI API-ја [3] [19]. Затим добијени вектор прослеђује функцији collection.find која врши претрагу векторске базе података companiesdb помоћу AstraDB API-а [2] [8]. Добијени одговор се сортира и форматира, а резултат се исписује на стандардни излаз ради даље обраде [12].

- Грешке: у случају лошег позива скрипте , скрипта исписује упутство за коришћење и излази са кодом -2. У свим осталим ситуацијама, ако се током извршавања појави било каква грешка, скрипта исписује детаљан опис грешке и такође завршава рад са кодом -1.

4. Скрипта sentiment_analyzer.py:

- Примена: на основу наслова и текста чланка одређивање sentiment score ради даље обраде од стране модула company_matcher.
- Систем рада: скрипта спаја наслов и текст чланка у један улазни стринг и позива претходно истренирани HuggingFace [9] модел из библиотеке Transformers [9]. Резултат анализе је комбинација два параметра, label, који може бити POSITIVE, NEGATIVE или NEUTRAL, и представља тон текста, и score, који има вредности између 0 и 1, и представља сигурност модела у дати одговор. NEUTRAL се мапира на 0, POSITIVE на +score, а NEGATIVE на –score. На крају се бројчана вредност исписује на стандардни излаз ради даље обраде [12].
- Грешке: у случају лошег позива скрипте , скрипта исписује упутство за коришћење и излази са кодом -2. У свим осталим ситуацијама, ако се током извршавања појави било каква грешка, скрипта исписује детаљан опис грешке и такође завршава рад са кодом -1.

Све скрипте се налазе у директоријуму scripts и позивају се из одговарајућих C++ модула, функцијом _popen у Windows окружењу [12]. Параметри за API позиве дефинишу се централно у фајлу settings.cfg, што омогућава флексибилна подешавања без промена у изворном коду. Овако дизајнирана помоћна инфраструктура омогућава раздвајање одговорности, лаку замену модела и библиотека и једноставно отклањање грешака у појединачним скриптама.

Тестирање и резултати

У овом поглављу описујемо поступак тестирања развијеног система и представљамо кључне резултате добијене кроз симулације инвестиционих стратегија. Тестирање је реализовано на скупу података прикупљеном у временском периоду од 5. до 9. маја 2025. године, током којег је систем преузео и обрађидио укупно 6096 финансијских вести из различитих извора. Уз вести, прикупљене су и одговарајуће цене акција свих компанија на National Association of Securities Dealers Automated Quotations (NASDAQ) и The New York Stock Exchange (NYSE) берзама са интервалима од једног минута [15] [20].

Након фазе прикупљања и обраде улазних података, за тестирање инвестиционе стратегије коришћена је C++ апликација, simulate_investment.cpp, која симулира куповину и продају на основу прага сентимента и сличности између вести и компанија. За сваку симулацију meren је укупан број трансакција, број профитабилних трансакција и коначну вредност портфолија. Резултати се приказују кроз основне статистичке показатеље као што су: број трансакција, проценат профитабилних и просечан повраћај, те омогућавају упоредну анализу различитих параметара.

У наставку ћемо прво дефинисати испитне сценарије и критеријуме за одабир параметара, затим приказати репрезентативне резултате симулација, а на крају дискутовати о добијеним резулататима и потенцијалним унапређењима.

Тестни сценарији

У оквиру тестирања развијеног система дефинисани су испитни сценарији кроз комбинације четири кључна параметра стратегије, уз фиксно време чувања позиције од 24 сата (86400 секунди). Параметри и њихови домени су следећи:

- Учешће портфолија (investPercent): Од 0.5% до 5% у корацима од 0.5%, представља проценат портфолија који инвестирамо при свакој куповини.

- Праг за куповину (buyThreshold): Од 0.8 до 0.9 у корацима од 0.05, предстаља минималан sentiment score потребан да би дошло до куповине.
- Праг за продају (sellThreshold): од –0.5 до –0.1 у корацима од 0.1, представља вредност испод које се продају све позиције одређене компаније.
- Праг сличности (similarityThreshold): од 0.5 до 0.65 у корацима од 0.05, представља минималан similarity score потребан да би дошло до куповине.

За све комбинације ових параметара покренуто је око 600 симулација, уз филтрирање непотпуних због недостатка података, а метрике које су праћене укључују:

- Укупан број трансакција (totalTrades).
- Број профитабилних трансакција (profitableTrades).
- Тачност алгоритма (accuracy = profitableTrades/totalTrades).
- Коначну вредност портфолија (Final balance).
- Очекивани повраћај у дужем временском периоуд (expectedReturn = profitableTrades·medianWin + (totalTrades–profitableTrades)·medianLoss).

Сви резултати симулација за поједине конфигурације налазе се у simulation_result.log фајлу. Урађена је и додатна класификација према томе да ли се као критеријум оптимизације узима максималан крајњи баланс или максималан очекивани повраћај, при чему се за сваку од ових категорија идентификују најбољи параметри.

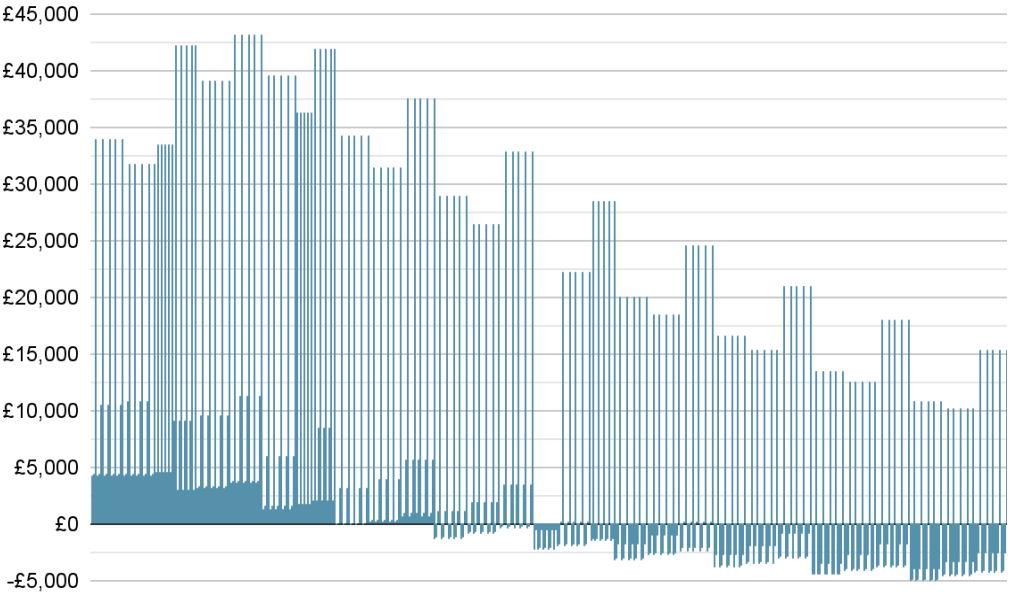
Резултати тестирања

Sažetak ključnih pokazatelja:

- Стратегија са највећом зарадом:
  invest 1 %, buy ≥ 0.90, sell ≤ –0.50, similarity ≥ 0.65
  Final balance: $53213.50
- Стратегија са највећом очекиваном зарадом:
  invest 5%, buy ≥ 0.90, sell ≤ –0.50, similarity ≥ 0.60
  Final balance: $7409.97 | Expected return: 172.242
- Број симулација: 600
- Број профитабилних симулација: 355

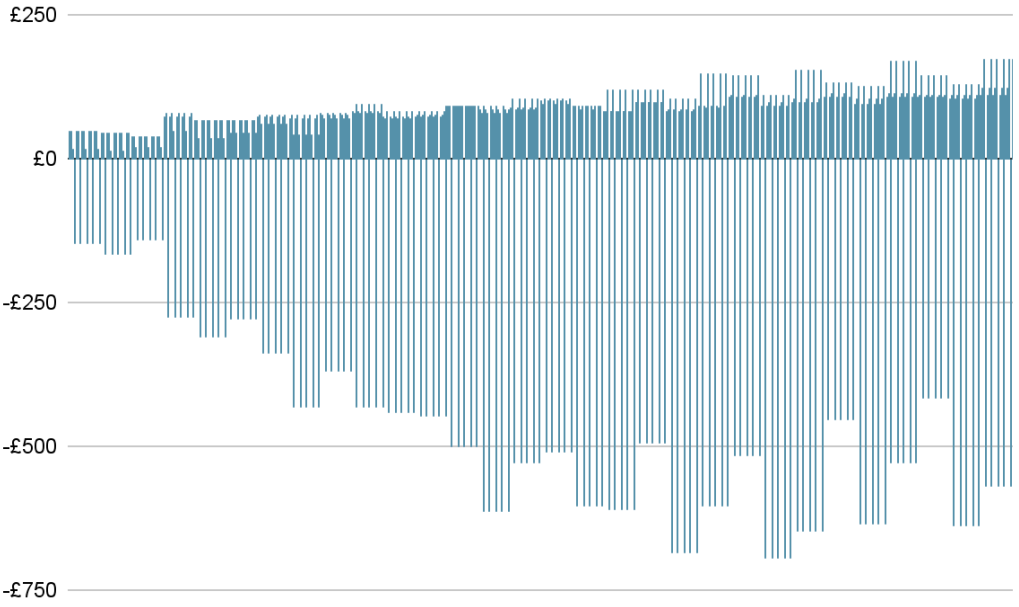
Графикони:

1. Крајна добит:



Овај графикон показије коначну добит за сваку од симулација поређану по редоследу покретања симулације.

2. Очекивана добит:



Овај графикон приказује очекивану добит за сваку од симулација поређану по редоследу покретања симулације.

Анализа резултата

У оквиру анализе резултата размотрићемо значајна запажања добијена из репрезентативних симулација, оцену поузданости стратегије и идентификацију кључних фактора који утичу на перформансе. Циљ је да извршимо анализу података и истакнемо ограничења тестирања и предложимо правац будућих оптимизација.

1. Стопа профитабилности и дистрибуција добитака: Од укупно 600 симулација, 355, односно 59%, је било профитабилно, што указује на то да стратегија углавном успева да ухвати добитне сигнале. Међутим, детаљна анализа показује да су највећи утицај на коначни резултат имале појединачне, екстремно профитабилне трансакције, док је већина трансакција имала мали утицај. Ово указује на асиметричну дистрибуцију зараде која садржи елемент високог ризика.
2. Ниска прецизност: Ниједна симулација није прешла прецизност од 48%, што је испод 50%, односно случајног погађања. То значи да би, чисто статистички, насумични улаз и излаз дали бољи проценат поготка. Међутим, мали просечни губитци (median loss) и неколико великих добитака у неким симулацијама оправдавају употребу ове стратегије у смислу коначног профита.
3. Утицај количине учешћа: Податак да су вредности investPercent од 1%, 1.5% или 2%, доносиле најпоузданије резултате, без иједне негативне симулације, указује на повољан однос ризика и добити за мале улоге. Са повећањем investPercent-а, расте волатилност и број губитних сценарија.
4. Ограничења узорка и стања маркета: Тестирани период од само пет дана, од 5. до 9. маја 2025. године, и свега 600 симулација представљају релативно мали узорак. Тржишна флуктуација од ±1.2 % у индексима попуд S&P 500-а [21] током тог периода показује да је тржиште било релативно стабилно. За поузданије закључке потребно је проширити временски оквир и обухватити периоде повећане али и мање волатилности.
5. Корисност метрике очекиваног повраћаја: График очекиваног повраћаја није дао јасан увид у оптималне параметре због јаког утицаја екстремних вредности. Међутим могуће је да би за дуже временске периоде, ова метрика могла да буде од користи.

Закључак

У овом раду развијен је и тестирана имплементација система за аутоматску обраду и анализу сентимента финансијских вести, која се састоји из модула за преузимање, парсирање и обраду вести, као и скрипти за екстракцију текста, векторизацију, идентификацију компанија и финалну анализу сентимента. Резултати практичног дела показују да систем прецизно издваја релевантне чланке и доследно одређује тон садржаја, што потврђује да су коришћени модели и дизајн погодни за семантичку обраду у реалном времену.

Иако је анализа вести поуздана, примењена инвестициона стратегија са унапред дефинисаним праговима за куповину и продају није дала оптималне резултате у смислу очекиваног приноса. У раду је у дискусији указано на бројне факторе који утичу на ову појаву, те је могуће да се уз додатна тестирања можемо можемо приближити жељеним резултатима.

Као најзначајније смернице за даља унапређења система издвајају се:

1. Прелазак на локална решења: Коришћење локалних решења, за векторску базу података и векторизацију текста, уместо cloud сервиса смањило би се кашњење упита и повећала би се скалабилност.
2. Интеграција адаптивне инвестиционе стратегије: Уместо коришћења фиксних праговних вредности, можемо развити модел, заснован на машинском учењу, који би динамички подешавао параметре куповине и продаје на основу актуелних тржишних трендова, као и волатилности и анализе историјских података, чиме би се остварила виша реактивност система и бољи потенцијални приноси.

Ови кораци представљају природан ток за будућа истраживања и практичну примену система у стварним трговачким окружењима.

Прилози

Прилог 1:

```
// Get the initial start time for the first polling window
string previousTime = getUTCTimeOffset(cfg.lookbackSeconds);
```

Прилог 2:

```
// Start polling loop
while(true)
{
    // Get current time for the upper bound of the query window
    string currentTime = getUTCTimeOffset(0);

    // Construct API request URL with published_after filter
    string apiUrl = cfg.baseApi
              + "&published_after=" + previousTime
              + "&published_before=" + currentTime;
```

Прилог 3:

```
// Print log with the time window being requested
safeCout("[INFO] ", "Requesting news published between " + previousTime + "
and " + currentTime + "\n\n");
try
{
    // Fetch MarketAux news for the given time window
    string response = fetchNewsFromAPI(apiUrl);
    ...
}
catch(const exception& e)
{
```

```
    // Handle errors
    safeCerr("[Error] ", string(e.what()) + "\n");
}
```

Прилог 4:

```
// Extract news items from the raw JSON string and add them to the global
queue (newsQueue)
// Process the response in a new thread so it's non-blocking
thread processorThread(extractNewsFromResponse, response);

// We don't wait for it to finish
processorThread.detach();
```

Прилог 5:

```
// Update the starting point for the next request window
previousTime = currentTime;

// Wait before making the next request
safeCout("[INFO] ", "Waiting " + to_string(cfg.delaySeconds) + "seconds
before next request..." + "\n\n");

// Pause execution for the configured delay time
this_thread::sleep_for(seconds(cfg.delaySeconds));
```

Прилог 6:

```
size_t pos = 0;
while(true)
{
```

```
        // Find the start of a new article by locating the next "title" field
    pos = response.find("\"title\":", pos);

    // Exiting the loop if there are no more articles found in the response
    if(pos == string::npos)
    {
        safeCout("[INFO] ", "No more articles to extract from JSON
response.\n\n");
        break;
    }

    NewsItem item;

    // Extract title
    size_t start = response.find("\"", pos + 8) + 1;
    size_t end = response.find("\"", start);
    item.title = response.substr(start, end - start);
    pos = end;

    // Extract url
    pos = response.find("\"url\":", pos);
    if(pos == string::npos)
    {
        break;
    }
    start = response.find("\"", pos + 6) + 1;
    end = response.find("\"", start);
    item.url = response.substr(start, end - start);
    pos = end;

    // Extract published_at
    pos = response.find("\"published_at\":", pos);
    if(pos == string::npos)
    {
        break;
    }
    start = response.find("\"", pos + 16) + 1;
    end = response.find("\"", start);
```

```
    item.publishedAt = response.substr(start, end - start);
    pos = end;
    ...
```

Прилог 7:

```
// Skip duplicates if we've already seen this URL
if(seenUrls.count(item.url))
{
    safeCout("[INFO] ", "Skipping duplicate: " + item.url + "\n\n");
    continue;
}
```

Прилог 8:

```
// Try to fetch full article text from the URL using article_scraper.py
try
{
    // Get the full article text by calling the Python script
    item.text = getArticleText(item.url);

    // Push the item into the global queue
    {
        lock_guard<mutex> lock(newsQueueMutex);
        newsQueue.push(item);
    }

    // Marks an item as seen
    seenUrls.insert(item.url);

    // Confirm successful processing
    safeCout("[INFO] ", "News article from URL: " + item.url + "\n -
successfully processed and added to queue.\n\n");
```

```
}
catch(const exception& e)
{
    // Print any error that occurred
    safeCerr("[WARN] ", "News article from URL: " + item.url + "\n - failed
to be processed and was not added to queue.\n");
    safeCerr("[ERROR] ", string(e.what()) + "\n\n");  // print the specific
error
}
```

Прилог 9:

```
while(true)
{
    // Check if there are news articles in the queue to process
    NewsItem currentNews;
    bool hasNews = false;

    {
        lock_guard<mutex> lock(newsQueueMutex);
        if(!newsQueue.empty())
        {
            // Get the oldest item from the queue (FIFO order)
            currentNews = newsQueue.front();  // Remove the processed article
from the queue
            newsQueue.pop();
            hasNews = true;
        }
    }

    if(hasNews)
    {
    ...
```

Прилог 10:

```
try
{
    // Extract the title and text from the news item
    string title = currentNews.title;
    string text = currentNews.text;

    // Detect a company in the article using the detectCompanyInNews function
    tuple<string, string, double> company = detectCompanyInNews(title, text);

    string companyName = get<0>(company);
    string stockSymbol = get<1>(company);
    double similarity = get<2>(company);

    // Create a CompanyStatus object with the results
    CompanyStatus status;
    status.companyName = companyName;
    status.stockSymbol = stockSymbol;
    status.similarity = similarity;
    status.sentimentScore = analyzeSentiment(title, text);   // Perform
sentiment analysis for the article
    status.timeAdded = currentNews.publishedAt;         // Add the timestamp

    // Add the CompanyStatus to the companyStatusQueue for further processing
    {
    lock_guard<mutex> lock(companyStatusQueueMutex);
    companyStatusQueue.push(status);
    }

    // Confirm successful processing
    safeCout("[INFO] ", "Sentiment for " + companyName + " (" + stockSymbol +
") from url " + currentNews.url + "\n - successfully processed and added to
queue.\n\n");
}
catch (const exception& e)
{
    // Handle errors during sentiment analysis
    safeCerr("[Error] ", string(e.what()) + "\n");
}
```

Литература

[1] Marketaux. *Marketaux API Documentation* (https://www.marketaux.com/documentation) прегледано 27. 5. 2025.

[2] DataStax. *AstraDB Data API & Vector Search* (https://docs.datastax.com/en/astra-serverless) прегледано 27. 5. 2025.

[3] OpenAI. *Embeddings API* (https://platform.openai.com/docs/guides/embeddings) прегледано 27. 5. 2025.

[4] L. Lelane. *Newspaper3k Documentation* (https://newspaper.readthedocs.io/) прегледано 27. 5. 2025.

[5] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space*. Advances in Neural Information Processing Systems, 26, 3111–3119.

[6] Han, Y., Liu, C. & Wang, P. (2023). *A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge*. arXiv:2310.11703.

[7] Pang, B. & Lee, L. (2008). *Opinion Mining and Sentiment Analysis*. Foundations and Trends® in Information Retrieval, 2(1-2), 1–135.

[8] DataStax Labs. *astrapy: Astra DB Data API Client* (https://github.com/datastax-labs/astrapy) прегледано 27. 5. 2025.

[9] Hugging Face. *Transformers Documentation* (https://huggingface.co/docs/transformers/) прегледано 27. 5. 2025.

[10] Herlihy, M. & Shavit, N. (2011). *The Art of Multiprocessor Programming*, Chapter 2 (Shared-Memory Programming). Elsevier.

[11] Sommerville, I. (2015). *Software Engineering* (десето издање). Pearson.

[12] R. Stevens, S. Rago. *Advanced Programming in the UNIX® Environment*, 3. издање, Addison–Wesley, 2013. (Поглавље 13: Interprocess Communication: Pipes and FIFOs)

[13] Kitware. CMake Documentation (https://cmake.org/documentation/) прегледано 27. 5. 2025.

[14] K. Reitz. *Requests: HTTP for Humans* (https://docs.python-requests.org/) прегледано 27. 5. 2025.

[15] Polygon.io. *Stocks API Reference* (https://polygon.io/docs) прегледано 27. 5. 2025.

[16] cppreference.com. std::thread (https://en.cppreference.com/w/cpp/thread/thread) прегледано 27. 5. 2025.

[17] cppreference.com. std::mutex (https://en.cppreference.com/w/cpp/thread/mutex) прегледано 27. 5. 2025.

[18] cppreference.com. popen (https://en.cppreference.com/w/cpp/io/c/popen) прегледано 27. 5. 2025.

[19] OpenAI. *openai-python* (https://github.com/openai/openai-python) прегледано 27. 5. 2025.

[20] Nasdaq. Nasdaq *Stock Screener* (https://www.nasdaq.com/market-activity/stocks/screener) прегледано 27. 5. 2025.

[21] Yahoo Finance. S&P 500 (^GSPC) Chart (https://finance.yahoo.com/chart/%5EGSPC) прегледано 27. 5. 2025.