CrowdAlpha

Harnessing Social Sentiment to Outperform the Market

BSC Computer sciences final project submission
Yaniv Gutman, 204636351
June 2025
Afeka college of engineering

https://github.com/yanivgu/crowdalpha

Contents

Executive summary

The CrowdAlpha project set out to develop an algorithmic trading system that leverages user-generated content from a social trading platform to build a portfolio based on sentiment analysis. The project's primary objective was to create a profitable and robust model that exceeds the performance of a market benchmark (S&P 500) by at least 2%, through innovative integration of sentiment signals and user credibility metrics.

The project was structured in two phases: a Proof of Concept (POC) to validate feasibility, followed by a full-scale system design for production use.

The POC results were promising: a total simulated gain of 21.8% over a 12-month period compared to 12% for the S&P 500, with lower drawdowns and more consistent growth. The best strategy was simple yet effective: equal capital allocation to the top 10 sentiment-ranked stocks.

The engineering methodology involved collecting and filtering social media posts containing stock mentions, applying sentiment analysis using Azure AI's LLMs, and weighting those sentiments according to user-specific performance metrics (e.g., 2-year gains, platform experience). Sentiment scores were aggregated to generate daily trading signals, optimized through an unsupervised machine learning algorithm (Differential Evolution) that maximized portfolio gain over time.

The full-scale implementation proposes a real-time system leveraging Azure cloud technologies such as Azure Kubernetes Service (AKS), Azure Functions, and SQL Database. It supports semi real-time processing of social sentiment and signal generation, feeding directly into the trading engine for execution. This cloud-native architecture ensures high scalability, modularity, and streamlined integration across components.

Conclusions indicate the potential of sentiment-based trading with credibility-weighted signals. However, challenges remain, including the limited language scope (only English), lack of overnight holding, and the need for more sophisticated risk modeling. Future work should explore multilingual sentiment analysis, dynamic risk-aware strategies, holding-period enhancements, and improved compute efficiency using SparkML and Azure AI bulk processing.

Overall, CrowdAlpha demonstrates strong potential to serve as a scalable and profitable foundation for a next-generation trading algorithm driven by collective intelligence.

Project Overview and Goal Definition

Description

Project Goal: Develop a stock trading algorithm that constructs a trading portfolio based on user social media posts, on a trade-focused social media platform.
The algorithm will analyze the sentiments expressed in the media and integrate the sentiment analysis into a weighting mechanism to make trading decisions.

The algorithm will perform sentiment analysis on posts within the social network, identifying the stocks mentioned and evaluating the sentiment expressed for each stock individually. Additionally, the system will assign a relative weight to the sentiment analysis output based on the profile of the author, which includes factors such as experience in trading, past performance, and more.

The system will consist of three main components:

1. Social media monitoring, analyzing posts and storing the resulting data, including author characteristics.
2. Model construction based on the sentiment analysis output, author attributes, and integration of external parameters.
3. Back-testing the algorithm's performance based on social media information and historical stock prices.

Goals

1. **Objective:** Develop a Proof of Concept (POC) for a profitable trading algorithm based on crowd wisdom, using social media sentiment as a core signal.
2. **Project Purpose:** This POC is exploratory in nature and aims to evaluate the technical feasibility and performance potential of the proposed algorithm. If the results meet or exceed defined performance benchmarks, a decision will be made on whether to proceed with full-scale development.
3. **Project Targets**:
    a. Achieve an average annual return of at least 2% above the market benchmark during the testing period. The benchmark index used for comparison will be the S&P 500.
4. **Project Metrics**:
    a. Portfolio return generated by the algorithm compared to the return of the benchmark index – S&P 500.

Architecture

This section outlines the system components and their interactions.
Since the project is a POC project, this section will outline the architecture used for the POC, as well as proposed architecture for the full implementation of the solution, relying on principles laid by the POC, allowing for easy transition from the POC to the full solution.

Since the project is done for a company working with Microsoft Azure as its cloud service, the technologies used are solutions provided by Azure infrastructure, but have equivalents in other cloud providers, or self-hosted solutions.

POC Architecture

The architecture used for the POC focuses on gathering the relevant data for the machine learning process, requiring agility in adding and modifying data points, as well as running complex logics and building extensive data models for the machine learning process to run on, to be able to achieve the best results.

Furthermore, to reduce the wasted effort in the POC phase and be able to transition from the POC to the final architecture in case of a valid model, some of the components in the POC phase will be developed in a way that they can be re-used in some capacity in the full implementation.

General architecture

Below is a diagram depicting the general script-based architecture of the POC phase, describing the data flow from the sources, intermediate processing, and finally the ML process.

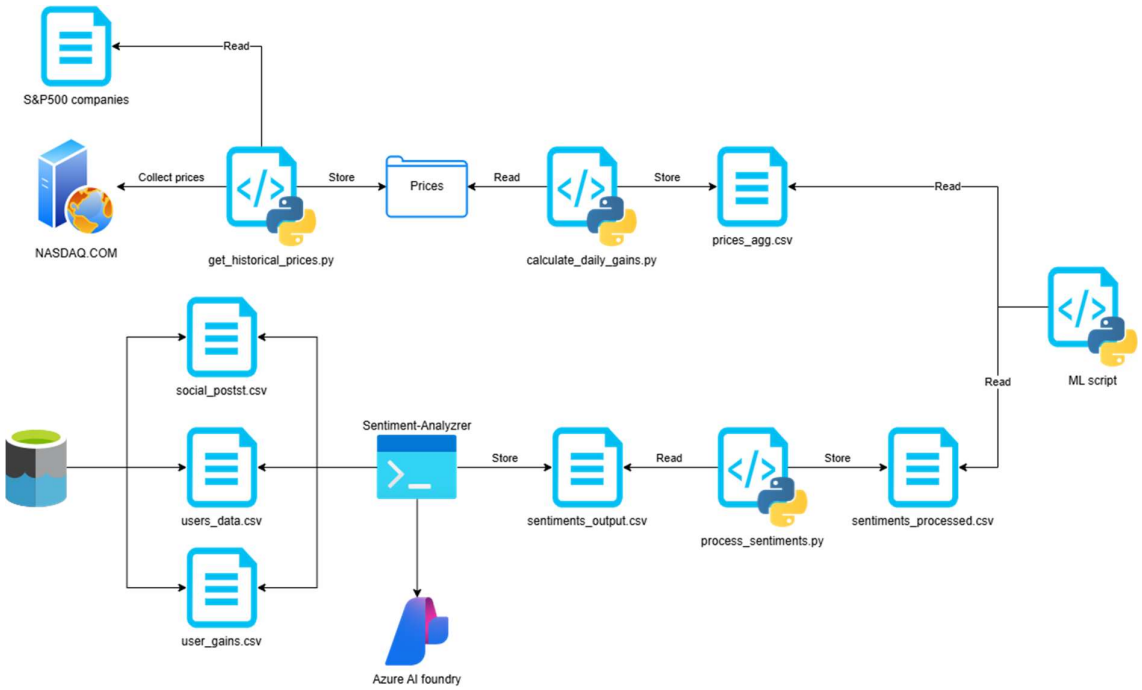

Figure 1: General architecture of the POC phase

Components

Historical prices & Price aggregation

Pair of scripts responsible for collecting and aggregating stock price data to be used by the machine learning process to evaluate performance of the models.

Responsibilities

- Collect historical open and close prices for specific stock symbols, over a required period.
- Aggregate the data into a single file containing trade date, symbol, and daily gain of that symbol in percentage.

Process walkthrough

The initial script is invoked manually with the following parameters:

- Symbols file path
- Output directory
- Start date
- End date

The process then reads the symbols file as an input, and calls NASDAQ API per symbol in order to retrieve historical open and close prices. Per symbol the prices are done stored in csv files in the output directory.

Once the prices are available, a second script reads all the prices from the output directory and aggregates them into a single file ordered by date, that contains the data for all the symbols. Additionally the script calculates the daily gain of each stock symbols and adds it to the file so it can be used to calculate performance over time.

Sentiments and user processing

The sentiments and user processing is a dotnet console application, that processes the social posts data as well as the user data, performs sentiment analysis using Azure AI Foundry, and builds the sentiment results that may be used by the model.

The reason this process is in dotnet and not python, is due to the reason it may be reused in the full implementation architecture as code to perform the sentiment analysis may be used once, while only the running process will need to be re-written into a cloud-hosted service.

Responsibilities

- Read files containing social media data, user data, and user performance data
- Filter relevant social media posts
- Perform sentiment analysis using Azure AI Foundry
- Match social media to the relevant user data, as well as performance data

Process walkthrough

The application has configuration for connectivity to Azure AI Foundry, as well as configuration for the source and output files, and other relevant parameters.

The process starts by reading the user data and user performance data files so they may be used immediately on demand. Once the data is loaded, the process starts reading the social media posts file asynchronously, simultaneously reading more rows from the file as well as

processing it, and eventually writing the data to file asynchronous. The reason the processing is asynchronous is because the social media file is large and to reduce requirement for a lot of memory, and avoiding possibility of process crashing and losing all processed data.

All of the social media posts are initially filtered by the process, to only posts that contain mentions of the relevant symbols – the symbols that compose the S&P 500 index.
Posts that do not contain at least one symbol are discarded, and the valid posts continue for the sentiment analysis process.

The sentiment analysis process uses Azure AI to analyze the sentiments in the content towards the relevant symbols, and respond with numeric sentiment to each symbol mentioned. Since the processing for the sentiment analysis takes time, it is paralleled to increase performance and reduce run time.

Once the sentiments are available per symbol, the data is filtered again to contain only the relevant symbols in check, matched with the relevant use data and user performance data, and eventually saved to a new csv file with the processed sentiments data and relevant metadata on the user.

Data model preparation

The data model preparation is a dedicated step to aggregate the data and build a flat data model that the machine learning process may use to learn and evaluate the best model that fits the requirements.

Full implementation architecture

The architecture requirements for the production system is different than the POC architecture, since the POC focuses on building and evaluating a trading model using machine learning resulting in a parameterized model to use in the production system.

The architecture here takes the model output from the POC phase, and implements it in a way that the relevant input data flows into the model, the data is then evaluated, and decisions are made based on the results.

General architecture

Below is a diagram depicting a proposed production architecture of the full implementation of the process, describing the data flow from the sources, intermediate processing, and finally the execution based on the resulting model.

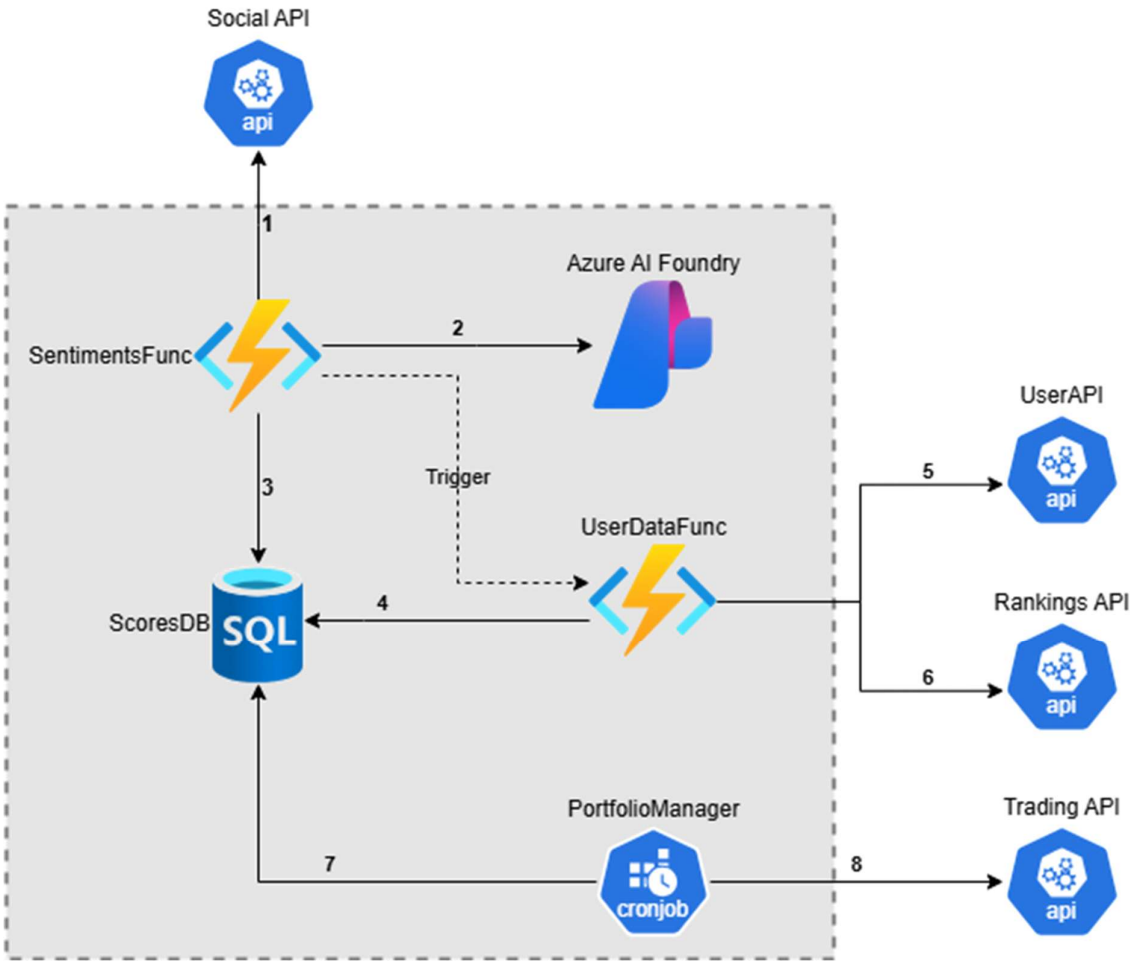

Figure 2: General architecture for full implementation of the solution

External APIs

This section describes the external APIs utilized by the system, which serve as critical sources of both input data and output execution channels. Leveraging these existing APIs—which are outside the project's scope for development—allows the system to efficiently access diverse data types, such as social media content, user profiles, and financial performance metrics, and execute trading orders directly in the market. Effective integration of these existing APIs is fundamental for the real-time responsiveness and accuracy of the sentiment-based trading algorithm, enabling the system to act swiftly and reliably on market insights and user-generated signals.

Social API

Responsible for providing details on social media posts in social trading platform, including post content, attachments, time, and owner data.

The API is used by SentimentsFunc process, to retrieve recent social media data as the initial data source and process it into the system.

User API

Responsible for providing static user data and attributes, such as profile info, experience, certificates, and more.
Additionally, the API also returns user privacy information – indication if the user allowed for his data to be shared (public) or not (private) – critical information as the algorithm will run only on users that opted to share their information (public).

The API is used by UserDataFunc to collect and aggregate user data to be matched with the sentiment scores, and used for user credibility for sentiment weighting

Rankings API

Responsible for providing user performance metrics – gain over multiple timeframes, max drawdown (portfolio peak-to-valley decline, indication of risk taken), and risk score (calculated metric on the risk the user is taking in his activity).

The API is used by UserDataFunc to collect and aggregate user performance to be matched with the sentiment scores and used for user credibility for sentiment weighting.

Trading API

API responsible for exposing ability to send trade order to the market, to buy or sell relevant instruments.

The API is used by PortfolioManager to act on algorithm recommendations and apply the necessary changes in the algorithm portfolio.

Internal components

This section outlines the internal processes that need to be developed for the full-scale implementation of the solution. These components handle critical functions such as data retrieval, analysis, and execution. Each component has defined responsibilities, data inputs, and outputs, providing clarity and efficiency in workflow integration.

SentimentsFunc (Azure Function)

This is the main component responsible for retrieving social media content and performing sentiment analysis on the data, and creating an output table of the sentiment of the content creator toward the symbols mentioned in the post.

Responsibilities

- Triggered periodically by a CRON scheduler.
- Fetch recent social posts using the Social API.
- Filter posts to include only those with valid symbol mentions.
- Perform sentiment analysis on each post using Azure AI Foundry.
- Store the results and metadata in ScoresDB.
- Trigger UserDataFunc to initiate user enrichment.

Sequence diagram

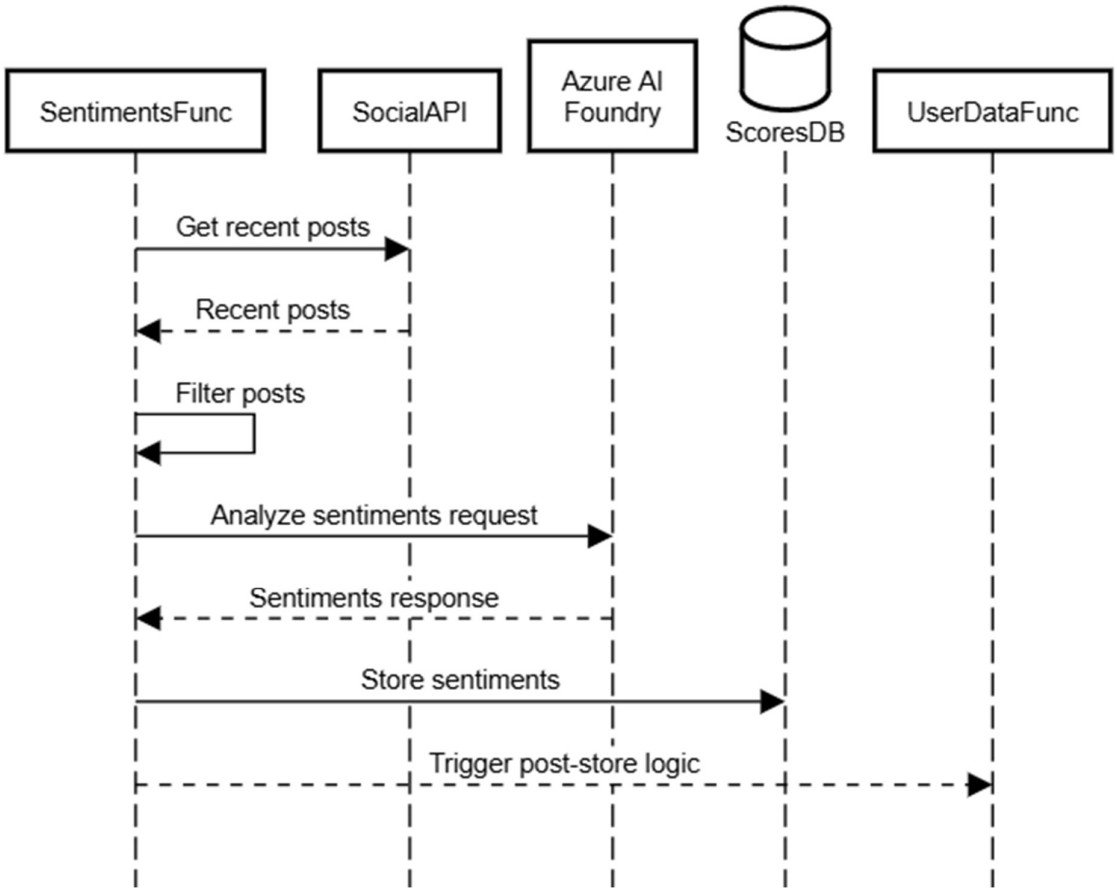

Figure 3: SentimentsFunc processing flow - from input to output

Process walkthrough

The process is an Azure Function triggered by a CRON schedule at fixed intervals (e.g., hourly). Upon execution, it retrieves all social posts from the Social API that were created since the last run.

Next, the function filters out posts that do not mention any relevant stock symbols (from a predefined list). Posts without valid symbols are discarded.

The remaining posts are then processed using Azure AI Foundry. Each post is sent to the language model with a fixed system prompt to ensure consistent output. The model returns a list of symbols mentioned and the sentiment toward each symbol a numeric value in a configurable range between −{maxSentimentScore} and +{maxSentimentScore}.

Post-processing is applied to ensure only valid symbols remain. Since each post includes at least one valid symbol, all posts passing the initial filter are retained for sentiment output.

The sentiment results, along with post metadata (e.g., author, timestamp, symbols), are then persisted to ScoresDB.

Finally, once all data is saved, the function triggers the UserDataFunc process to begin user-level data enrichment for the identified authors.

UserDataFunc (Azure Function):

This component is responsible for enriching the sentiment analysis output with relevant user information to enable credibility-weighted scoring. The user-level data helps assess the trustworthiness and expertise of each content author, which is used in the next step of symbol scoring and portfolio construction.

Responsibilities

- Triggered by SentimentsFunc upon completion of sentiment analysis.
- Identify relevant users who posted content in a configurable lookback period (default: 1 year).
- Retrieve user metadata and performance metrics in bulk using User API and Rankings API.
- Upsert user information into ScoresDB: insert new records or update existing ones.
- Ensure stale users are not updated and excluded from scoring.

Sequence diagram

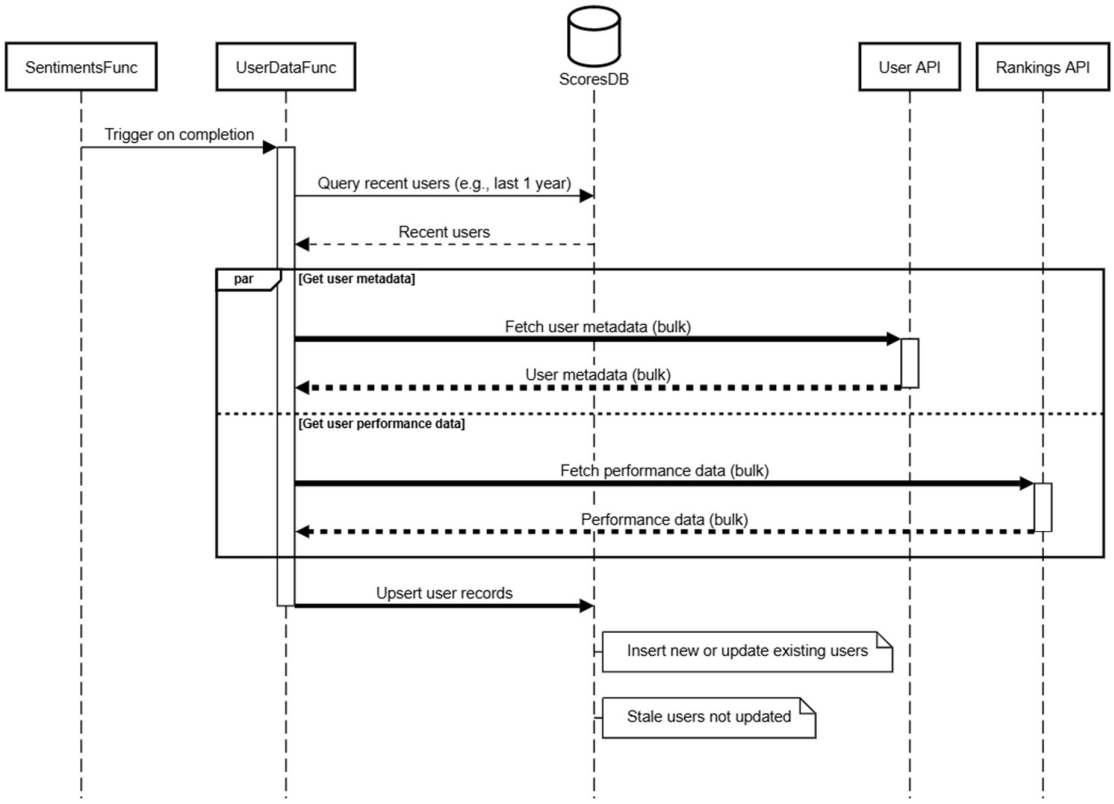

Figure 4: UserDataFunc processing flow

Process walkthrough

The process is an Azure Function triggered by SentimentsFunc upon completion of the sentiment processing phase. It begins by querying ScoresDB for all unique users who have created content within a configurable time window (default: 1 year).

Next, it performs two batch operations:

- It calls the **User API** to fetch static user data, including experience, credentials, and public/private profile status.
- Simultaneously, it calls the **Rankings API** to retrieve user performance metrics, including returns over various timeframes, drawdown, and risk score.

Once data is retrieved, the function processes the user records:

- If a user already exists in the ScoresDB, their record is updated.
- If not, a new user record is inserted.
- Users not active within the time window are skipped to avoid unnecessary updates.

Finally, all processed data is persisted in ScoresDB. This data serves as input for credibility scoring used later in symbol score calculations, allowing the system to weight sentiment more accurately based on the reliability and history of the content creators. in transforming raw sentiment into a trust-weighted signal, ensuring that more reliable contributors influence portfolio decisions more heavily.

PortfolioManager (Cron Job):

This component is responsible for aggregating sentiment data, applying author credibility scores, calculating symbol-level scores, and executing portfolio rebalancing based on the generated signals. It is designed as a Kubernetes-based cron job (AKS) due to its longer execution time and the need to sequence trading orders carefully.

Responsibilities

- Triggered daily via an AKS cron job.
- Query ScoresDB to retrieve post-level sentiment and associated user scores.
- Aggregate sentiment per symbol using author credibility weighting.
- Generate portfolio signals and determine asset-level actions.
- Execute trading decisions using the Trading API.
- Ensure sequential trade handling (e.g., wait for close confirmation before opening a new position).

Sequence diagram

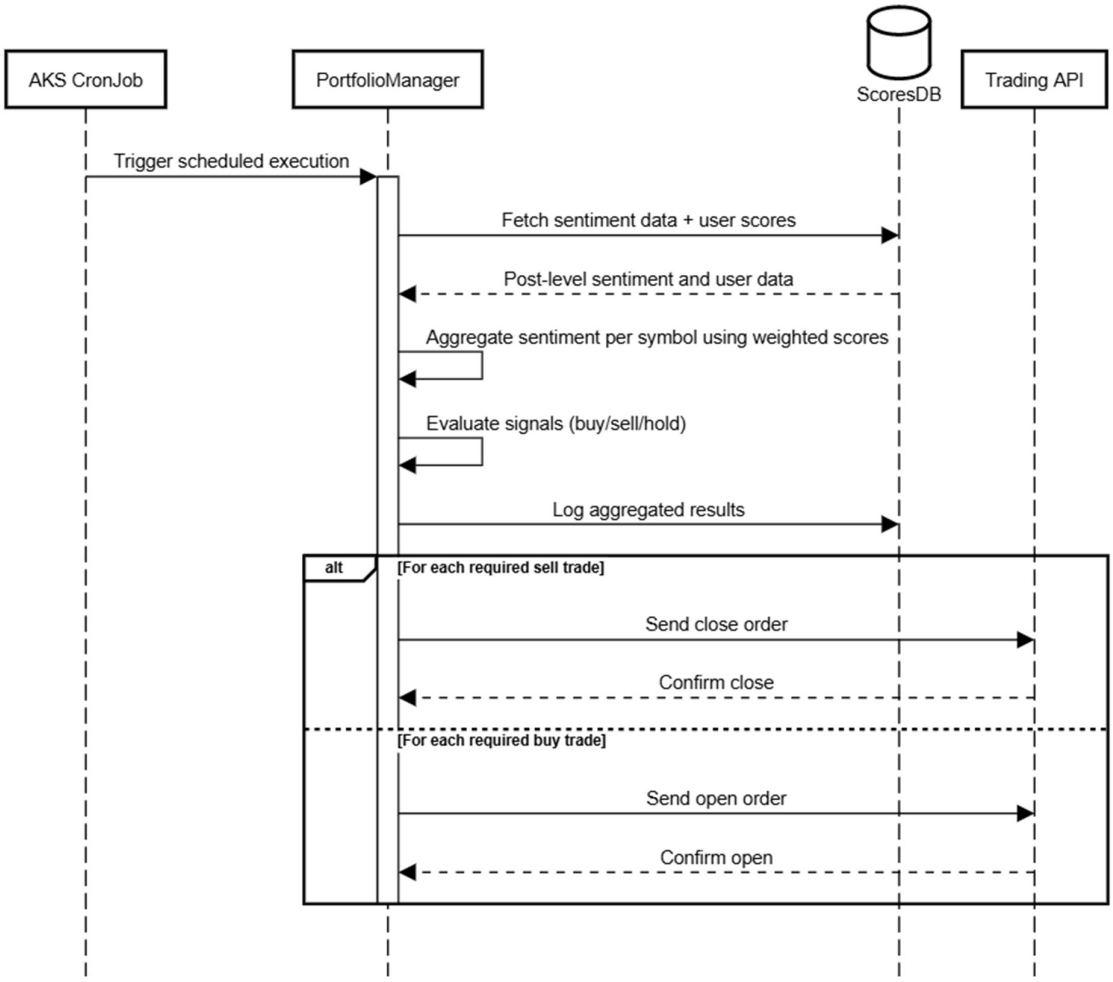

Figure 5: PortfolioManager processing flow

Process walkthrough

The process is implemented as an AKS-hosted cron job triggered periodically (daily). Once triggered, it queries ScoresDB to retrieve all post-level sentiment entries along with their associated user credibility scores.

For each symbol, the process aggregates sentiment scores using a weighted average, where the weight is based on the author's credibility score. This results in a single net sentiment score per symbol for the given evaluation window.

Next, the component evaluates which symbols to buy, sell, or hold based on a set of pre-defined thresholds or scoring logic. It then compares the resulting signal portfolio to the current portfolio state.

Trading operations are executed using the Trading API. The system enforces strict sequencing: it must wait for all 'close' orders to be confirmed before issuing any new 'open' orders. This ensures clean position transitions and avoids over-exposure.

The outcome of each execution is logged and persisted for performance tracking and audit purposes. PortfolioManager thus serves as the execution engine that turns weighted sentiment into actual market exposure through the Trading API.

Infrastructure and Technologies

The system architecture is designed around scalable, modular components that align with the project’s goals of data-driven trading decisions and operational efficiency. The infrastructure leverages a combination of serverless functions, containerized workloads, cloud-native machine learning, and a centralized relational database to support end-to-end functionality from data ingestion through execution.

Azure AI Foundry

Azure AI Foundry is a managed platform within Microsoft Azure that provides access to powerful large language models (LLMs) via APIs. It allows the system to perform natural language processing tasks—such as sentiment analysis—by submitting text-based prompts to models like GPT. In this project, the Foundry is used to analyze user posts and assign sentiment scores toward mentioned stock symbols.

The platform supports both single and batch processing modes, enabling scalable and cost-efficient execution. Its tight integration with Azure's security infrastructure ensures safe handling of data and managed access, making it suitable for production-grade enterprise use.

ScoresDB (SQL Database)

ScoresDB is a centralized relational database (e.g., Azure SQL or PostgreSQL) used to store all structured outputs from sentiment and user analysis. It serves as the main data exchange layer between SentimentsFunc, UserDataFunc, and PortfolioManager.

It supports efficient upserts (update if exists, insert if new) and filtering based on time or user context. The table schema includes sentiment scores, timestamps, user IDs, symbol references, and derived credibility metrics—essential for computing weighted sentiment signals.

Azure Functions (Runtime Environment)

Azure Functions is a serverless computing platform ideal for lightweight, event-driven tasks. It is used for **SentimentsFunc** and **UserDataFunc**, which perform short, modular jobs triggered by CRON schedules or completion events.

The platform offers built-in scalability, monitoring, and fault handling, allowing these microservices to run independently and efficiently without managing underlying infrastructure. Its low operational overhead makes it a practical fit for both prototyping and production.

AKS Cron Job (Azure Kubernetes Service)

AKS is Azure's managed Kubernetes service, used here to deploy the **PortfolioManager** as a containerized job. This choice is due to PortfolioManager's long runtime and need for sequencing—such as waiting for a 'close' trade to complete before issuing an 'open'.

The AKS Cron Job allows precise scheduling, resource control, and recovery handling, making it ideal for managing this more complex and stateful workload. It also benefits from Kubernetes-native logging and monitoring.

Data sources

This section outlines the data used during both the POC and final implementation phases. The POC phase relied on manually collected datasets and public APIs to support offline evaluation and model prototyping. In contrast, the final phase represents a scalable, automated production setup powered by API-driven architecture, integrated into a live system for real-time decision making.

POC Phase

Data was collected from a combination of internal systems and public sources such as the NASDAQ API and internet-accessible datasets. It was used primarily to simulate the full pipeline in an offline setting using static CSV files. The goal was to validate model feasibility, tune key parameters, and assess whether sentiment-based trading strategies could outperform a benchmark.

During the Proof of Concept (POC) phase, data was collected from a combination of internal company systems and external sources such as the NASDAQ API and public datasets on the internet. The primary aim was to simulate and evaluate the model offline using manually prepared CSV files for ease of control, transparency, and reproducibility.

The following structured datasets were used:

Social Posts

Collected from internal company systems. Used by the SentimentsFunc to extract sentiment toward specific symbols. Only English-language posts were included to maintain sentiment clarity.

Field Name	Description	Type
OwnerID	Unique identifier of the user	Integer
MessageText	Text content of the social post	String
CreateTime	Date and time the post was published	Datetime

User Data

Extracted from internal user profile databases. This data supported credibility weighting by identifying active, engaged users.

Field Name	Description	Type
OwnerID	Unique identifier of the user	Integer
PlayerLevel	User level/status in platform	Integer
MonthsActive	Number of months the user was active	Integer

User Gains

Collected to provide a basic indicator of user trading skill. This helped in generating credibility scores for weighting sentiment influence.

Field Name	Description	Type
OwnerID	Unique identifier of the user	Integer
Gain	2-year historical return	Float

Price Data (Raw per symbol)

Downloaded via NASDAQ API using a custom Python script. Used to compute daily returns by comparing open and close prices.

Field Name	Description	Type
Date	Trading day	Date
Open	Market open price	Float
Close	Market close price	Float

Price Data (Aggregated)

Combined into one file to reduce processing overhead. These were used as inputs in backtesting to evaluate portfolio returns.

Field Name	Description	Type
Date	Trading day	Date
Symbol	Ticker symbol of the company	String
Daily_Gain	Gain from open to close	Float

S&P 500 Symbol List

Uploaded into an internal database. Processes exist to maintain and update the list to ensure trading decisions are made against the correct benchmark universe.

Fetched once from a public list and assumed to be mostly stable over the test period. Used to define the investment universe for scoring and evaluation.

A static CSV downloaded once, listing companies assumed stable over the test window.

Final Phase

The final system phase uses fully automated API calls to dynamically retrieve and process all required information. Data integrity, consistency, and timeliness are critical, as these feeds are used to operate the model live and support trade decisions.

In the production-ready system, all data will be accessed via APIs to ensure consistency, automation, and real-time readiness. These APIs represent the future-state architecture and will be integrated into the full implementation.

Social Posts API

Supplies the real-time feed of social media posts. Used in SentimentsFunc to extract and score symbol-related sentiment.

Field Name	Description	Type
ownerId	ID of the post author	Integer
messageText	Text body of the post	String
createTime	Timestamp when the post was created	Datetime
language	Language of the post	String

User Data API

Returns structured user data to UserDataFunc. MonthsActive is computed from the difference between registration and first deposit date. Certifications are used to improve credibility scoring.

Field Name	Description	Type
id (ownerId)	Unique user ID	Integer
playerLevel	User tier or activity level	Integer
registrationDate	Date when user registered	Date
firstDepositDate	Date of first platform deposit (to infer activity)	Date
certifications	List of user qualifications (e.g., analyst, advisor)	List

Performance Data API

Provides dynamic performance metrics used to evaluate recent user success. This contributes to credibility scoring and improves sentiment weighting.

Field Name	Description	Type
id	Unique user ID	Integer
gain	Performance return for given period	Float
period	Time window label (e.g., '1y', '6m')	String

S&P 500 Symbol List

Managed internally in a system database with maintenance logic for additions/removals.

Trading API

Used to fetch current portfolio allocation. This enables real-time comparison with target allocation and informs trade decisions.

Field Name	Description	Type
symbol	Ticker symbol in portfolio	String
quantity	Quantity of shares held	Integer
price	Current market price of the symbol	Float

Price Data

Excluded from live operations, since real-time pricing is not necessary for signal generation. All price-based evaluation was limited to the POC phase.

No longer required in the live phase, as price history is only used for evaluation purposes.

Methodology

Sentiment analysis

In order to retrieve the sentiment scores of symbols as response from Azure AI LLM engine, a relevant prompt must be supplied to the LLM for it to be able to break down the content into the relevant symbols and output a specific sentiment per symbol, and not only a global sentiment on the content which may contain both negative towards one stock, and positive towards the other.

Additionally, in order to have multiple levels of sentiment based on the knowledge and intimacy of the writer of the relevant data, a “maxAbsSentiment” parameter is supplied to allow to have a range of sentiments and not simply “Positive”, “Neutral”, and “Negative”. The parameter is injected to the prompt in the calling code, and the LLM engine receives it as numeric range.

The prompt is supplied as “System prompt” to the LLM engine, while the social media content is supplied as “User prompt” to be analyzed.

Below is the prompt used after analysis of results:

> You are a financial sentiment analyzer.
> Analyze the sentiment of the given text and assign a numerical score to each stock symbol mentioned.
> Stock symbols start with a $ sign. Do not include scores for symbols not mentioned.
> Scores must range from -_maxAbsSentiment (very negative) to _maxAbsSentiment (very positive), with 0 indicating neutrality.
> Sentiment should reflect the overall tone towards each symbol, not merely its mention.
> For instance:
>
> - "I think $AAPL is a great company" should yield a positive score for AAPL.
> - "I don't like $GOOGL" should yield a negative score for GOOGL.
> - "What do you think about $AMZN?" should yield a neutral (0) score for AMZN.
>
> Distinguish between subjective opinions and general market inquiries.
> If multiple symbols are mentioned, score each one individually.
> The magnitude of the score should reflect the strength and justification of the sentiment.
>
> For example, "I think $AAPL is a great company because of its strong earnings" should score higher than "I like $AAPL".
> Consider company performance, market trends, and relevant context in your analysis.
>
> Return the result as a JSON dictionary with symbol names (without $) as keys and integer sentiment scores as values.
> Respond with JSON only.
>
> Example:
> { "AAPL": 2, "GOOGL": -1, "AMZN": 0 }

Machine learning

This project applies unsupervised machine learning techniques to optimize parameters that influence trading signals. Since we do not have labeled outcomes for every input (i.e., no direct target to train on), we rely on simulation-based evaluation: we define an objective function that measures portfolio gain and attempt to find weights that maximize it. This unsupervised search allows us to align the model's outputs with real-world investment goals.

Dataset Processing

During the POC phase, sentiment data processing was conducted using a C# console application. This application consumed four input CSV files: the S&P 500 symbols list, social post data, user metadata, and user gain data. Posts were filtered to exclude any that did not mention at least one valid symbol from the S&P 500 list. The valid posts were each sent individually to Azure AI Foundry for sentiment scoring via large language model inference.

Once sentiment responses were returned, they were filtered again to retain only relevant symbols. Each enriched sentiment entry was then merged with its corresponding user metadata and gain record, producing a final dataset with the following structure:

OwnerID, CreateTime, PlayerLevel, TwoYearGain, MonthsActive, Symbol, SentimentScore

A Python script (process_sentiments.py) was used for post-processing. It:

- Calculates *Date* field based on *CreateTime* of the post, that represents the relevant trade date – media uploaded after the closing of the stock exchange is only relevant for the next trading day.
- Mapped *PlayerLevel* strings to integer levels from 1 to 6 for easier processing
- Grouped data by *OwnerID*, *Symbol*, and *Date*.
- Spread each *SentimentScore* across all days until the next post or a defined end date, and add calculated *daysSincePost* column for each row, representing the number of days since the original post.
- Removed duplicates, retaining only the most recent sentiment for each user-symbol-date combination, so if the user uploaded multiple posts about the same symbol on different days, each time a post is uploaded the relevant *sentimentScore* is updated, as well as the *daysSincePost* counter.
- Exported the cleaned dataset to a CSV file.

Finally, a second Python script (modular_portfolio/main.py) consumed the processed sentiment file and a separate price aggregation file, which contained daily gains by symbol and date. This script combined these inputs into a final machine learning dataset used in the optimization phase to learn the best-performing set of weights.

Feature Engineering and Model Evaluation

In the machine learning phase, each sentiment entry is transformed into a weighted score used for optimization. The score is calculated using the following formula:

$$Score = SentimentScore \cdot (w_0 \cdot playerLevel + w_1 \cdot TwoYearGain + w_2 \cdot MonthsActive + w_3 \cdot \max(0{,}365 - daysSincePost)$$

Where w0, w1, w2, and w3 are weight parameters (floats between 0 and 1) optimized using Differential Evolution.
$\max(0, 365 - daysSincePost)$ is used to have natural decay of the sentiment over time, with a maximum of 1 year, in order to reduce the impact of stale sentiments over the aggregated score.

Model evaluation is performed in several steps:

- For each trading day, scores are aggregated per symbol by summing the scores of all posts that mention it.
- The top 10 symbols with the highest score sum are selected as that day's portfolio.
- Using the gain data for each selected symbol, the average daily return is computed and accumulated.
- The total portfolio return over the test window is calculated as a compounded percentage gain.

The objective function returns this final percentage gain, or applies a high penalty if invalid results are encountered.

A separate visualization script allows further evaluation of model behavior. This script accepts a set of weights, performs the same scoring and aggregation process, and outputs:

- Cumulative gain curve
- Drawdown visualization
- Volatility analysis

These visual tools help assess how consistent and risk-aware different configurations are under real trading constraints.

Differential Evolution Optimization

Differential Evolution (DE) is a population-based, stochastic optimization algorithm well-suited for solving complex, non-differentiable, and multi-modal optimization problems. It evolves a set of candidate solutions over iterations by applying mutation, crossover, and selection operations.

In DE, each generation is composed of vectors representing potential solutions. Mutation is applied by adding the weighted difference of two population vectors to a third vector. A crossover step then combines this mutant vector with the target solution to produce a trial candidate, which replaces the original if it yields a better objective value.

The objective function used during optimization is defined as follows:

```
def objective(weights, merged_df):
    df = merged_df.copy()
    gain = calc_total_gain(weights, df)
    del df    # clear memory
    if gain is None:
        return 1e6
    return -gain
```

This function evaluates a candidate set of weights by computing the total gain using a helper function calc_total_gain. This function runs a simulated portfolio over the provided data and calculates the overall profit based on the supplied weight configuration. If the gain is valid, the function returns its negative (since differential_evolution performs minimization). If no gain is calculated (e.g., due to a data issue or invalid strategy), it returns a high penalty value to discourage the optimizer from exploring that weight configuration.

In this project, optimization of the portfolio strategy relies on fine-tuning weights for credibility scoring and allocation. The following block of code demonstrates how the system uses the differential_evolution algorithm from the scipy.optimize module to optimize a custom-defined objective function:

```
result = differential_evolution(
    objective,
    bounds,
    args=(merged_df,),
    polish=True,
    disp=True,
    updating='deferred',
    workers=cpu_count,
    popsize=15,
    maxiter=100,
)
```

Explanation of Parameters:

- **objective**: A user-defined function that calculates a score or loss metric based on candidate parameters. In this case, it evaluates the portfolio's total gain using a set of weighting inputs and returns the negative value (as DE minimizes).
- **bounds**: A list of tuples specifying the lower and upper bounds for each parameter. Here, it is defined as [(0, 1)] * 4, allowing each of the four weights to explore the full range from 0 to 1.
- **args**: A tuple containing extra arguments to pass to the objective function. In this case, (merged_df,) passes the data to be used for evaluating the gain.
- polish=True: After DE finishes, a local optimization (using L-BFGS-B) is applied to refine the best solution found. Setting this to False skips the local refinement step.
- **disp**=True: Enables live display of optimization progress to the console. Each generation's best score is printed to standard output, allowing visual tracking of convergence.
- **updating**='deferred': Specifies how the population is updated during parallel execution. 'deferred' waits until all workers complete their evaluations before updating the population, providing better stability. Alternative: 'immediate' updates immediately after each worker finishes.
- **workers**=cpu_count: Uses all available logical CPU cores to run population evaluations in parallel, significantly speeding up the optimization when processing-intensive functions are involved.

- **popsize**=15: Defines the base population size multiplier. The actual number of individuals evaluated per generation is popsize × number of parameters. For 4 weights, this yields a population of 60.
- **maxiter**=100: Sets the maximum number of generations (iterations) for the evolutionary process. The algorithm may stop earlier if convergence criteria are met, such as if the improvement over iterations falls below a threshold (tol) or if no improvement is detected over several generations.

What is the Polish Step?

The polish=True step activates a post-optimization local search using the L-BFGS-B algorithm. This is a gradient-based optimization technique that can find more precise minima by refining the best solution found by DE. It ensures fine-tuning near convergence and improves the stability of the final result.

Together, the use of Differential Evolution with a polish step helps ensure a robust and globally-optimized solution for the model's weighting parameters.

Why Use Both Differential Evolution and L-BFGS-B?

Differential Evolution is robust for global search and can avoid many local minima due to its population-based exploration. However, like most evolutionary algorithms, it can still converge prematurely or stall in flat regions of the search space.

To refine the best solution further, the algorithm optionally performs a local search at the end (polish=True) using L-BFGS-B, a gradient-based optimization technique. This local refinement improves convergence precision and typically helps fine-tune the solution that DE identifies. By combining the global search power of Differential Evolution with the precision of L-BFGS-B, we achieve a more stable and well-optimized solution for portfolio weighting.

Results

The Proof of Concept phase was executed with two primary analytical windows:

- Data collection and analysis – January 1st, 2024 to May 31st, 2025
- Evaluation and back-testing – June 1st, 2024 to May 31st, 2025

During the data collection phase, over 200,000 original English-language posts were gathered from social media. These posts were filtered to retain only content that included relevant stock symbols. Sentiment analysis was applied using a language model to determine the tone toward each mentioned stock. To ensure consistency and clarity, posts in other languages were excluded. The dataset was further refined by focusing only on stocks that are part of the benchmark index. Additional metadata, such as user profiles and performance history, was combined with the sentiment output to create a rich dataset for model analysis.

In the evaluation phase, the algorithm was tested over a 12-month period using the previously gathered data. A simulated portfolio was managed by the algorithm during this time. The results were:

- **Total gain** of approximately 21.8%.
- **Benchmark (S&P 500) return** during the same period: 12%.
- **Alpha** (excess return): 9.8%.

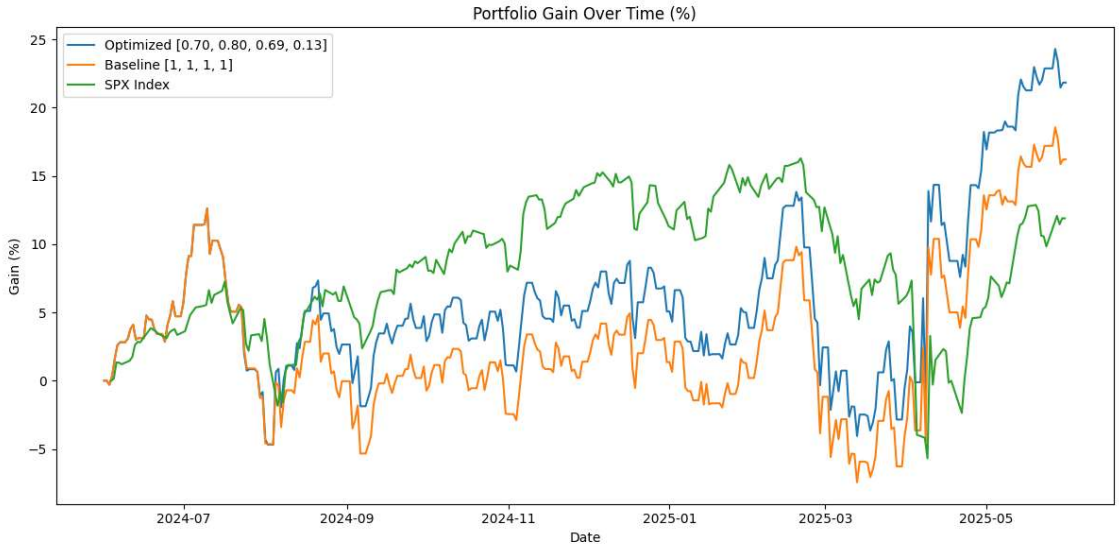

Figure 6: Portfolio gain over time between June 1st 2024 and May 31st 2025. Blue - optimized weights of the algorithm, Orange - baseline weights of 1 for the algorithm, Green - S&P 500 index.

As shown in *Figure 1*, the algorithm (blue line) maintained a generally upward trajectory throughout the evaluation period. Gains were achieved steadily rather than coming from a small number of isolated high-performance spikes. This consistent growth pattern is an encouraging indicator of robustness.

Even during months when the algorithm underperformed relative to the benchmark, it still managed to achieve positive returns.

In terms of risk, the algorithm also showed promising characteristics:

- Maximum drawdown: ~15%.
- Benchmark drawdown: ~18%.

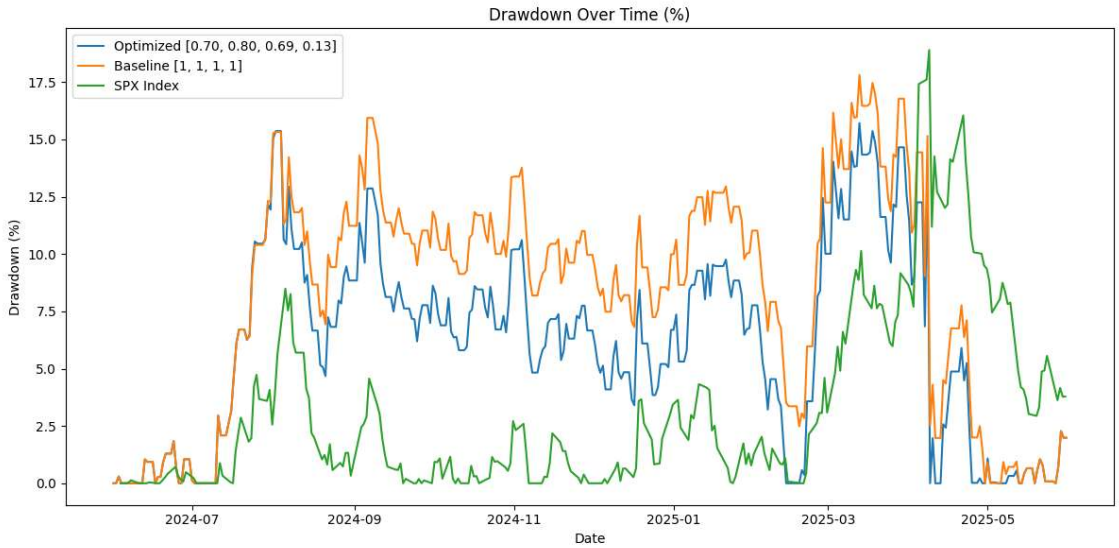

Figure 7: Portfolio drawdown over time between June 1st 2024 and May 31st 2025. Blue - optimized weights of the algorithm, Orange - baseline weights of 1 for the algorithm, Green - S&P 500 index. Peak representing max drawdown.

These results show that the algorithm not only outperformed the benchmark but also provided slightly better protection against downside risk. However, it's important to note that these findings are based on back-testing, and live market conditions could introduce additional challenges not captured here.

During model development, various portfolio strategies were explored. These included:

- Proportional allocation based on sentiment scores.
- Broader diversification by including the top 15 or top 20 stocks.

Ultimately, the best performance came from a simpler approach:

- Top 10 scoring stocks, with equal capital allocation.

This straightforward strategy yielded the best overall results and balanced performance with simplicity.

Conclusions

The results of this Proof of Concept support the potential of using sentiments from social media posts and user credibility metrics to drive an algorithmic trading strategy that can achieve better performance than the benchmark index. Even a relatively simple portfolio construction method—such as equal weighting across the top sentiment-scored assets—managed to outperform the S&P 500.

However, the performance was not consistent across the entire test period. The algorithm only began to significantly outperform the benchmark in the final months of the evaluation window. This suggests that the portfolio may have lacked diversification and relied too heavily on a small number of outperforming stocks. Further research is required to develop a model that delivers more balanced returns over time, rather than showing strong results only from start to end.

In terms of risk, the algorithm demonstrated higher volatility in some periods compared to the benchmark. A key way to address this would be by increasing the number of assets included in the portfolio, thereby improving diversification and lowering exposure to individual stock swings.

One notable limitation was the use of only English-language content. While this ensured sentiment accuracy, it also excluded a large amount of potentially valuable data in other widely-used languages such as Spanish, German, and French. Expanding language support to include those that are already supported by large language models could significantly enhance coverage and model performance.

Another simplifying assumption was that the back-testing considered only daily gains based on the difference between open and close prices—without holding positions overnight. While this approach reduces model complexity and rebalancing requirements, it also omits performance changes that occur between trading sessions and might impact real-world applicability.

In conclusion, although the POC model did outperform the benchmark within the test timeframe, the performance lacked consistency and introduced greater risk, making it insufficient for immediate implementation. Nonetheless, the results establish a promising foundation for further research into building a more balanced, scalable, and production-ready trading model.

Future work

To continue developing the model and improve both reliability and performance, future work is proposed across four main areas:

1. Data and Feature Enhancements:

- **Multilingual Support**: Expand the model's ability to analyze posts in multiple languages such as Spanish, German, and French, to significantly increase data coverage and enrich sentiment inputs.
- **User Profiling Enhancements**: Improve user credibility scoring by integrating multiple performance metrics (e.g., 2-year, 1-year, 6-month, average annual return) and including professional credentials such as certified financial advisor, professional trader, or stock analyst.
- **LLM Model change**: In the POC the LLM model used for the sentiment analysis was "*gpt 4o mini*" as it is cost efficient, however there are models that are better and more accurate at performing sentiment analysis, that are more suitable here and were not considered for the POC due to cost reasons.

2. Trading Assumptions and Execution Strategy:

- **Holding Period Adjustments**: Explore holding positions overnight to capture inter-session price changes and evaluate the impact on performance and risk.
- **Market Open Rebalancing**: Implement rebalancing strategies that operate only at market open, using a tolerance band to allow a percentage of deviation between actual and target allocations before executing trades. Additionally, evaluate rebalancing only when the list of portfolio assets changes to reduce turnover and trading costs.

3. Model Optimization and Risk Management:

- **Explore Weight Formulas**: Explore other weight formulas to evaluate the score of each social post over time.
- **Model Optimization**: Refine weighting mechanisms, explore momentum and volatility-based adjustments, and experiment with dynamic rebalancing strategies. Include evaluation of portfolio performance over time.
- **Risk Awareness**: Introduce risk and drawdown awareness into the model, in attempt to maximize performance while also minimizing risk taken.

4. Processing Optimization and Performance:

- **Distributed Processing with SparkML**: Transition the ML workflow to use SparkML to improve processing speed and scalability, especially as data volume grows.
- **Azure AI Cost Efficiency**: Replace per-call sentiment analysis with bulk-processing calls to Azure AI services to significantly reduce computation costs.

These directions emphasize the potential to enhance the model's relevance and feasibility as a tool for generating long-term market outperformance, by refining inputs, reducing unnecessary operations, and managing risk more effectively.