Automated Trading with Cashew Capital

Report on an AI Studio Project

—

Advisors
Raisa Lobo, COO
Raisa@cashewcapitalfund.com

Bharath Venkataraman, CEO and CIO
Bharath@cashewcapitalfund.com

Boshen Parthasarathy, CTO
Boshen@cashewcapitalfund.com

Kevin McGee, CFO
Kevin@cashewcapitalfund.com

Contributors
Devina Gera
Fridah Ntikah
Humaira Sarwary
Kevin McGee
Krishna Kanodia
Lorraine Estrella De Leon
Lydia Chen
Nha-Van Nguyen
Osasikemwen Ogieva – oogieva25@amherst.edu

Source Code
https://github.com/CashewCapital20/Automated-Trading/

Table Of Contents

Overview

The goal of this project is to use AI and ML algorithms to make informed investment decisions by identifying trading opportunities using technical analysis.

We utilize data engineering, machine learning, and trading strategy evaluation to create a robust framework for predictive stock trading. Historical and real-time price data are sourced from Benzinga and persisted in MongoDB Atlas for efficient reuse.

Buy and sell signals are predetermined via percent changes between current and future prices and classification models are trained to predict them using technical indicators like EMA, RSI, MACD, volatility and Stochastic Oscillators.

Machine learning models we build include random forest, logistic regression, gradient boosting and ensemble stacking classifiers. We tune their hyperparameters with grids and evaluate them via classic metrics.

Real-time trading scripts simulate decision-making using delayed data from Benzinga and one of our pre-trained and persisted models, A user-friendly web interface collects user input in the form of a stock ticker and provides actionable buy, sell, or hold recommendations,

Data

Sources

Data was sourced primarily from Benzinga, using its Python client. Some placeholder data was simulated.

Persistence

Whenever we fetch data afresh from Benzinga, to avoid hitting limits, we also upload it to a MongoDB Atlas database, so we can continually reuse it.

This is done in the fetch_load_mongo.py file. This program takes in a stock symbol, fetches the last 383 days of historical price data for the asset as a list of price candles (open-high-low-close data), which is then converted to a dataframe and uploaded to MongoDB. It also uploads the last 26 days of data separately for recent analysis.

This file also provides functionality for calculating the technical indicators. These include the Exponential Moving Averages (the 12-period EMA, the 26-period EMA); the Relative Strength Index (RSI); the Stochastic Oscillator (the K and D lines); and the Moving Average Convergence Divergence (MACD), including its signal line and histogram.
We then combine these indicators into the dataframe, our return value.

Throughout, there is careful exception handling and specified error messages for possible failures in code and the API.

The test_fetch_load_mongo.py file verifies that the above described data fetcher works as expected.

The mongo_workflow_fix.py file specifies a program which connects to our MongoDB Atlas database instance and retrieves documents we stored there earlier from Benzinga, based on a specific stock ticker.

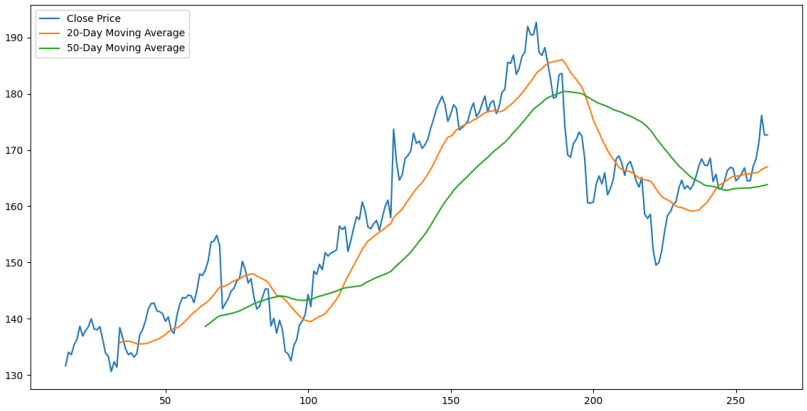
Exploratory Analysis

The dataset_exploratory_analysis.py file fetches historical stock data from Benzinga's API for five stock tickers ("GOOG", "NVDA", "PG", "SPY", "PLTR") for the last 383 days.

Here we import and use the calculate_indicators function from our data fetcher and calculate the indicators on our collected data. Additionally, we calculate the rolling mean

and the 20-day rolling standard deviation, and visualize how all of these measures evolve with time, especially in comparison to the close price.

Plots are saved as images in the /exploratory_analysis folder.



Moving Averages and Close Price for GOOG

We also check for null values.

Training

The bulk of this work is in the train_model.py and train_more_models.py files.

Data, Labels and Features

We begin by connecting to our MongoDB collection, retrieving price data documents and converting them to a dataframe (in the fetch_data function).

The prepare_labels function generates a trading signal column based on future price movements of the asset. These signals will be used as the labels for training and evaluating our model.

Inputs into this function are future_period – the number of days ahead to look for the price change and threshold – the percentage change threshold to classify signals. Default values are future_period=2 days, and threshold=5%.

We calculate a future_price column by shifting the price column upward by future period (2) rows. This allows comparison of the current close price to the future price, which lets us compute the percentage_change between the current price and the future price.

Finally, we use the percent change and the threshold to determine the signals. We start by initializing the signal column with all values set to 0.

- Set signal to 1 if the price change is greater than the positive threshold (indicating a "buy").
- Sets signal to -1 if the price decreases after a previous "buy" signal, indicating a "sell".
- Sets signal to -2 if the price change is less than the negative threshold (indicating a significant drop and a strong sell).
- Sets signal to 2 if the price increases after a previous "strong sell" signal, indicating a "re-entry buy".

The prepare_features_and_labels function segments our data into X (features) and y (labels) in preparation for model fitting. The features are the macd, the macd_hist, the rsi, the slowk and the slowd indicator columns. The newly created signal column is the label.

Random Forest Classifier Model

Our random forest classifier has 100 estimators, 20 maximum features and a maximum depth of 7. It was trained on 70% of our data with 30% reserved for testing, and saved as a .joblib file in our repository.

Ensemble Model and Some Other Models

In train_more_models.py, we go on to build a logistic regression model, a support vector classifier, an AdaBoost classifier (with 100 estimators) and finally a stacking classifier.

The stacking classifier contains: a random forest classifier (with 100 estimators), a gradient boosting classifier (with 100 estimators), a logistic regression model, a support vector classifier and an AdaBoost with 100 estimators. It has a logistic regression model as the final estimator. This program is primarily about laying out a data pipeline for model training, evaluation and hyperparameter selection with Grid Search.

As such, the data is simulated. We generate 500 data points for our features (MACD, RSI, Stochastic Oscillator, and close price) from random distributions. We also include new indicators (momentum, daily return, and volatility). We also run some visualizations for these values to understand what our random data appears like. Then we define our signal column as a buy (1) or hold/sell (0) signal based on the next day's close price. We split data into training and testing sets (80% training, 20% testing).

After training, we evaluate each model using accuracy, a confusion matrix and a classification report (which provides detailed metrics like precision, recall, and F1-scores).

We use GridSearchCV for tuning hyperparameters (number of estimators and maximum depth) of both the random forest and gradient boosting models, logging and using the best parameters. Then we take a hybrid approach, predicting based on the averages of both these models. We evaluate this prediction, too, using accuracy, a confusion matrix and a classification report.

Model Comparison

The model_comparison.py script provides a comparative analysis of different machine learning models for stock trading predictions:

- **Regression Models**: Predict continuous outcomes and are evaluated with RMSE.
- **Classification Models**: Predict discrete outcomes (e.g., buy/sell signals) and are evaluated with accuracy.

Real Time Trading

Our implementation of a real-time trading system is an integration of several Python scripts.

The delayed_stock_quote.py fetches trading data from Benzinga which is 15 to 20 minutes behind the present, and this is what we use for simulating our real time trading.

The real_time_trading.py script executes real-time trading using this data and the pre-trained random forest classifier model we persisted as a .joblib file in our repository to make trading decisions.

These trade logs are uploaded to our MongoDB database by the trade_logs.py script. This program also calculates the remaining funds and inventory after each transaction.

The paper_trade_demo.py script is a stand-alone trading simulation which also uses our pre-trained random forest classifier model. It fetches random historical 5-minute candle data for a trading day, loads our model and simulates trading decisions based on model predictions, printing the results.

Back Testing

The script back_test_predictions.py is designed to evaluate the performance of a trading strategy by calculating various financial metrics, such as Sharpe Ratio, Max Drawdown, Profit Factor, and Win Rate. It also visualizes the cumulative returns and drawdowns over time.

User Interfacing

The interface.py and app.py scripts create a web-based, user-friendly interface for our project. Users should be able to enter a stock ticker and get a recommendation to buy, sell or hold.