Carnegie Mellon University

Algorithmic Trading

Andrew Li, Ishaan Singh, Tiffany Zhan, Ethan Zhang

Research Mentor: David Handron
Research Program: CFSURP

Contents

1 Setup and Environment

Among the many available online brokers, we chose Interactive Brokers (IB) for its comprehensive data access and broad functionality. IB offers multiple web, desktop, and app platforms, but for algorithmic trading, Trader Workstation (TWS) provides the most robust set of tools to work with.

Todo. Create an IB account and submit an application.

Tip 1.1 (API Access Requirement). In your application be sure to indicate that your net worth exceeds $20,000, as Interactive Brokers requires this minimum to hold an IBKR Pro account. IBKR Pro is necessary to access the trading API. If you're only paper trading, selecting IBKR Pro has no other consequences.

1.1 Installing Trader Workstation

Todo. Visit the `TWS API documentation`→ page and follow the instructions specifc to your OS to download and install Trader Workstation (TWS).

Tip 1.2 (`setup.py` Permissions Error). On macOS or Linux, if running `python3 setup.py install` returns **error: could not create 'ibapi.egg-info': Permission denied**, give the current directory writable permissions with `sudo chmod -R 777 .` (note the trailing period, which specifies the working directory).

Here are some of our recommended TWS settings changes:

(1) Go to File → Global Configuration... → API → Settings

(2) Enable "ActiveX and Socket Clients"

(3) Disable "Read-Only API"

Note. The default ports are 7497 for paper trading and 7496 for live trading.

1.2 Python Environment Setup

In this project, we use Python for its compatibility with the IB wrapper library `ib_insync` and support for financial sentiment models like FinBERT.

To setup your environment:

- Download Python (if you haven't already)
- Install `ib_insync` with `pip install ib_insync`
- Install `NumPy`, `Matplotlib`, and `Pandas` with `pip install pandas numpy matplotlib`

2 Using the API and ib_insync

The wrapper library `ib_insync` simplifies interactions with the TWS API, allowing operations that would typically require 10–15 lines of code to be performed with a single function call. For more details, refer to the official `ib_insync docs`→. In this section, we introduce some basic `ib_insync` operations that will enable us to interact with the Interactive Brokers API.

2.1 Basics

Establishing Connection

To begin, verify that you can connect to the API using the code below. If you reach an error, refer to **Tip 1.1** and **Tip 1.2** for troubleshooting.

```
from ib_insync import *
ib = IB()
ib.connect('127.0.0.1', 7497, clientId=1) # 7497 = TWS paper port
print("Server time:", ib.reqCurrentTime())
ib.disconnect()
```

The first three lines are essential for any script that interacts with IB. From this point on, we omit them from code examples for brevity.

Contracts

Below are several examples of how to define contracts:

```
Contract(conId=270639)
Stock('AMD', 'SMART', 'USD')
Stock('INTC', 'SMART', 'USD', primaryExchange='NASDAQ')
Forex('EURUSD')
CFD('IBUS30')
Future('ES', '20180921', 'GLOBEX')
Option('SPY', '20170721', 240, 'C', 'SMART')
Bond(secIdType='ISIN', secId='US03076KAA60');
```

Some additional commands for working with *contracts*

- `ib.reqContractDetails(contract)` — Requests detailed information on a given contract object.

- `ib.qualifyContracts(contract)` — Uses the information returned from `reqContractDetails` to fill in missing fields in the original contract.
- `ib.reqMatchingSymbols(str)` — Requests a list of stock contracts that match a given symbol pattern (only returns stocks.)

2.2 Getting data

Requesting Historical Data

To retrieve historical OHLC (open-high-low-close) data, use the function `reqHistoricalData`. Once the data is received, you can use `matplotlib` to plot the bars. [LINK TO SECTION]

```
import matplotlib.pyplot as plt
contract = Stock('TSLA', 'SMART', 'USD')

#Returns the earliest date available bar data
print('Earliest data available: ', ib.reqHeadTimeStamp(contract,
↪   whatToShow='TRADES', useRTH=True), end = '\n')

#Request hourly data of the last 60 days
bars = ib.reqHistoricalData(
        contract,
        endDateTime='',
        durationStr='60 D',
        barSizeSetting='1 hour',
        whatToShow='TRADES',
        useRTH=True,
        formatDate=1)

#Creates a DataFrame object given a list IB objects
df = util.df(bars)
print('Example 5 lines of DataFrame: \n', df.head(), end = '\n')

#Line graph of close data
df.plot(y='close')
plt.title('TSLA Close Price Chart (60 Days)')
plt.show(block = False)

#Candlestick graph of bars
util.barplot(bars[-100:], title=contract.symbol)
plt.title('60-Day Candlestick Chart for TSLA')
plt.show()
```

To receive real-time updates alongside historical data, set `endDateTime` to an empty string and `keepUpToDate=True` in your `reqHistoricalData()` call.

Requesting Live Market Data

To request live market data, we establish a local WebSocket connection to the server. The server then continuously sends OHLC data at regular intervals. For Interactive Brokers, the minimum interval for live data updates is 5 seconds.

In practice, `reqRealTimeBars()` is used to stream real-time bar data reliably. However, this requires a market subscription in order to Recall that `reqHistoricalData()` with `keepUpToDate = True` also supports live updates, but it is less robust. After a network interruption, it often requires a full reconnection. In contrast, `reqRealTimeBars()` resumes automatically and backfills missing data.

2.3 Executing Orders

Placing Market Orders

To place a basic market order, use the `MarketOrder()` constructor and the `ib.placeOrder()` method. Below is a sample call:

```
from ib_insync import *
ib = IB()
ib.connect('127.0.0.1', 7497, clientId=1)

contract = Stock('AAPL', 'SMART', 'USD')
ib.qualifyContracts(contract)

order = MarketOrder('BUY', 100)
ib.placeOrder(contract, order)
```

This code snippet purchases 100 shares of AAPL at the current market price. Replace `BUY` with `SELL` to issue a sell order. Always use `ib.qualifyContracts()` beforehand to ensure the contract is fully defined.

Limit and Stop Orders

`LimitOrder()` allows you to control the maximum price you pay (or minimum price you receive), while `StopOrder()` triggers a market order when a certain price is reached. Example:

```
limit_order = LimitOrder('SELL', 100, 185.50)
stop_order = StopOrder('SELL', 100, 180.00)
```

```
ib.placeOrder(contract, limit_order)
ib.placeOrder(contract, stop_order)
```

Be mindful that limit orders may not execute if the market price doesn't reach your limit price.

Order Status and Updates

To monitor status updates (e.g., order filled, partially filled), use event-driven callbacks:

```
def onStatus(order):
    print('Order status updated:', order.orderId, order.status)

ib.pendingTickersEvent += onStatus
```

This sets a callback function that prints updates whenever order status changes.

Order Lifecycle Management

All placed orders remain tracked in `ib.orders()` and `ib.filledOrders()`. You can cancel an open order with: `ib.cancelOrder(order)` You can also check `order.filled`, `order.remaining`, and `order.status` for real-time tracking.

Bracket Orders and Order Groups

Bracket orders help automate take-profit and stop-loss logic. A bracket consists of three orders: entry, take-profit, and stop-loss.

```
entry = MarketOrder('BUY', 100)
takeProfit = LimitOrder('SELL', 100, 195.00)
stopLoss = StopOrder('SELL', 100, 175.00)

for o in [entry, takeProfit, stopLoss]:
    ib.placeOrder(contract, o)
```

Use `ib.bracketOrder()` to generate a bracket set, then submit them with `ib.placeOrder()` for each.

Note on Execution

Execution requires that your IBKR account is properly configured and funded. For paper trading, ensure you are connected to port 7497. Note that market data and order execution may be delayed or rejected if required subscriptions or permissions are missing.

3 Financial Indicators

For model construction and backtesting, financial indicators are critical. The `yfinance` library provides an efficient way to download historical market data collected from Yahoo Finance. In theory these indicators can be collected with web scraping and individual API calls, but `yfinance` greatly simplifies the process. In the sections below, we will go over installation of the library, and how to use its most common features.

3.1 Basics

Downloading the library

To download the `yfinance` library into your computer, run

```
pip install yfinance --upgrade --no-cache-dir
```

This command will download yfinance, and also all dependencies. In specific, the baseline dependencies are pandas 0.24, numpy 1.15, requests 2.21, and multitasking 0.0.7.

Utilizing the library

The key to using the `yfinance` library is the Ticker object, which contains all the information and methods regarding a certain stock symbol. For example, if we wished to create said ticker object for Apple Inc., we would do

```
important yfinance as yf

aapl = yf.Ticker("AAPL")
```

For the sake of brevity going forward, we will be omitting the import of yfinance each time. Do remember that this is critical and necessary for every usage however.

3.2 Basic Data Collection and Manipulation

Historical Data

For cumulative data collection, using code such as

```
aapl = yf.Ticker("AAPL")

data = aapl.history(period="1mo",interval="1d")

data = aapl.history(start="2023-01-01",end="2024-01-01")
```

will generate a Pandas DataFrame struct into data. Here, note that period and interval are adjustable, with period meaning how much data, and interval representing the granularity of said printed data. Example options would be '1d', '5d', '1mo', etc. Note running `data.shape` would provide the size of this DataFrame struct for reference.

Additionally, the second line is another option for data collection, where the user can specify a start and end day for the data. Note the granularity of data here is assumed to be 1 day.

Understanding DataFrames

In `yfinance` and many other financial libraries, most data will be provided to you in the form of Pandas DataFrames. Thus, it is key to understand how to manipulate and use these structures. While plenty of resources exist online regarding usage, a few key ones we will list out for convenience are

- `data['Close'].plot()` to plot closing prices
- `data['Returns'] = data['Close'].pct_change()` to compute daily returns
- `data.resample('W').mean()` to get weekly aggregates

These are just a few key ideas of manipulations of the DataFrames we have found to be useful. In later sections you will find more elaborate and comprehensive uses of yfinance DataFrames, that should simplify your work and give creative foundation for your own code.

3.3 Advanced Data Collection

Below are examples of advanced, more specific data that requires increased attention and manipulation to be of use.

Company Info

To obtain basic level metadata regarding companies, such as industry, sector, and more, one can use the `info` dictionary attribute from `yfinance`, as so:

```python
info = ticker.info
print(info['sector'], info['marketCap'])
```

This will return a dictionary containing fields of all necessary metadata, such as industry, fullTimeEmployees, marketCap, and more. To see all these actual features, run

```
print(ticker.info['industry'])
```

Company Actions

Another important factor in stock consideration is the actions in dividends and stock splits regarding a specific ticker. To access this, run something like

```python
actions = ticker.actions
print(actions)
```

The result will provide a DataFrame with dividends and split ratios organized by date. Hypothetically, if your code requires only dividends for simplicity (as they are significantly more applicable to trading), you could run something such as

```python
dividends = ticker.dividends
print(dividends)
```

Split are less useful, but perhaps you would like to consider them separately. If so, run

```python
splits = ticker.splits
print(splits)
```

Financial Statements

For key info about income, balance sheets, and cash flow, `yfinance` provides them to us with use of

```python
income = ticker.financials
balance = ticker.balance_sheet
cashflow = ticker.cashflow
```

These will return Pandas DataFrames with columns separated by time and rows being the key values under consideration.

To extract data more specific to certain time frames, such as last quarter, run

```
ticker.financials.loc["Total Revenue].iloc[0]
```

Shareholders

Sometimes understanding the shareholders and their portions are useful in determining company structure and future. To get this information, run

```
major = ticker.major_holders
institutions = ticker.institutional_holders
```

3.4 Advanced Data Manipulation

While `yfinance` provides a large variety of raw data, it abstains from providing key technical indicators. Thus, for purposes of modeling and trading, we will need to construct these indicators ourselves with data collected from `yfinance`. In this section, we will go over some key indicator code that we found useful, alongside how to use it. For the most part, however, it will be up to the reader to code in the necessary indicator calculations.

Example Technical Indicator Code

Here, we will exemplify how to create and use code pertaining to technical indicators with the `yfinance` library. For example, to compute moving averages over certain timespans, potential code you can use is

```
data["MA20"] = data["Close"].rolling(window=20).mean()

import ta
data["RSI"] = ta.momentum.RSIIndicator(data["Close"]).rsi()
```

Underneath, note there is code for calculating the relative strength index, with use of the `ta` package. Going forward, you may want to begin downloading many of these packages as a prerequisite header in your files, as they will come in handy frequently.

Plotting

Now that you have seen an example for calculating these technical indicators, we will look into another key component: visualizing your code. To do so, we will use the `matplotlib` library. More documentation can be found online, but here we will guide you on the most applicable usage, alongside an example. To quickly visualize the stock, note the rudimentary usage of matplotlib can be seen as follows.

```python
import matplotlib.pyplot as plt

data["Close"].plot(title="AAPL Closing Price")
plt.show()
```

This will plot the closing price of, in this case, AAPL, given that the DataFrame "data" was extracted to be the historical pricing of AAPL from `yfinance`. Additionally, a common financial indicator is the Bollinger Bands. For practice, attempt first calculating these bands from extracted data from either IB or `yfinance`. Then, use matplotlib to overlay it with a graph of stock price.

Extensions into ML

As you might see elaborated in future sections, `yfinance` serves as a strong way to scrape historical stock data, for use in backtesting engines and machine learning systems, provided in other popular packages such as backtrader.

4 Strategy Development and Comparison

So far, we have collected historical market data, calculated financial indicators, and visualized stock performance. We now turn to the task of constructing actual trading strategies and testing them on historical data (backtesting). This chapter shows how to build strategies using `ib_insync`, evaluate their performance, and compare multiple strategies on different securities.

4.1 Developing a Strategy with `ib_insync`

In algorithmic trading, a strategy is a set of rules that determines when to buy and sell. We will begin by implementing a relatively robust strategy for SPY (an ETF tracking the S&P 500), using the following three technical indicators:

- **EMA (Exponential Moving Average)**: A smoothed version of the average price over a period (20 bars here), giving more weight to recent prices.
- **MACD (Moving Average Convergence Divergence)**: A momentum indicator computed as the difference between short-term and long-term EMAs. We also compute a signal line, which is the EMA of the MACD line itself.
- **RSI (Relative Strength Index)**: A momentum oscillator measuring recent price changes to evaluate overbought or oversold conditions. Values above 70 often suggest overbought (sell), and values below 30 suggest oversold (buy).

We use `ib_insync` to fetch SPY's historical prices, calculate these indicators using the `ta` package, and then define conditions for when we will buy or sell.

```python
from ib_insync import *
import pandas as pd
import ta

ib = IB()
ib.connect('127.0.0.1', 7497, clientId=1)

spy_contract = Stock("SPY", "ARCA", "USD")
spy_bars = ib.reqHistoricalData(
    spy_contract, '', "5 D", "1 min", "TRADES", useRTH=True, formatDate=1
)
spy_df = util.df(spy_bars).set_index('date')

```

```python
spy_df['ema20'] = ta.trend.ema_indicator(spy_df['close'], window=20)
macd = ta.trend.MACD(close=spy_df['close'])
spy_df['macd'] = macd.macd()
spy_df['macd_signal'] = macd.macd_signal()
spy_df['rsi'] = ta.momentum.rsi(spy_df['close'], window=14)

ib.disconnect()
```

Explanation:

(1) We first connect to the IB Gateway and request 5 days of 1-minute historical data for SPY.

(2) We calculate the EMA20, MACD and MACD signal line, and RSI using the `ta` library.

(3) These features are stored in a DataFrame that we'll use for simulation.

4.2 Backtesting and Performance Evaluation

Backtesting means running the strategy on past data to simulate trading and assess profitability and risk. For every time step, we simulate what a trader would do: either enter a position, exit a position, or hold.

```python
initial_cash = 100000
cash = initial_cash
position = 0
portfolio = []
trades = []

for i in range(1, len(spy_df)):
    row = spy_df.iloc[i]
    price = row['close']
    signal = None

    if row['close'] > row['ema20'] and row['macd'] > row['macd_signal'] and row['rsi'] < 70:
        if position == 0:
            qty = cash // price
            if qty > 0:
                position = qty
                cash -= qty * price
                trades.append({'type': 'BUY', 'price': price, 'time': row.name})
```

```
    elif row['close'] < row['ema20'] and row['macd'] < row['macd_signal'] and row['rsi'] > 30:
        if position > 0:
            cash += position * price
            trades.append({'type': 'SELL', 'price': price, 'time': row.name})
            position = 0

    value = cash + position * price
    portfolio.append({'time': row.name, 'value': value})

perf_df = pd.DataFrame(portfolio).set_index('time')
```

Explanation:

- We start with a fixed cash amount ($100,000).
- We iterate through each time row in the DataFrame and decide to buy if all three conditions are met (price above EMA, MACD above signal, and RSI $<$ 70).
- We sell if the reverse of all those conditions are met.
- We simulate each trade, track the cash, portfolio value, and store each action.
- The portfolio value is recalculated at every step.

Performance Metrics

Once we finish the backtest, we compute performance statistics:

- **Total Return**: The difference between final portfolio value and initial cash.
- **Max Drawdown**: The largest loss from peak portfolio value to a following trough.
- **Sharpe Ratio**: A standardized risk-adjusted return metric. It is calculated as:

  $$\text{Sharpe Ratio} = \frac{\text{mean return}}{\text{return standard deviation}} \times \sqrt{252 \times \text{trading bars per day}}$$

  In our case, we use 390 bars/day to reflect 1-minute bars over standard U.S. market hours.
- **Win Rate**: The fraction of round-trip trades (buy then sell) that were profitable.

4.3 Building a Simpler Strategy on AAPL

To demonstrate comparison, we create a second strategy using AAPL stock. This strategy is much simpler: we buy when the price crosses above the 20-period EMA and sell when it drops below.

This shows how complexity does not always lead to better returns — sometimes simpler strategies perform comparably or better.

```python
aapl_contract = Stock("AAPL", "SMART", "USD")
aapl_bars = ib.reqHistoricalData(
    aapl_contract, '', "5 D", "1 min", "TRADES", useRTH=True, formatDate=1
)
aapl_df = util.df(aapl_bars)
aapl_df['EMA20'] = aapl_df['close'].ewm(span=20).mean()

cash = 100000
position = 0
portfolio = []

for i in range(20, len(aapl_df)):
    price = aapl_df['close'].iloc[i]
    ema = aapl_df['EMA20'].iloc[i]

    if price > ema and position == 0:
        position = cash // price
        cash -= position * price

    elif price < ema and position > 0:
        cash += position * price
        position = 0

    value = cash + position * price
    portfolio.append({'time': aapl_df['date'].iloc[i], 'value': value})

perf_df = pd.DataFrame(portfolio).set_index('time')
```

Explanation:

- We download 5 days of 1-minute data for AAPL.
- We calculate EMA20 and use a basic crossover rule: buy when price > EMA20, and sell when price < EMA20.

- Like the SPY strategy, we simulate positions and track the portfolio value.
- Note: This strategy does not track win rate since trades are not logged explicitly.

4.4 Comparing Two Strategies: SPY vs AAPL

To visualize the difference in performance, we plot the portfolio values over time for both strategies.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 6))
plt.plot(perf_spy.index, perf_spy['value'], label='SPY Strategy')
plt.plot(perf_aapl.index, perf_aapl['value'], label='AAPL EMA
↪ Crossover')
plt.title('Equity Curves: SPY vs AAPL')
plt.xlabel('Time')
plt.ylabel('Portfolio Value ($)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Explanation:

- We use `matplotlib` to generate equity curves (value of portfolio over time) for both SPY and AAPL strategies.
- This helps compare volatility, trend, and final return visually.

4.5 Conclusions and Takeaways

This chapter has introduced the basic flow of developing, backtesting, and comparing algorithmic trading strategies using `ib_insync`. We now understand:

- How to define trade signals using indicators.
- How to simulate trades and track portfolio performance.
- How to visualize and compare strategies using equity curves and performance metrics.

In the next chapter, we will transition from historical backtesting to real-time trading, using live market data from Interactive Brokers to make and execute decisions on the fly.

5 Backtesting with Backtrader

Once you have an idea for a strategy, it is critical to test it on historical data before implementing it in real time. Backtesting will allow you to confirm if your strategy has potential by simulating its execution through previous time periods using historical data. There are many tools and API's that can help you backtest many of which require a subscription. We will use a popular python library called BackTrader that helps us backtest our strategy on historical data.

5.1 Getting started with Backtrader

To start backtesting with BackTrader, there are two key features that are required to begin backtesting.

- Data source
- Strategy Class

5.2 Retreiving Data for Backtesting

Before backtesting, we need data to backtest on which can be done in multiple ways including **Yahoo Finance**, **VisualCharts**, CSV, and Pandas. We will show an example of using a pandas database that we retreive from Interactive Brokers.

```python
from ib_insync import *
import pandas as pd

def get_ib_historical_data(symbol, start_date, end_date,
    client_id=1):
    """
    Get historical data from Interactive Brokers using ib_insync
    """
    print(f"Fetching {symbol} data from {start_date} to
     {end_date}...")

    # Connect to IB
    ib = IB()
    try:
        ib.connect('127.0.0.1', 7497, clientId=client_id)
        print(f"Connected to IB with client ID {client_id}")

        # Create contract
        contract = Stock(symbol, 'SMART', 'USD')
```

```python

        # Calculate duration string
        duration_days = (end_date - start_date).days
        if duration_days <= 365:
            duration_str = f"{duration_days} D"
        else:
            duration_str = f"{duration_days // 365} Y"

        # Request historical data
        bars = ib.reqHistoricalData(
            contract,
            endDateTime=end_date.strftime('%Y%m%d %H:%M:%S'),
            durationStr=duration_str,
            barSizeSetting='1 day',
            whatToShow='TRADES',
            useRTH=True,  # Regular trading hours only
            formatDate=1,  # Return as datetime objects
            timeout=2   # Timeout after 2 second
        )

        print(f"Retrieved {len(bars)} bars for {symbol}")

        # Convert to DataFrame
        df = util.df(bars)

        # Clean up the data for BackTrader
        if not df.empty:
            # Ensure proper column names for BackTrader
            df = df.rename(columns={
                'open': 'open',
                'high': 'high',
                'low': 'low',
                'close': 'close',
                'volume': 'volume'
            })

            # Set date as index
            if 'date' in df.columns:
                # Convert date column to proper datetime index
                df['date'] = pd.to_datetime(df['date'])
                df.set_index('date', inplace=True)

            # Ensure the index is proper datetime (not just date objects)
```

```
            if df.index.dtype.name == 'object':
                df.index = pd.to_datetime(df.index)

            # Remove timezone info if present (BackTrader prefers naive datetimes)
            if hasattr(df.index, 'tz') and df.index.tz is not None:
                df.index = df.index.tz_localize(None)

            print(f"Data processed for {symbol}")
            print(f"Date range: {df.index[0].strftime('%Y-%m-%d')} to {df.index[-1].strftime('%Y-%m-%d')}")
            # print(f"Sample data:\n{df.head()}")

        return df

    except Exception as e:
        print(f"Error fetching data for {symbol}: {e}")
        return pd.DataFrame()

    finally:
        ib.disconnect()
        print(f"Disconnected from IB")
```

IB offers us a tool to request historical data which we utilize to create our pandas dataframe. **ib.reqHistoricalData()** takes in multiple parameters we need to specify for before it can retreive data.

- Contract (the ticker you want to look up)
- endDateTime (the end date of the data you want to look up)
- durationStr (a string indicating how far back from the end date to look)
- barSizeSetting (the length of each bar)
- whatToShow (a string indicating what kind of data to return)

Since **ib.reqHistoricalData()** returns a BarDataList, we use util.df() to convert it to a Pandas Dataframe. Then we format the columns so that Backtrader can properly understand it and we covert the date column to datetime objects and set it as the index. Note: Backtrader often bugs when timezones are included so we remove all timezones from the datetime objects.

Now that we have a dataframe we can backtest on, we need a strategy/algorithm we use to trade

6 Real-Time Execution with IB

Once a strategy has been developed and tested using historical data, the final step is to implement it in real-time. This allows the algorithm to monitor live markets and autonomously place trades when certain conditions are met.

In this chapter, we implement a real-time trading system using the Interactive Brokers API and the `ib_insync` library. The strategy will scan multiple tickers simultaneously, evaluate technical indicators, and place market orders based on live data feeds.

6.1 Overview of Real-Time Logic

A real-time algorithm needs to perform the following steps in a loop:

(1) Connect to IB Gateway (or TWS) using the appropriate port.

(2) Define all stock contracts to be monitored (e.g., SPY, AAPL, MSFT).

(3) For each stock:

- Request recent historical data (used as a proxy for live bars).
- Calculate indicators (EMA, MACD, RSI).
- Make a buy/sell/hold decision based on signal logic.
- Place orders as necessary.

(4) Wait for 60 seconds (1-minute bars), then repeat.

This loop continues as long as the script is running, allowing the system to execute trades dynamically throughout the day.

6.2 The Real-Time Trading Script

The full implementation for this real-time strategy is included below. It handles multiple stocks, checks indicator conditions, and uses basic risk management by capping exposure per stock.

```python
from ib_insync import *
import pandas as pd
import datetime
import time

ib = IB()
ib.connect('127.0.0.1', 7497, clientId=1)
```

```python

symbols = ['SPY', 'AAPL', 'MSFT']
contracts = {symbol: Stock(symbol, 'SMART', 'USD') for symbol in symbols}

for contract in contracts.values():
    ib.qualifyContracts(contract)

account_values = ib.accountSummary()
net_liq = float(next(row.value for row in account_values if row.tag == 'NetLiquidation'))
cash_at_risk = 0.1

last_action = {symbol: None for symbol in symbols}

while True:
    print("\n---", datetime.datetime.now(), "---")
    for symbol in symbols:
        contract = contracts[symbol]

        try:
            bars = ib.reqHistoricalData(
                contract,
                endDateTime='',
                durationStr='2 D',
                barSizeSetting='1 min',
                whatToShow='TRADES',
                useRTH=True,
                formatDate=1
            )

            df = util.df(bars)
            if df.empty or 'close' not in df:
                print(f"{symbol}: No data.")
                continue

            df['EMA20'] = df['close'].ewm(span=20).mean()
            df['RSI'] = df['close'].rolling(14).apply(
                lambda x: 100 - 100 / (1 + x.pct_change().mean() / x.pct_change().std()) if x.pct_change().std() else 50
            )
            df['MACD'] = df['close'].ewm(span=12).mean() - df['close'].ewm(span=26).mean()
```

```python
            df['Signal'] = df['MACD'].ewm(span=9).mean()

            latest = df.iloc[-1]
            price = latest['close']
            ema = latest['EMA20']
            rsi = latest['RSI']
            macd = latest['MACD']
            signal = latest['Signal']

            quantity = int((net_liq * cash_at_risk) // price)
            position = ib.positions()
            pos_qty = next((p.position for p in position if p.contract.symbol == symbol), 0)

            if price > ema and macd > signal and rsi < 70 and last_action[symbol] != 'buy':
                if pos_qty < 0:
                    ib.placeOrder(contract, MarketOrder('BUY', abs(pos_qty)))
                ib.placeOrder(contract, MarketOrder('BUY', quantity))
                last_action[symbol] = 'buy'
                print(f"{symbol}: BUY {quantity} @ {price}")

            elif price < ema and macd < signal and rsi > 30 and last_action[symbol] != 'sell':
                if pos_qty > 0:
                    ib.placeOrder(contract, MarketOrder('SELL', pos_qty))
                ib.placeOrder(contract, MarketOrder('SELL', quantity))
                last_action[symbol] = 'sell'
                print(f"{symbol}: SELL {quantity} @ {price}")

            else:
                print(f"{symbol}: HOLD | Price: {price:.2f}, EMA: {ema:.2f}, MACD: {macd:.2f}, Signal: {signal:.2f}, RSI: {rsi:.2f}")

        except Exception as e:
            print(f"{symbol}: Error fetching or processing data: {e}")

```

```
    time.sleep(60)
```

Explanation of Key Components:

Account and Risk Configuration

- We retrieve the account's Net Liquidation Value (NLV), which reflects total available capital.
- We allocate a fixed portion (e.g., 10%) to any single position. This avoids excessive risk.

Fetching Historical Bars

- We use `reqHistoricalData()` with a duration of 2 days and 1-minute bar size.
- This provides recent bars to calculate indicators, simulating a live feed.

Technical Indicators

- EMA20 is calculated using exponential weighting.
- RSI is manually estimated using a rolling formula for flexibility.
- MACD and signal line are approximated using standard 12-26-9 EMA rules.

Signal Generation and Order Logic

- If price > EMA and MACD > Signal and RSI < 70, we BUY (if not already long).
- If price < EMA and MACD < Signal and RSI > 30, we SELL (if not already short).
- The system also closes existing opposing positions to avoid holding both long and short.

Position Tracking

- We fetch existing positions using `ib.positions()`.
- This ensures we don't enter duplicate trades or exceed allocation limits.

Order Execution

- We use `MarketOrder()` to enter and exit positions immediately.
- All contracts are qualified before use with `ib.qualifyContracts()`.

Time Delay and Loop

- After each full iteration, we pause 60 seconds using `time.sleep(60)`.
- This ensures that each evaluation is synced to 1-minute intervals.

6.3 Important Considerations

Market Hours

Ensure the script is only run during regular trading hours (typically 9:30 AM to 4:00 PM EST). Data feeds and order placement may be restricted outside this window.

Error Handling and Logging

In production systems, logging trade activity and error messages is critical. Consider writing trade actions and indicator values to a file for later analysis.

Slippage and Latency

Market orders may not fill exactly at the observed price, especially during periods of high volatility. You can reduce slippage by adding limit/stop orders or building delay compensation logic.

7 Sentiment Analysis

In financial markets, price movements reflect the bets that investors place on the beliefs they hold about the market and the world at large. These beliefs are shaped by flows of public information, including news articles, analyst reports, and online discussions. Whether it be traditional media or social medias like Reddit and Twitter, financial narratives influence investor behavior, leading to changes in prices, volatility, and liquidity.

Although investor actions can sometimes appear irrational, they often follow patterns influenced by group sentiment, expectations, and social dynamics. Market trends often emerge when large groups of investors respond similarly to new information. These collective behaviors can be observed, interpreted, and, in some cases, predicted.

The rapid growth of real-time financial information available online has created new opportunities to analyze how language impacts markets. Most of this information is unstructured text, which poses challenges for extracting useful signals. Semantic analysis of financial news addresses this by identifying relevant content and linking it to market behavior. This technique has been applied to tasks such as predicting short-term returns, modeling risk, and developing trading strategies. However, challenges remain, including delayed market reactions, the complexity of financial language, and a high noise-to-signal ratio.

> **Todo.** Install the required Python packages: `undetected-chromedriver`, `beautifulsoup4`, `pandas`, `transformers`, and `torch`.

Tip 7.1 (Version Matching)**.** Change `version_main=138` when initializing `undetected-chromedriver` to match your Chrome browser version.

7.1 Web Scraping Financial News

Web scraping is the process of programmatically extracting information from web pages. It is a core component of many financial data pipelines, particularly when real-time or niche information is not available through formal APIs. In this project, we scrape financial news articles to analyze their semantic content and assess their potential market impact.

To accomplish this, we use `Selenium` to control a web browser and `BeautifulSoup` to parse the underlying HTML structure. Selenium provides a reliable way to interact with modern websites that rely on JavaScript for content rendering. Unlike simple HTTP requests made with libraries such as `requests`, Selenium executes scripts and simulates real user behavior, ensuring that the complete page content is loaded before parsing.

Some financial news platforms, such as `Investing.com`, include mechanisms to discourage automated access. These measures include JavaScript-based rendering, dynamic element loading, and bot detection services like Cloudflare. Attempting to access these pages using static HTTP methods often results in incomplete or blank responses.

Why Selenium and undetected-chromedriver?

Selenium launches a full browser instance, allowing the page to render JavaScript and behave as if a human user is browsing. When combined with `undetected-chromedriver`, a tool that helps bypass bot detection, we can use Selenium to scrape for dynamic and protected content.

Many financial news websites, including Investing.com use bot protection services, such as Cloudflare or PerimeterX, to block automated traffic. These systems can detect and block traditional scraping tools by analyzing browser fingerprinting, mouse movements, execution timing, and other behavioral cues.

When Selenium launches a browser in its default mode, it often exposes signs that it is being controlled by automation. As a result, websites may serve a CAPTCHA, display a "human verification" page, or block access entirely.

`undetected-chromedriver` is a patched version of ChromeDriver designed to avoid these detections. It modifies browser launch parameters and masks traces of automation, allowing the browser to appear more like a genuine user. This helps us bypass bot protection pages and access the full content of dynamically loaded, protected sites like `Investing.com`.

Once the page is fully rendered, we extract the relevant text using `BeautifulSoup`. This includes headlines, article bodies, timestamps, and metadata. The extracted data is then cleaned and stored for downstream analysis, such as sentiment scoring or linking to stock price movements.

Code Example: Scraping Investing.com

```python
from bs4 import BeautifulSoup
import undetected_chromedriver as uc
from selenium.webdriver.support import expected_conditions as EC

driver = uc.Chrome(headless=True, version_main=138)
url = "https://www.investing.com/equities/apple-computer-inc-news"
driver.get(url)
soup = BeautifulSoup(driver.page_source, "html.parser")
container = soup.select_one("div[class*='article_articlePage']")

```

```python
paragraphs = container.find_all('p') if container else []
driver.quit()
```

Note. Using (Cmd+Option+I) or (Ctrl+Shift+I), depending on your OS, inspect the page source to identify the CSS selectors (e.g., `article`, `a[data-test='article-title-`
that contain the information you want to extract.

7.2 Understanding FinBERT

FinBERT is a domain-specific language model fine-tuned on financial text to accurately capture sentiment within the context of markets, companies, and economic events. Unlike general-purpose language models, FinBERT is trained on corpora such as earnings reports, analyst commentary, and financial news articles, enabling it to understand the nuanced tone and specialized terminology frequently used in finance.

FinBERT classifies input text into three sentiment categories: **positive**, **negative**, and **neutral**. These categories are designed to reflect the perspective of a rational investor. For instance:

- **Positive** sentiment suggests favorable conditions or outlook, which can imply a buying opportunity.
- **Negative** sentiment corresponds to adverse news such as declining performance, regulatory challenges, or macroeconomic risks, often signaling caution or potential sell decisions.
- **Neutral** sentiment indicates balanced, inconclusive, or non-impactful information that is unlikely to influence immediate trading behavior.

FinBERT is widely applied in tasks such as sentiment scoring, risk monitoring, event-driven trading strategies, and market forecasting. Its ability to interpret textual information with financial context makes it a critical tool in modern quantitative finance pipelines.

Todo. Install FinBERT dependencies with `pip install transformers torch`

Sentiment Prediction Example

```python
from transformers import AutoTokenizer,
    AutoModelForSequenceClassification
from scipy.special import softmax

tokenizer =
    AutoTokenizer.from_pretrained("yiyanghkust/finbert-tone")
```

```python
model = AutoModelForSequenceClassification.from_pretrained("yiyanghkust/finbert-tone")

def predict(text):
    inputs = tokenizer(text, return_tensors="pt", truncation=True)
    logits = model(**inputs).logits.detach().numpy()[0]
    scores = softmax(logits)
    return {"positive": scores[0], "neutral": scores[1], "negative": scores[2]}

print(predict("Apple stock surged after beating earnings expectations."))
```

7.3 Combining Scraping with Sentiment Analysis

Once financial news articles are scraped and cleaned, we apply sentiment analysis to extract signals that may reflect investor reactions. Each article's title and body text are passed through FinBERT to obtain sentiment classifications: `positive`, `negative`, or `neutral`.

To quantify sentiment for downstream analysis, we convert these labels into numerical values. Specifically, a sentiment score is computed as:

$$\text{Sentiment Score} = \#\text{Positive} - \#\text{Negative} \tag{7.2}$$

Neutral classifications are excluded from this computation, as they are assumed to have no directional impact. This raw score reflects the overall tone of an individual article.

Daily Aggregation and Ticker Weighting

After computing sentiment scores for each article, we aggregate them at the daily level. This allows us to link sentiment to same-day or next-day market behavior. Aggregation is performed as a simple sum of sentiment scores across all articles published on a given date:

$$\text{Daily Sentiment}_t = \sum_{i=1}^{N_t} (\text{Sentiment Score}_i) \tag{7.3}$$

where N_t is the number of articles published on day t.

In order to focus on articles most relevant to a specific stock, we assign greater weight to news headlines that explicitly mention the stock ticker. For each article, we apply a binary weighting factor:

$$w_i = \begin{cases} \alpha > 1, & \text{if ticker is in the headline} \\ 1, & \text{otherwise} \end{cases} \tag{7.4}$$

This weighted sentiment score becomes:

$$\text{Weighted Sentiment Score}_i = w_i \cdot (\#\text{Positive} - \#\text{Negative}) \tag{7.5}$$

This approach prioritizes articles that are likely to be more influential, under the assumption that headlines are more salient to investors than body text.

Output Format

The final output is a table of daily sentiment scores for each stock of interest. Each row corresponds to a unique (ticker, date) pair and includes:

- The total number of articles scraped
- The weighted sentiment score
- Optional metadata such as average confidence or most frequent terms

These values can be joined with price data for further modeling, including return prediction, volatility estimation, or reinforcement learning-based trading.

Note. Headline weighting is particularly useful in high-volume news environments, where many articles mention market events without directly relating to a specific stock. Giving extra weight to headlines ensures the model focuses on the most relevant information.

```python
import pandas as pd

df = pd.DataFrame(data)
df['sentiment'] = df['title'].apply(lambda x: predict(x) if
↪ isinstance(x, str) else None)
```