TRADELAB

Summer 2019 Coding Project

Santosh Sivakumar
santosh.sivakumar.22@dartmouth.edu

GOAL

I am a trader interested in visualizing/evaluating the performance of a specific trading strategy on my portfolio. By understanding my strategy's capabilities and limitations, I hope to make an informed decision of how best to trade my stocks and implement my strategies. Beyond ways to shift my investments (strategic techniques), I hope to use historical data to better understand the characteristics that determine a financially successful portfolio.

TRADELAB

1. Creation/manipulation of portfolios.

   The interface allows for 2 broad types of portfolio creation: user-inputted and computer-generated.

   User-Inputted Portfolio

   TradeLab users have the opportunity to specify which stocks they'd like to invest in, and the corresponding dollar amounts they'd like to allocate to each stock. Given this information, the software runs a check to determine compatibility with an internal data set, and trades all stocks deemed compatible. This feature is most relevant for investors in specific industries (ie. technology/retail) who prefer to hand-pick stocks.

   Computer-Generated Portfolio

   For traders who desire a more robust/diverse portfolio, TradeLab offers the option of fully-automated portfolio generation, where the user can select a few specific parameters to guide automation.

   Portfolios are assembled using a simple K-Means clustering algorithm, where stocks are represented as Vectors (with each coordinate representing % change in share price over a week) and are clustered based on fluctuation patterns. Vector coordinates are determined from stock performance during a user-specified "portfolio initialization" window. From these clusters, the user has the option to choose all stocks within a cluster (uniform), or to fully diversify by picking a single stock from each cluster (diverse).

   Through this approach, back testers can visualize performance of trading strategy across a wide array of portfolios in order to gain a more holistic understanding of the capabilities of a specific software. This feature is most relevant for strategy developers.

2. Backtesting of trading strategies.

   TradeLab executes a user-selected trading strategy on a given portfolio across a user-specific time window, simulating all trades within the window as guided by the strategy. Current software supports 3 popular trading strategies (Mean Reversion, Moving-Day Average, Generalized Pairs-Trading), and the interface is extremely customizable to allow for a wide array of trading techniques (inspired by Quantopian).

   Beyond strategy execution, TradeLab provides useful metrics for evaluating portfolio performance, and more importantly, allows the software-savvy user to design/implement comparison metrics of their choosing. Current software compares strategy-driven portfolios with their untouched equivalents, but example APIs demonstrate other metrics to evaluate performance (comparison against S/P 500, against average of all portfolios with given strategy, etc).

3. Customization.

   TradeLab's software is designed to be user-friendly, most significantly through its' ability to be customized and personalized. Inspired by MATLAB, public classes and methods allow for easy creation of new strategies/portfolio generation techniques/performance evaluation metrics.
   Guidelines for customization are documented below.

Software Architecture

I: Testing/Visualization

TestClasses

Contains public methods to execute supported trading strategies on given portfolio, calling necessary methods to compute stock trades and evaluate portfolio performance.

Public methods: execute Mean-Reversion strategy, execute Moving-Day Average strategy, execute Cluster (Generalized Pairs-Trade) strategy, execute Pairs-Trading strategy

Extension: for user-created trading strategies, create method to take input portfolio/strategy dates/strategy parameters, output necessary data (currently: start/end portfolio value, cash)

```
public double [] executeSTRATEGYNAME (Portfolio port, String
      strategyStartDate, String strategyEndDate,DataCollection
      dc, DateModifications dm, OTHER PARAMETERS)
```

TestingServlet

Used to visualize software on local server (requires installation of Apache Tomcat Servlet interface). Main public method to extend HTTPServlet functionality.

Methods: doGet (override from HTTPServlet) to parse input data (from HTML interface), perform necessary computations (calling java classes of interest), render output HTML page (done through PrintWriter class).

Notes: very specific to user's visualization needs, can be circumvented through simple command-line interaction.

Auxiliary: HTML pages/ CSS files

Used to dynamically visualize software/prototype full-scale product. Can be customized.

Displayed in "Images" folder.

II: Portfolio Assembly

Portfolio

Basic class-representation of stock portfolio. Consists of list of constituent stocks/current number of shares held in each stock, as well as loose cash held by the portfolio owner (used for trading).

```
Portfolio (String inputName, Double inputCash, ArrayList<String>
      inputStocksInPortfolio, ArrayList<Double>
      inputValuesInPortfolio)
```

Notes: additional object-specific variables can be created to store necessary information (ie. $ amounts invested in each stock, specific stock characteristics (volatility, % to buy/sell).

PortfolioMaker

Public class to assemble portfolios based on user-specified parameters. Implements computer-generated and user-generated portfolio construction methods, calling auxiliary classes/private helper methods for parsing/clustering of stock vectors.

Generation methods:
diverseHAC/diverseKMeans/uniformHAC/uniformKMeans/UserInput to assemble portfolio of interest. For computer-generated (all except UserInput), method creates stock vectors from file containing stock names/prices, calls corresponding clustering algorithm (KMeans/Hierarchical Agglomerative Clustering), retrieves clusters. For diverse portfolios, one stock is selected from each cluster/for uniform, all stocks from one cluster become portfolio.

Notes: HAC methods are currently not implemented on final software due to clustering algorithm's weak resistance to outliers. For computer generated portfolios, cash amount is split evenly across all stocks (for each amount correspondingly governing a different number of shares); for user-inputted portfolios, all stocks are subject to validity checks during time-frame/if stock doesn't pass, it is scrapped from the portfolio.

Private helper methods:

createVectors assembles a vector list of all stocks in a given file, calling createVector for each stock and checking for vectorValidity before adding.

createVector creates a vector for a specific stock, parsing stock's data at weekly intervals to compute % change in share price. % changes become coordinate in stock vector.

Note: method can be reconfigured based on desired qualities used to cluster stocks (ie. name, average share price, etc)

vectorValidity checks stock data against DC (DataCollection) Map of all trading days in data set. If stock data has not been tabulated on a known trading day, stock is scrapped.
Note: method currently does NOT extrapolate data if not present; can be reconfigured.

Extension: portfolio-generation methods can be easily created using existing helper methods and appropriate helper classes.

```
public Portfolio PortfolioName (String filename, DataCollection
       dc, DateModifications dm, ArrayList<String> vectorNames,
       OTHER PARAMETERS)
```

III: Clustering Algorithms for Computer-Generated Portfolios

KMeans

Public class to assemble + return list of clustered vectors, organized based on closest euclidian distance to mean, with number of means specified in signature.

Public Cluster method takes in List of vectors as input, assigns *k* arbitrary means, groups all vectors into *k* clusters around means, recomputes/reclusters means repeatedly. Clusters are returned as an ArrayList of ArrayLists of Vectors.

```
public ArrayList<ArrayList<Vector>> cluster (ArrayList<Vector>
      vectors, int numClusters)
```

Private methods: initialize *k* empty clusters, determine whether clustering is finished, identify closest cluster, compute midpoint of cluster. All methods are evoked during individual rounds of clustering.

Notes: cluster termination method relies on a hard-coded threshold. Normalized distance between cluster centers are computed, and clustering is only determined to be finished if distances are all less than normalized threshold.

Drawbacks: algorithm may not be fully-outlier resistant, and uniform portfolios with specific sizes are often not achieved due to unpredictability of cluster sizes.

HAC (Hierarchical Agglomerative Clustering)

Two approaches: one exclusively uses HAC algorithm to assemble and return cluster, other integrates HAC with K-Means for increased flexibility with user parameters. Both ultimately implement a bottom-up approach to identify nearest vectors + assemble clusters in a binary tree.

Public methods:

assembleClusters makes use of KMeans integration, performing KMeans clustering on a specified number of subclusters (each of which is assembled using HAC); clusters are outputted as an ArrayList of ArrayList of Vectors. Method takes in vector list, desired cluster size, distortion factor. Relies extensively on private methods to neatly assemble clusters of given size.

```
public ArrayList<ArrayList<Vector>> assembleClusters
      (ArrayList<Vector> vectors, int clusterSize, double factor)
```

sortClusters exclusively uses HAC, takes in vectors as input, performs clustering operations as necessary, returns a Map of vectors to cluster IDs for further manipulation.

Used in conjunction with ArrayListOfClusters, which takes in sorted clusters and returns clusters in ArrayList format.

```
public HashMap<Vector, Integer> sortClusters (ArrayList<Vector>
       vectors, int numClusters)

public ArrayList<ArrayList<Vector>> ArrayListOfClusters
       (HashMap<Vector,Integer> clusterHashMap, int numClusters)
```

Private methods: numerous private methods to assemble clusters from sorted HAC/KMeans groupings. Most significant private method works downward in binary tree until subtree (cluster) of desired size is identified. Full documentation of private methods can be explored using individual method docstrings.

Note: HAC is NOT implemented in final version of software, due to limited resistance against outlier vectors. It was frequently noted that vectors often naturally grouped into large clusters with a majority of vector, and numerous 1-vector clusters as driven by extreme data. Even when coupled with KMeans clustering, resultant clusters varied greatly in size / made for unusable portfolios.

Auxiliary: Vector

Simple class to implement a vector for a given stock. Contains instance variables for stock name, list of coordinates (represented as doubles). Can be changed as necessary.

```
Vector (String inputName, ArrayList<Double> inputNumbers)
```

Class methods: magnitude computes vector magnitude summed across all coordinates, distance returns Euclidian distance between two vectors (can be changed if desired).

Note: distance method requires two vectors of equal # of dimensions.

IV: Trading Strategies

Mean-Reversion Strategy

At a high-level, the mean reversion strategy optimizes for fluctuations in share price by expecting the price to "revert to the mean". Using this expectation, shares are sold if a stock's share price exists above the mean (past a specified threshold) and are bought if a stock's price is below the mean (past threshold).

Public methods: Initialize method establishes "mean" value for each stock on initial day of strategy, for comparison during execution period. AllUpdates method executes strategy on each trading day within given time frame.

```
public void initialize (Portfolio port, DataCollection dc,
       DateModifications dm, String date)

public void allUpdates (Portfolio port, DataCollection dc,
       DateModifications dm, String startDate, String endDate)
```

Private methods: methods to identify current share price's position relative to mean, determine whether to buy/sell shares, execute trades. All sales on a given trading are executed prior to any purchases, to maximize funds available for purchase.

```
MeanReversionStrategy (String inputName, double
       inputThresholdPercentage, double inputVolatilePercentage)
```

Where thresholdPercentage corresponds to the % beyond which stock is to be sold/bought, and volatilePercentage corresponds to the % to sell.

Note: class requires fully assembled portfolio, user-specified parameters.

Moving-Day Average (MDA) Strategy

At a high-level, this strategy compares short and long-run trends for share price movement to determine patterns to exploit. On the first day a local (short-run) average exceeds a long-run one, the MDA strategy sells shares in the corresponding stock. If the short-run price average falls below the long-run average, the strategy buys shares, following the "current trend".

Public methods: Initialize method checks for date/portfolio validity. AllUpdates method executes strategy on each trading day within given time frame.

```
public void initialize (Portfolio port, DataCollection dc, String
       startDate, DateModifications dm)

public void allUpdates (Portfolio port, DataCollection dc,
       DateModifications dm, String startDate, String endDate)
```

Private methods: compute short and long-run average prices, determine stock's position re. buy/sell, execute necessary operations. All sales on a given trading are executed prior to any purchases, to maximize funds available for purchase.

```
MDAStrategy (String inputName, int inputLongRunFrameFrame, int
        inputShortRunFrameFrame, double inputVolatilePercentage)
```

Where long/short-run frames correspond to the number of days within each period (example: LR = 200, SR = 50), and volatilePercentage corresponds to the % to sell.

Note: class requires fully assembled portfolio, user-specified parameters.

Cluster (Generalized Pairs) Strategy

At a high-level, the strategy is intended to act on a uniform portfolio (stocks of a similar pattern). After computing an expected average across all stocks within the portfolio, day-to-day trades are executed by comparing individual stocks to the "cluster average", in order to take advantage of the average ratio-regressive nature of the portfolio.

Public methods: AllUpdates method executes strategy on each trading day within given time frame.

```
public void allUpdates (Portfolio port, DataCollection dc,
        DateModifications dm, String startDate, String endDate)
```

Private methods: compute current mean/standard deviation across all stocks, compute (for each stock) z-score relative to average statistics, determine whether to buy/sell shares, execute trades. All sales on a given trading are executed prior to any purchases, to maximize funds available for purchase.

```
ClusterTradingStrategy (String inputName, Double
        inputThresholdNumSD, Double inputVolatilePercentage)
```

Where thresholdNumSD corresponds to the number of Standard Deviations beyond which stock is to be sold/bought, and volatilePercentage corresponds to the % to sell.

Note: class requires fully assembled portfolio, user-specified parameters.

Pairs Trading Strategy

At a high-level, the strategy takes advantage of two similar stocks and computes a working ratio between the share prices of both stocks. On a given trading day, the strategy determines the current ratio's position relative to the expected ratio, and covers both bases by selling in a way to optimize for both events that may have led to observed ratio. For example, if the current ratio exceeds what is expected, strategy sells shares in

stock in numerator of ratio/buys shares in denominator of ratio, with no determination of events causing observed ratio.

Public methods: Initialize method establishes “expected” ratio between pair of stocks across initialization period, for comparison during execution period. AllUpdates method executes strategy on each trading day within given time frame.

```
public void initialize (Portfolio port, DataCollection dc,
      DateModifications dm, String startDate, String endDate)

public void allUpdates (Portfolio port, DataCollection dc,
      DateModifications dm, String startDate, String endDate)
```

Private methods: rebalance portfolio on given day (sell/buy appropriately).

```
PairTradingStrategy (double thresholdPercentage, double
      volatilePercentage)
```

Where thresholdPercentage corresponds to the % beyond which stock is to be sold/bought, and volatilePercentage corresponds to the % to sell.

Notes: class requires portfolio of TWO stocks.

Implementation note: Strategy NOT included in final implementation of software due to highly-specific portfolio constraints [two stocks, preferably similar].

V: Helper Classes

DataCollection

Public class to retrieve stock/trading day data from given files, determine portfolio/vector validity, compute portfolio values. Primarily used in interaction with stock database.

Public methods: readData parses data from input file, creates extensive database of stock names/dates and share prices (format: stockID = name + "#" + date), database of all known trading days (window: 1962-2017). checkIfPortfolioDataExists determines whether all stocks in portfolio have been recorded on specific trading day (safety precaution); checkIfVectorDatExists checks for data point for specific vector, specific date. vectorNamesList interacts with input file to create an ArrayList of all known stock vectors in data set. valueOfPortfolio computes summed value of all shares in portfolio on a given date (useful in performance evaluation).

```
public void readData (String stockFile, String tradingDayFile)

public ArrayList<String> vectorNamesList (String filename)

public boolean checkIfVectorDataExists (String stockName, String
       stockDate)
public boolean checkIfPortfolioDataExists (Portfolio port, String
       stockDate)

public double valueOfPortfolio (Portfolio port,String priceDate,
       DateModifications dm)
```

Note: the DataCollection class is used extensively across other classes, but only one instance is to be created and passed around (for large data files, creating database is very time/memory intensive).

DateModifications

Public class to perform simple manipulations on dates (Strings).

Public methods: increment/decrement dates in accordance with month lengths, leap years.

```
public String incrementDate (String date)

public String decrementDate (String date)
```

Private methods: public increment/decrement methods make use of smaller private methods to interpret given dates. Full documentation in docstrings.

Value

Public class to represent share information on a given day.

```
Value(double inputOpen, double inputClose, double inputVolume)
```

Where “open” is stock opening price, “close” closing price, “volume” is volume of shares sold.

Note: for TradeLab software, closing prices were used.

Usage

Through TradeLab's user-interface, traders can choose a combination of portfolio creation/trading strategy they'd like to implement, provide parameters as necessary, and execute their trades within a desired time frame.

Parameters are retrieved by the servlet (if used), necessary classes are called to manipulate/access data as necessary, and details are rendered in a user-friendly HTML format.

Beyond simple backtesting, the software is designed to be highly customizable (*a la* MATLAB) for the software-familiar trader. A few examples of ways to do this:

> As a trader, I'm very interested in not only seeing how my strategy does compared to null, but also to compare my portfolio against some benchmarks (S&P 500, NASDAQ 100, etc). So, I design a public method (nested within TestClasses) to do just this.
>
> ```
> public void compareWithNASDAQ100 (String strategyStartDate,
>       String strategyEndDate, STRATEGYPARAMETERS)
> ```
>
> Assuming I've created an auxiliary method to initialize a NASDAQ portfolio (which I could manually do as well through TradeLab's user-input option), I simple execute my strategy of interest on this portfolio + output its performance alongside that of my portfolio.

> Maybe I'm not satisfied with the current trading strategies supported, and I want to create one of my own. Admittedly, while writing this I have limited knowledge of many other trading strategies, but for the purposes of this demonstration I'll make one up.
>
> My desired strategy: when a stock crosses a certain threshold (relative to initialization share price), I act accordingly, and perform the opposite operation *n* days later. For example, my stock rises 20% above the threshold-dictated limit; my strategy should sell now and buy in 10 (*n*) days.
>
> A few signatures of constructor/methods I could create:
>
> ```
> TestStrategy (String inputName, Double inputNumDays, Double
> inputThresholdPercentage, Double inputVolatilePercentage,
> HashMap<String,String> lastTradingDay)
>
> private boolean isSellDate(String stockName, String date)
>
> private void update (Portfolio port, String stockDate, DataCollection dc,
> DateModifications dm)
> ```

Notes / Musings

I chose to do this project in java for two reasons. One was to become more comfortable with the language itself (until now I had exclusively worked in Python), and the second was because of how object-oriented I discovered this simulation to be. Creating separate classes for different portfolios/strategies/etc. not only introduced me to the beauty of OOP but also made for (relatively) easy-to-organize software. This project also tracked my growth/comfort with java across the summer, as illustrated in the wide diversity of coding syntax/level of adherence to stylistic convention.

HTML/CSS code are primarily from a bootstrap template, courtesy of ColorLib. Tomcat's servlet interface didn't allow me to do much re. integrating data with HTML in the "results" section, hence the messy TestingServlet code. My preferred mode of usage is the command-line interface, but I do recognize the utility/beauty of a workable HTML interface.

It was interesting to note that HAC clustering was so error-prone with outlier stocks. Even after integrating with KMeans, I frequently found clusters of size 0 alongside others of size 500. Still looking for a way to avoid this (though, admittedly, it might just be a reflection of the diverse nature of stocks). In the same vein, I'm currently developing an algorithm for Gaussian Mixture Modeling-based clustering.

From my trials (mostly for debugging purposes), I found it interesting (though most familiar with the field probably won't) how large of a difference strategic trading makes. There were certain test-cases, on pretty robust/high cash portfolios, where implementing a strategy led to a ~500% increase in shareholder profits compared to no action. Interesting firsthand way of understanding the profitability of the trading industry!

If nothing else, testing different combinations within the software makes for a great data-mining pastime!