CSCI 3901 Final Project

Due date: 11:59pm Monday, April 15, 2024 in git.cs.dal.ca. The repository is at https://git.cs.dal.ca/courses/2024-2inter/csci-3901/project/xxx.git where "xxx" should be replaced by your timberlea login id. The repository is already created for you, so you should just need to clone it.

Goal

The course project is your opportunity to demonstrate all of the concepts from the course in one body of work.

Project Structure

The final project will have you apply your problem solving and development skills. The solution will require you to bring together

- Abstract data types and data structures
- Java implementation
- Basic algorithms
- Software development techniques including version control, testing, debugging, and defensive programming
- Good software program design
- Database design and use

This project is _not_ expected to include a user interface component or software that directly accesses a device's hardware.

Problem

Financial advisors work in investment firms to help individuals manage savings. We will focus on a scenario where the financial advisors and investment firms are using the stock market for investments.

Individuals register with an investment firm and are assigned a financial advisor. The financial advisor works with the individuals to understand their needs and tolerance for risk and ultimately assign the individual an investment profile, like a conservative investor, a moderate investor, an aggressive investor, and so on. The profiles then recommend specific mixes of stock investments.

Analysists classify company stocks into investment **characteristics** like large-cap, small-cap, and fixed income and **sectors** like finance, energy, and consumer disposables. An investor profile then outlines a percentage of investments to hold in each characteristic. For example, a profile might specify 65% large-cap, 15% mid-cap, 10% small-cap, and 10% fixed income. It might also recommend sectors like 10% financial sector stocks and 15% energy sector stocks. In this project we will only deal with sectors and not with investment characteristics.

Companies periodically issue dividends to their shareholders. For example, a company might provide 50 cents for each share, so someone who holds 100 shares would receive $50. Depending on the client preference, the investment firm might then take that dividend and buy more stock in that same company with the dividend. The wrinkle here is that the dividend rarely matches the cost of the company stock to be able to buy an integer number of shares. Consequently, the investment firm internally manages its own system where it has some shares of its own and tracks fractions of shares to investors itself. For example, it might hold 1 share in a company and internally remember that Alice holds .25 units of that share, Bob holds .45 units of that share, and Claire holds .30 units of that share. Once Alice has enough fractions of share to make a single share, the company transfers one of its full shares into Alice's investment portfolio.

The project

Create a class called "InvestmentFirm" that an investment firm will use to manage its investors' investments. The class can use internal data structures, databases, or files to store information.

The information managed by the class should survive between execution of programs that use the InvestmentFirm class.

Here is the minimum information that you will need for the project.

Ultimately, we want you to answer the following questions for the investment firm:

- What is the proportion of holdings in each investment sector for an investor?
- Which financial advisors create similar investment portfolios?
- Which investors have holdings that differ a lot from their stated investment profiles?
- Which stocks would you recommend that investor hold in comparison with other similar investors?

Your task is to create classes that will gather the information for this system and resolve these queries.

Methods for InvestmentFirm

These method definitions intentionally avoid indicating any return structure unless it's key to the method. Choosing what the methods return and how it is returned is part of the project's design element around obtaining usable and robust code.

Getting data into the system

defineSector(String sectorName)

Declare that the given sectorName is one of the investment sectors to which a stock can belong and, as a consequence, be one of the sectors listed as part of an investment profile. We would typically define the sectors as the investment system is first created; investment sectors rarely change over time.

Typical examples of sectors are energy, materials, industrials, consumer discretionary, consumer staples, health care, financials, information technology, telecommunication services, utilities, and real estate.

defineStock(String companyName, String stockSymbol, String sector)

Declare that the given companyName will be one that an investor an buy into. The company will be identified on the stock exchange by the given stockSymbol and the company belongs to the identified investment sector. We only declare the stock as the company is admitted to the stock exchange, so it happens infrequently.

setStockPrice(String stockSymbol, double perSharePrice)

Identify that the company with the given stockSymbol has just traded at the give perSharePrice value, meaning that one share of the stock was bought or sold at that per share price. This perSharePrice is the current understanding of the value of one share of the company.

We would expect setStockPrice to be called on every trade of the stock exchange for the company, which means it will be called frequently each day.

defineProfile(String profileName, Map<String, Integer> sectorHoldings)

Create an investment profile, to be identified by profileName, that reflects how much value (in dollars) that an investment portfolio should hold in each of a given set of sectors. The sectorHoldings parameter identifies the holdings per sector: the Map key is the sector name and the value is an integer that, when divided by 100, gives the percentage of stock value to hold in that sector. A sector that isn't in the map would have a holding of 0% in the portfolio.

For example, a map with values
"energy" -> 25
"health care" -> 20
"information technology" -> 55

Means that the portfolio should have 25% of its value in energy stocks, 20% of its value in health care stocks, and 55% of its value in information technology stocks.

There should always be a "cash" sector available to the profile specification, identified by a "cash" sector name and representing a cash holding in a portfolio.

int addAdvisor(String advisorName)

Declare that advisorName is a financial advisor working for the firm. That advisor will then be assigned clients.

Return an identifier for the financial advisor to be used in other methods.

int addClient (String clientName)

Declare that clientName is a client of the investment firm. The client will late be associated with one or more investment accounts.

Return an identifier for the client to be used in other methods.

int createAccount(int clientId, int financialAdvisor, String accountName, String profileType, boolean reinvest)

Declare that the client identified by clientId is opening a new investment account. That investment account will be managed by financialAdvisor and, when talking with the client, it should be identified to them with the given accountName like “RRSP”, “TFSP”, or “house fund”.

The mix of investments in the account will aim to meet the investment profile identified by the profileType name; that profile will have been created by the defineProfile() method.

Stocks in the investment account may issue dividends. If the “reinvest” parameter is true then the disburseDividend() method (described later) will use the dividends to buy more stock in the company. If the “reinvest” parameter is false then the dividents will accumulate in the cash balance for the account.

Return an identified for the investment account to be used in other methods.

tradeShares(int account, String stockSymbol, int sharesExchanged)

Consider the investment account identified by the “account” parameter. This method either buys or sells shares in the company identified by stockSymbol. The number of shares traded is given by sharesExchanged. If sharesExchanged is positive then you are buying the shares for the account. If sharesExchanged is negative then you are selling the shares for the account.

The value of each share is whatever the last recorded trade value is with the setStockPrice() method; if no trade has ever happened then apply a default share price of $1.

If you are buying shares then the value of those shares must already be in the account as cash for the trade to succeed. If you are selling shares then the proceeds of selling the shares will be added to the cash balance in the account.

As a special case, if stockSymbol is “cash”, then sharesExchanged is a transfer of cash into or out of the account (positive value means cash into the account and negative value means cash out of the account). In this case, sharesExchanged is the same as the dollar amount, as if the share cost is $1 per share. The cash balance for an account should never go below zero.

changeAdvisor(int accountId, int newAdvisorId)

Record that the investment account identified by accountId will now be managed by a new financial advisor in the company, who is identified by the newAdvisorId parameter.

Reporting on the system

double accountValue(int accountId)

For the investment account identified by accountId, return the market value of the account.

The market value of one company’s investment is the number of shares of that company in the account times the value of each share. The market value of the investment account is then the sum of the market values of the companies in the account along with the cash balance in the account.

double advisorPortfolioValue(int advisorId)

For the given financial advisor, report the market value that the advisor is managing for the company.

The value of the advisor’s portfolio is the market value of all accounts managed by the financial advisor.

Map<Integer, Double> investorProfit(int clientId)

Report how much profit client would make if that client sold all of their stocks at the current market prices (from setStockPrice()) in all of their investment accounts. Cash balances in the investment accounts is not considered as part of the profit.

The profit from selling one company’s stock is the current market value of the company’s stock less the average cost base (ACB) for the stock in the investment account. The ACB is basically what you paid for the stock when you bought it. Each time you buy stock, what you paid for the stock is added to the ACB. Each time you sell stock, you decrease the ACB by a proportion corresponding to the number of shares sold.

For example, if we buy 10 shares of company A at $10 per share (total cost of $100) and another set of 10 shares of company A at $15 per share (total cost of $150) then the ACB is $250 over the 20 shares. If we then sell 5 shares then we are selling 5/20 = ¼ of our shares, so the ACB becomes $250 * 3/4 = $187.50

Map<String, Integer> profileSectorWeights(int accountId)

For the given account, compute what proportion of the account value is held in stocks of each of the sector types. The value is from the market value of each stock held that is associated with the given sector.

In the returned Map, the Map key is the sector name and the Integer value is the percentage of value held in that sector times 100. Round the holdings to the nearest integer.

If an account held $1000 in the finance sector, $500 in the consumer staples sector, and had $250 of cash (total account value of $1750) then you would return a map with
"finance" -> 57
"consumer staples" -> 29
"cash" -> 14

You can opt to also return sectors with no holdings in the account, having a percentage of 0%.

Set<Integer> divergentAccounts(int tolerance)

Each investment account has an investment profile that outlines what percentage of stock should be held in each sector. As the market value changes and as we buy and sell stock in the account, our holdings in each sector changes. Despite these changes, the financial advisors are expected to keep the value of each investment account near the profile levels.

The divergentAccounts method reports all of the investment account ids whose sector distributions differ significantly from their target profile. The sector distributions differ significantly if one or more sector, including cash, has a percentage value (as in profileSectorWeights) that is too far away from the profile's target percentage. "Too far away" is defined by the "tolerance" parameter to the method; that tolerance, when divided by 100, is the percentage by which we can deviate.

For example, if "tolerance" is 5 and an account's profile says to have 50% financials and 40% energy, and 10% cash, then report the account if any of the following conditions apply:

- The financials holdings in the account are more than 55% of the holdings or less than 45% of the account value
- The energy holdings in the account are more than 45% or less than 35% of the account value
- The cash holdings in the account are more than 15% or less than 5% of the account value

int disburseDividend(String stockSymbol, double dividendPerShare)

The company idenfied by the given stockSymbol just declared dividends of "dividendsPerShare" for each share. This method disburses these dividends to each account that we know about that holds shares in the given company.

If the investment account is not marked for reinvestment then the dividends add to the cash balance of the account.

If the investment account is marked for reinvestment then we take the value of the dividends and buy shares in the company at that value (share value as last seen by setStockPrice()).

However, there is a wrinkle to this operation. The stock market only operates on integer numbers of shares. Meanwhile, dividends rarely equate to an even number of shares for the company. For example, company A might issue a dividend that amounts to $110 for one account. Meanwhile, the share price for company A is $100. Consequently, the account should get 1.1 shares. However, the stock market will only sell 1 share or 2 shares. The investment firm will intervene to manage the extra 0.1 shares.

The investment firm intervenes as follows. If an account X needs 0.1 shares then the firm will buy a full share and track that the account owns 0.1 parts of it and leaving 0.9 parts owned by the firm. If a second account Y needs 0.2 shares of that same stock then the firm will give 0.2 shares to account Y and will now just hold 0.7 parts of the share.

Before disbursing the dividends, the firm will hold some number of shares S in the company. After disbursing the dividends, the firm will need to be holding some other number of shares T in the company. The method returns the number of shares that the company has had to purchase to manage the dividends, namely T-S. If disbursing the dividends lets the company avoid holding more shares on its own then it will return a negative number of shares.

Any fraction of a share below 0.0001 is to be rounded up to 0.0001 or down to 0.

Analyzing the system

The following methods do a bit of analysis. We will seek to compare how similar two sector distributions might be or how similar two stock portfolios might be. These comparisons will happen with a cosine similarity measure[1].

Let's use a comparison of sector distributions as an example of the concept behind the cosine similarity measure. Create a vector that captures every sector. Vector coordinate 1 will be the percentage holdings in the finance sector, coordinate 2 will be the percentage holdings in the energy sector, and so on. Each investment account then has one vector representing all of the sector weights. We can view this vector as belonging to a high-dimensional space. Given the vectors for two accounts, we can look at the angle between these two vectors in high-dimensional space. If the angle between the vectors is close to 0 then the vectors are similar. If the angle is large then the vectors are less similar.

[1] Wikipedia, Cosine Similarity, https://en.wikipedia.org/wiki/Cosine_similarity, accessed March 20, 2024.

Computing the angle between high-dimensional vectors may not be convenient. However, we can calculate the cosine of the angle easily. If that cosine is close to 1 then the angle is near 0 and the vectors are similar. As the cosine gets closer to -1, the angle is larger and the vectors are increasingly dissimilar.

How do we compute the cosine "easily"? We recall the following formula for the dot product of vectors a and b: $\mathbf{a} \bullet \mathbf{b} = ||\mathbf{a}|| * ||\mathbf{b}|| * \cos(\theta)$
Where $||\mathbf{a}||$ is the length of vector **a**, $||\mathbf{b}||$ is the length of vector **b**, and θ is the angle between **a** and **b**. Rearranging the formula gives
$\cos(\theta) = \mathbf{a} \bullet \mathbf{b} / ||\mathbf{a}|| * ||\mathbf{b}||$

We have a quick calculation for the dot product: if **a** = (a_1, a_2, a_3, …) and **b** = (b_1, b_2, b_3, …) then the dot product is $a_1b_1 + a_2b_2 + a_3b_3 + \ldots$ meanwhile, the length of a is square_root($a_1^2 + a_2^2 + a_3^2 + \ldots$) Consequently, we have a quick formula to get the cosine value from our distribution vectors.

Map<String, Boolean> stockRecommendations(int accountId, int maxRecommendations, int numComparators)

This method seeks to recommend stocks to buy or sell for a given account. It will return up to maxRecommendations in total. The recommendations are returned in a Map, with the stock to recommend as the key and the value as True if there is a recommendation to buy it and as False if there is a recommendation to sell it. Let's examine how we find the recommendations.

For each investment account in the system, create a vector of all the stock holdings in the account. Next, find the "numComparators" other investment accounts whose cosine similarity measure to the the given account (identified by the accountId parameter) is closest to 1. Call this set of accounts S. Then S would be accounts that have the most similar stock holdings to accountId.

If accountId has stock x and the majority of accounts in S do not have stock x then you will recommend to sell x. If accountID does not have stock x and the majority of accounts in S have stock x then recommend to buy x.

Since you are limited in recommending "maxRecommendations" buys or sells, prefer to recommend buys or sells where there is the bigger majority. In the case of tied majority, recommend for buys) the stocks that have the bigger total investment from the accounts in S and recommend for sells the stocks in which accountId has the biggest holding.

For example, suppose that numComparators is 10. If accountId holds stock ABC and 7 of the 10 comparators in S do not hold stock ABC then we will recommend to sell stock ABC. If, instead, 6 of the 10 comparators in S hold stock ABC then stock ABC is not part of our recommendation.

Similarly, suppose that accountId holds neither stocks ABC and XYZ, with 7 of the 10 comparators holding stock ABC and 8 of the 10 comparators holding stock XYZ then we will report stock XYZ to buy before recommending stock ABC.

Set<Set<Integer>> advisorGroups(double tolerance, int maxGroups)

Investment accounts will vary a bit from their ideal sector profiles. Different financial advisors will have preferences on how the weights in each sector will vary. One advisor may have you invest more heavily in the information technology sector and less in the real estate sector while another advisor may have you invest more in finance and less in energy.

advisorGroups finds advisors with similar preferences. Suppose that advisors 1, 3, and 8 have similar preferences and advisors 2 and 5 have similar preferences, then the method returns { {1, 3, 8}, {2, 5} } as the two groupings of similar advisors with the advisors ids as the inner sets.

The questions are then: how do we identify similar advisors and how many groups of preferences might there be? We will identify similar advisors with k-means clustering[2].

k-means clustering will work as follows. Suppose that we know we are looking for k groups of preferences (clusters).

1. Extract the percentage weights of each sector for each account as vectors (as described for the cosine similarity measure).
2. Create k sector vectors with random values in the coordinates (though in the range of percentages). Call these vectors of random values the cluster representatives.
3. Associate each account with the cluster representative that is closest, by cosine similarity measure, to the account's sector vectors.
4. Recalculate each cluster representative as the average of all vectors associated with the cluster
5. Calculate the cosine similarity measure of each account sector vector to its cluster representative
6. If some distance to the cluster representatives is greater than a tolerance level and if at least one sector account vector is now in a new sector then go to step 3. Otherwise, end.

These steps assume that we know the number of clusters k. Here, we don't. So, we will start by making just 1 cluster. If k-means exits with all distances below the tolerance then we report 1 cluster. Otherwise, we re-run the cluster algorithm with 2 clusters. Again, if k-means exits with all distances below the tolerance then we report 2 clusters. Otherwise we try again with 3 clusters and continue until we try with maxGroups. If we reach maxGroups then we report whatever cluster division arises from those maxGroups.

The algorithm described relates to investment accounts while the method wants you to return groups of financial advisors. Given the clusters of accounts, consider the advisors of the accounts in one cluster as similar. In this instance, it is possible that one financial advisor may

[2] Wikipedia, K-means clustering, https://en.wikipedia.org/wiki/K-means_clustering, accessed March 20, 2024.

be reported in more than one cluster, and that's ok…it just means that that advisor may not have one strict preference.

Assumptions

- All individual names will be unique. You don't have enough information on people to distinguish one from another, so assume that the same name represents the same person.

Notes

- Do your analysis of the problem before starting to code. Although it there are many methods in the project, many of them can be done with SQL statements, which makes them short to write.
- The project description does not pull all the information that the project manages into a single list for you. You will want to read through all the functionality and accumulate your list of data needed to do the tasks before designing your database or data structures. Note that an inefficient design would try to read the entire contents of the database into memory data structures before doing work. Find a balance between work in the database and work in data structures.
- Include the type of return values and exception handling that you feel is most appropriate for your circumstances. Document that exception handling mode and be consistent with it.

Milestones

Only the final submission is graded. The intermediate milestones appear as points to get feedback on the project:

- April 1: High-level breakdown analysis of the problem
- April 5: Blackbox tests and plan+timeline of feature development
- April 15: Project due

All milestone material is due in the course git repository.

Marking scheme

- Meeting intermediate milestones (10%)
- External documentation (10%)
- Overall design and coding style (15%)
    - Design, including the quality and consistency of the decisions being made in the design 10%
    - Coding style 5%
- Working implementation, including efficient processing (50%)
    - Data entry Methods 10%
    - Reporting methods 15%
    - Analysis methods 15%
    - Robustness and error reporting 5%

        - Efficiency 5%
    - Test plan (15%)