CS 5820 Semester Project Report

Nutrition Tracker

Fall 2022

Group 22

Affan Moyeed, Matt Fackler, Prashan Thapa

Abstract

Our task was to design, implement and evaluate an intelligent agent that performed a simple task. The application we designed accepts user input in the form of barcodes and tracks how much of specific nutrients the user has consumed. It then makes suggestions on foods the user can eat based on their previous eating habits, so they can make up any nutrients the user has not consumed that the FDA recommends. To accomplish this task, we set up what we called rational variables that determine frequency, infrequency, and what percent of the FDA's daily value was considered high in content. We originally thought that four days was frequent and thirty percent of the FDA's daily value was high in content. We ran some tests on foods we knew were high in some nutrients and found we overestimated values as the application wasn't giving us correct recommendations on food we knew was sufficient. We eventually decided that eating something three times was considered frequent, not eating something for two days was infrequent, and about fifteen percent of the FDA's daily value is considered high content. These values helped us create more accurate recommendations and provide a better user experience. We also provided a list of improvements we could implement to increase general cosmetic appeal and increase algorithm complexity for better recommendations.

Table Of Contents

Introduction

Our aim for this project was to design, implement, and evaluate an intelligent agent that could either pass the Turing Test or perform some specific intelligent task. In our case, we specifically designed the application to perform the task of tracking a user's nutrient intake and making suggestions based on insufficiencies.

We have created a fully functional 'Nutrient Tracker' program, written in Python 3.10. We have utilized a database called OpenFoodFacts for our food items since it contains components, allergens, nutrition information, and all the other information we may need. We can extrapolate all this info from the unique product labels on each food item.

We also made our own custom database using CSV files based on data from the FDA. This data is eventually used to judge whether the user has consumed enough nutrients, what they need more of, what they can eat, and if they have something they like that they frequently eat. The use of CSV files over a database was purposeful for the use in demonstration and implementation simplicity.

The following report will give a full summary of our applications purpose, development, AI implementation, functionalities, weaknesses, and possible improvements. We will start by exploring the application's frontend and backend elements up close. We will talk about its environment, user interface, functionalities and methods. Then, we will go into both a general and specific description of its AI implementation. Then, we will provide our research about other applications and algorithms used to perform tasks that are similar to our application. We will then discuss our application trial runs and results. We will conclude by talking about our applications shortcomings and possible improvements.

General Application Information

Overview

The Nutrient Tracker application is a simple but effective command line interface application that makes use of user input, an open source Python database and what you could call "log files" that collect and assess data from a user to keep track of their eating habits. It also tracks how much the user has consumed of the four most commonly monitored nutrients based on FDA standards. Then, the application assesses a user's nutrient intake and makes suggestions on what the user can do to improve their diet. The food recommendations are based on the user's eating habits so that the user is given actual useful and enjoyable food instead of some random food selections they don't enjoy. The application also knows how to remove food from its database that the user no longer eats in case a user has a sudden change in diet.

This implementation follows a simple process. First, the user consumes a food item and then provides its barcode as user input to the application. Then, the application uses the

aforementioned Python database known as OpenFoodsFacts to look up the food item's info. The application will then record all food the user ate and track how much of the four key nutrients this provides. This is done using what might be called "permanent memory" in the form of CSV files.

The application also runs several tests on the inputted food. These tests are the basis for the AI implementation of the application because this is what we can change and edit to be considered more rational. There is a preliminary test run before any input is given that checks to see if a food item should no longer be considered frequently eaten. The second test is to see whether the food the user ate has a high content of one of the four nutrients. The third and final test checks to see if the users eats a food frequently and should therefore be added to the suggested foods.

After the user is done inputting food, the system can use the CSV files to assess the user's general nutrient intake for the day. It will then provide a short summary to the user on what nutrients they still need more of to fulfill the FDA daily value amount. The application is then designed to make suggestions on food that the user should eat to make up for those missing amounts. The application does have default, hard-coded foods it can suggest that are high in the four nutrients, but to improve user experience the application will first try to make suggestions based on the users eating habits. This info is pulled from a certain CSV file which is created using the previously mentioned tests that measure high nutrient content and consumption frequency. This will allow the user to have suggestions they can actually take advantage of instead of random suggestions for foods they may hate.

In summary, the application simply tracks the food the user eats to assess whether they are following FDA guidelines on nutrients. If they are not, then the application reads the past eating habits of the user to give personalized suggestions. Simply put, the application is trying to replicate advice a nutritionist might give you if they were simply focused on keeping your nutrient levels consistently satisfactory.

User Interface and Environment

Programming Environment

Our project was developed on Windows/Mac, using Python 3.10.

We made an effort to utilize the minimum third-party Python libraries feasible and to develop as much of the code from scratch as we could. This is to make sure we have as complete an understanding of the material as possible. The drawback is that some of our reasoning and algorithms might not be as "accurate" or as efficient as they would have been if we had used pre-existing libraries.

The only big exception to this rule is our database for our food products: OpenFoodFacts. However, this was purely for data purposes as collecting a database of every food item is obviously impractical.

User Interface

This program is generally very simplistic in its delivery to the user. We use simple command line inputs and preset Python modules. We also use a previously mentioned Python Database to look up food products called OpenFoodsFacts. We are also using simple CSV files over database software. However, this choice was slightly purposeful. The purpose was to showcase some of the applications capabilities without the complication of pulling up database software which is fairly hidden from the user without proper commands and access. Using simple CSV files allows for immediate feedback and provides the user a convenient look-up system. In most cases, we know it isn't exactly wise to provide the user access to data the program might manipulate but given the nature of this program, we found this to be acceptable. We do mention what we would change on a more general level in the "Potential Future Improvements" section on page 14 .

The startup user interface provides the user with the options to add a food, assess the day, or quit. Obviously the option to quit simply ends the program. The add a food option and the assess the day option are explained further in the "Functionalities" section below. Every function that can take user input has data validation in place to make sure the user doesn't throw any exceptions by accident.

Functionalities

The program utilizes five main functions for its execution:

1. menu()

   The menu function is the main organizer or the controller of the program. It provides the user with several options and based on the user's choice, activates or calls the appropriate function. It is also responsible for validating whether the user wants to continue further or exit the program.

   Initially, the program displays a welcome message letting the user know that the program is intended to track nutrients consumed by the user every single day. The program then provides some options to the user that the user can choose by entering any numbers from 1 to 3. Entering 1 makes the call to add_food() function and allows the user to add a new food that they consumed today. Similarly, entering 2 as a command will allow the user to assess nutrients that they have consumed so far today. Finally, entering 3 will exit the program and if the user types any number other than 1, 2 and 3, the program will throw an error message indicating to the user to restart the program or try a valid number as command. The program also has been set up in a way that the program will continuously

ask the user for a command after completing the task from each command until the user enters 3 and wants to exit the program.

2. add_food()
   The primary purpose of add_food() function is self-explanatory. The function will keep track of and store different foods or nutrients that the user has consumed throughout the day. As there are tons of nutrients in the real world and as it would be tremendously tedious and sort of ambiguous for our project to keep track of all nutrients, the program has been written in a fashion to keep track of the four most commonly tracked nutrients: Vitamin-d, Iron, Calcium and Potassium.

   The function begins by asking the user to enter the barcode for the food that they either consumed or are about to consume. The barcode is then used to get information about the food item from OpenFoodFacts. The library is the result of a non-profit project called Open Food Facts developed by thousands of volunteers from around the world. It is a free, online and crowdsourced database that provides information on various nutrients that are involved in the composition of different food products.

   When the call is made to OpenFoodFacts with user provided barcode, then it is expected to get food information in return but there can be some exception or errors and in those cases the item status 0 will be returned. As long as status 0 is returned, the user will be asked to input the barcode again. When status is equal to 1, it indicates that the food with appropriate barcode was found. However, as there's always a possibility that either the wrong barcode was given to the program or an unexpected food item was found, the program prints out the food information like product name and product brand, and asks the user if the printed food information is correct. If the user disapproves of the food information, then the user will be asked to reenter the barcode. However, if the user approves of the food information, then the program moves forward.
   A file "FDA_DV.csv" is provided or required for this program. The file includes the U.S. Food and Drug Administration's recommended amounts for vitamin-d, iron, calcium and potassium. Currently, the recommended amounts are as follows:
   - Vitamin-d: 20 mcg
   - Iron: 18 mg
   - Calcium: 1300 mg
   - Potassium: 4700 mg

   These nutrient names are read from the "FDA_DV.csv" file and appropriate values like amount of the nutrient and unit are extracted from OpenFoodFacts. In order to make sure that the units received for all nutrients are the same or equivalent, the program handles unit conversion. Depending on whether the unit is microgram (µg) or milligram, the

program takes appropriate steps for unit conversion. The "FDA_DV.csv" is closed after these steps are completed. Now, the program opens "today_food.csv" to keep track of food consumed on the day the program is executed. All the appropriate information extracted earlier from OpenFoodFacts that match with the "FDA_DV.csv" file are added in the "today_food.csv" which includes product name and amount of vitamin-d, iron, calcium and potassium in the product.

Once the program has added the food to "today_food.csv", the program will also ensure whether or not the food is frequently eaten. "past_food.csv" contains the information of previously consumed food and if recently added food already exists in the file then its count or frequency is raised. If the frequency is greater than 2 then it is regarded as frequently eaten food. "frequent_food.csv" contains the information for frequently eaten food. If the food already exists in the file then no further actions are required for "frequent_food.csv" but if the food doesn't exist in the file then the food has to go through a scan run by high_check() function. high_check() function ensures that the potential frequent food has enough desired nutrients. The food is only added to the "frequent_food.csv" if it meets the required amount of desired nutrients according to the rational variable. Finally, the user will be asked if they want to add another food or not, and depending on the answer, the program will either rerun the function or open the main menu.

3. assess_day()
   This function can be used to provide an overall analysis of different nutrients consumed by the user. It is able to compare the FDA's daily recommended amount for nutrients like vitamin-d, iron, calcium and potassium with the user's consumption and provide feedback. The main objective is to have the user consume the recommended amount for all nutrients, but if that is not the case then the program outputs nutrients that the user still needs to consume as well as the amount left to reach the recommended amount. Initially, the function opens the "FDA_DV.csv" file in read mode which contains nutrients as well as recommended amounts for those nutrients. The program goes through each nutrient and stores them in a hashmap with the nutrient name and recommended nutrient amount as key-value pairs.

   The program also utilizes another file called "today_food.csv" that contains nutrients and the amount of those nutrients consumed by the user so far on that day. Similar to the FDA hashmap, the program iterates through each nutrient and stores nutrient name and nutrient amount as key-value pairs in a hashmap. Once both "FDA_DV.csv" and "today_food.csv" have been iterated through, the files are closed.

As the program is designed to provide individual feedback for each nutrient, it becomes necessary to keep flags for each nutrient. That is why a flag with initial value of 0 is set for each nutrient i.e. vitamin-d, iron, calcium and potassium. After that, the current value or the amount of each nutrient consumed by the user so far is compared to the daily recommended amount by the FDA for each nutrient and if the amount of the nutrient consumed by the user so far is less than the daily recommended amount, then appropriate flags are raised or in other words, the values are changed from 0 to -1 for those nutrients. In the best case scenario, when the assess_day() function is called, the user has fulfilled all the requirements for daily recommended nutrient amounts by the FDA. In this case, no flags are raised and the user simply gets the feedback that all the nutrients requirements have been met for that day.
However, if any of the daily recommended nutrient requirements have not been met, then the program will alert the user with both nutrient names and amounts still needed to meet the requirement. The program has also been designed in a way that if the user is missing any of the nutrient requirements, then the program will recommend appropriate food items for consumption to the user that are rich in those nutrients. Initially, the recommendation is based on foods that are frequently consumed by the user or foods that have been stored in "frequent_foods.csv". The program iterates through the "frequent_foods.csv" file and tries to find a food that's rich in a nutrient that still needs to be consumed by the user. If the program is able to find the nutrient, then the program exits from "frequent_foods.csv" and recommends the food to the user but if the program is not able to find the nutrient then it iterates through the "default_foods.csv" file. The "default_foods.csv" file contains foods that the user might or might not have consumed in the past but are guaranteed to be rich in certain nutrients. As of now the file contains a food per nutrient as follows:

- salmon, vitamin-d
- spinach, iron
- low-fat yogurt, calcium
- avocado, potassium

As the program is written to keep track of only vitamin-d, iron, calcium and potassium, the program should be able to find the nutrient that the user still needs to consume and make appropriate recommendations on the food product. Once the nutrient is found, a food product is recommended to the user. In the worst case scenario if a nutrient is not found and the program is not able to display an appropriate recommended food product to the user, an error message is displayed. This should probably never happen but is checked, just in case.

4. check_past_foods()
   This function is the first function that gets executed. The main objective of this function is to make decisions on whether a previously considered frequent food still holds true. The file "past_food.csv" contains the information of all the previously added foods. The program iterates through each food in the file and determines whether or not to discard the food from being considered frequent. If the date the food was previously updated has passed a certain date then the food gets regarded as infrequent food and is removed from both "past_foods.csv" and "frequent_foods.csv".

5. high_check()
   This function checks whether the food item sent to it is considered high in any of the four tracked nutrients. The function sets a value based on a rational variable that it compares to the nutrient amount in the food. If the food has more of the nutrient than the rational variable value, it is considered high in content in that nutrient and that nutrient gets added to a list. Then, once the program has checked the food for all the four nutrients, it returns the list of what nutrients the food has a high content of.

AI Implementation

Designing an AI that makes suggestions based on past habits and based on nutrition advice is a bit complicated. The main struggle comes not from technological prowess but from the fact that different people will have different opinions on how specific measures should be defined.

The application is designed to recommend a specific food item to the user if it matches two criteria: if it is ‘high’ in one of the tracked nutrients and if it is eaten ‘frequently.’ This begs the question: what is considered a high level of a nutrient and how many times does one have to eat something for it to be considered frequent? These two questions are very much a matter of opinion and trying to set a finite way to measure these things could vary depending on the user. How much does the user eat? Does the user actually need more of a specific nutrient? If the user has a very simple diet with not much variety, do they consider eating something frequently to require much more consistency than someone with a very complex diet?

In actuality, trying to determine the values for these specific measures isn’t much different than simply trying to make the application pass the Turing Test. Say the user was sitting at a computer that they thought would send messages to a nutritionist or doctor but was actually sent to the AI application. If they were looking for suggestions on how to change their diet to receive enough necessary nutrients, they would have to send information about their past meals. It is fairly safe to say that a nutritionist would notice something commonly eaten in your diet that was a good source of nutrients and would comment on that specific food. They would most likely encourage the user to continue eating it. This is the goal of this application’s AI functionality. If

we could program the application to make a suggestion that the user actually thought came from a nutritionist or doctor that seems reasonable and rational, then the application would pass the Turing Test.

Note, the application isn't designed to be used a single day and then never used again. It should maintain the user's preferences in some sort of permanent storage. This application uses CSV files to store the information for future use so the AI will "remember" a user's personal food data. However, this raises another issue: when should a food item no longer be considered frequent?

When deciding if a food was frequently eaten, we also needed to be mindful if the food was high enough in a nutrient content for the application to care. There is no such restraint when deciding if a food is no longer considered frequently eaten. All we care about is whether this food has been eaten frequently *recently*. This again raises another question: what is considered recently?

This turns out to be another important question when considering passing the Turing Test. If the user were to contact a nutritionist and the nutritionist were to suggest they eat a food they haven't eaten in years, this would raise some suspicion. But in the mind of the application, without a way to measure infrequency, just as it measures frequency, this could result in some awkward suggestions from the application. Once again, monitoring infrequency is very much based on human opinion.

We, as the developers, could have simply just determined what we believed to be rational, but we decided to base our answers off of data from some sample test runs and known psychology. We will first discuss the general psychology we researched. You can read about our test runs in the "Rational Variable Tests" section below.

According to studies done by German psychologist Hermann Ebbinghaus, most people need to hear something of medium length 7 times before they remember it. He also said that if a person hears something and isn't trying to actively remember it, they will completely forget what they heard in 24 hours. These data points were the foundation for our decisions about the rational variable values.

Now, if we were to use these exact data points and simulate this in our application this would make the application a bit useless. Since a person needs to hear something about 7 times to remember it, it would make sense for us to say that a food eaten 7 times should be considered frequently eaten. However, during our tests on the application, 7 times felt a bit useless because it took a long time for the application to judge your preferences.

While this application does have components that give it the ability to try and pass the Turing Test, that isn't really the goal. The goal is to be a tool for quick computations and helpful advice, with the AI implementation aimed more towards being personal with the user. That being said,

as a general rule we decided to set the values for our rationality variables in a way that the application would be twice as intelligent as a human.

This implementation should be fairly easy to follow. It takes a normal human 7 repetitions to remember something. We want to make the AI application twice as smart as that. So we would cut that value in half and say the application could remember something if it heard it 3.5 times. In reality, hearing something 3.5 times doesn't make rational sense so we rounded that down based on data from the tests we ran. Therefore, in our application, a food eaten 3 times (before the frequency cut off point) is considered frequently eaten.

We used the same principle for when the application should consider a food item no longer frequently eaten. According to Ebbinghaus's studies, a person forgets something after about 24 hours. Since we wanted to make the application about twice as smart as a human, we set the time it took to forget something as double that time, or 2 days. So, everytime we boot up the program, it checks if any of the foods considered frequent were eaten in the last 48 hours. If they weren't, it forgets about them.

Determining the value of the rational variable that determines if a food is high in a specific nutrient was solely based on tests we ran using the application. These tests can be read about in the "Rational Variable Tests" section below.

Research on Similar Implementations

While we used our own, more simplistic system to recommend food and judge frequency, we did do research into the common ways other groups have implemented something similar. We looked into some very in depth recommendation algorithms that use deep learning to organize the immensely large amount of interests a person can have. We also read about something known as collaborative learning that uses other users' data to improve your experience and vice versa.

Basic Recommendation Algorithms

The goal of the recommendation system is to use artificial intelligence to process a large amount of user information. In recent years, deep learning technology has been an important part of data mining and artificial intelligence research. Deep learning can use a complex multivariate function as a training target and then plot its results on a grid environment. Then, the system finds available computing resources to run deep learning algorithms. After the computing resources are found, the data processing begins. After the data processing is completed, the computing resources will be released, and then an information server will feed back the running status of all computing resources and plot them on the grid. Then, the grid is used for selections, or in our case recommendations, at a later time. In more simplistic terms, these deep learning algorithms simply grab unused computer resources to run tests on user data. Then, that data is

graphed to show common trends based on user interests and those trends are used to make future guesses or recommendations.

The function of the module is to recommend the information that meets the requirements of the user according to the corresponding recommendation algorithm after acquiring the user's interest information. Of course, in the entire recommendation process, the users interest list contains a large number of interest information resources. This means we need to somehow sort out all this information effectively and without taking a ridiculously long time.

The majority of content-based suggestions are based on the user's choices or something known as the project rating, which is mined for recommendations through data analysis of projects with comparable content. Essentially, project ratings are a way of knowing how other people who seem to share similar interests as the current user may act. You might recognize this from a streaming services "You May Also Like" section.

When mining user behaviors and interests, the user may give explicit feedback through the user interface. For example, the user's direct likes and dislikes, and other visual information. This information is where the previously mentioned project rating comes from. Feedback is data that is accessed by a user indirectly, through clicks, browser activity, or certain behavioral traits of a purchase. Through the information that has been obtained, deep learning training is carried out to determine the user's preferences. The user is then matched with a list of recommendations that has a high degree of resemblance to their preferences.

Collaborative Filtering

One of the most extensively studied and utilized recommendation systems is collaborative filtering. The user's evaluation of the applications previous recommendations and associated historical information from both the current user and other users must be utilized by a collaborative filtering recommendation system.

A sub-algorithm called the "closest neighbor approach" is typically used in the collaborative filtering algorithm. This recommendation algorithm simply tries to find users with the closest current preferences to yours and will try to make use of their information to improve your experience. This means you could also be contributing to someone else's experience when you use any kind of app that uses collaborative filtering.

Obviously, when it comes to our project, we couldn't work with any kind of collaborative filtering algorithm because of the small scope of the project. We didn't put the project live on the internet nor have time to collect enough data. However, if we were to improve the project, using one of these algorithms seems like a very beneficial idea, not only for the evolution of the AI, but for the overall user experience.

Observations & Potential Improvements

Rational Variable Tests

When running tests on an AI application, usually, you are trying to find the supposed “best method” to accomplish your application's goal. In our case, the “best method” could simply be redefined as “what should the values of our three rational variables be?”

As previously mentioned, the three rational variables were:

1. How many times should a food be eaten to be considered frequently eaten?
2. How long should the user go without eating a food considered to be “frequently eaten” before it is removed from that category?
3. How much of a certain nutrient should be in a food item for it to be considered high in content of that nutrient?

We have already answered two of these questions in the “AI Implementation” section above. We used data from past psychology studies and the rule that the AI application should be about twice as smart as a normal human for it to be useful. Using this data and one simple rule, we determined that a food should be considered frequently eaten if it was eaten three times before the cutoff point. We also determined that the cutoff point for a food to be considered infrequent should be two days after it was last eaten.

It was also mentioned that we decided to round the half value 3.5 down to 3. This was decided because we originally rounded the value up to 4 and it felt a bit impractical. We mainly ran the tests to determine the third rational variable’s value based on nutrient content but during those tests we also decided we should round the first rational variable down to 3 because the application was taking too long to figure out our habits, and therefore wasn’t making informed decisions quick enough. Eating a food item once should mean nothing. Eating it twice could be considered an outlier. However, eating something three times seems like it should be the rational trigger that we enjoy this food. Eating it four times does further solidify this notion, but setting the limit to three felt more useful.

Regarding the actual tests, we ran the program and told the program that we ate foods we knew to be considered high in certain nutrients based on online nutritionist recommendations. This is actually where we got a list of foods to suggest to the user if the application didn’t have any data on them yet. We saved these foods to be used as convenient examples for both testing and for the user.

The rational variable that determines if a food item is high in a specific nutrient is a percentage value. We take the nutrient content in the food and compare it to the FDA daily value times the rational variable. If the nutrient content is greater than or equal to that value,we consider it high in content.

To run tests, we simply changed the percentage (the rational variable) until all the foods we knew to be high in content passed this condition. After all the tests, we set this value to be 0.15 or 15 percent. Since then, any other foods we would consider to be high in a nutrient have been considered high by the application.

Using the tests, scientific data, and simple rules we put in place, we believe we have set up the application to act as rationally as we can given our limitations. We discuss how we could make the application even more rational if we were to implement some improvements and make some changes that we list in the "Potential Future Improvements" section below.

Potential Future Improvements

This application has several things that could be improved, on both an AI and quality of life level. From having a better user interface to a more complicated AI implementation, here are the improvements we would make.

The most obvious and easiest improvement would be the use of a GUI. We could use something like Tkinter to create a very simple GUI that could use basic input boxes, display windows and buttons. This would provide a more pleasant user experience and could also be expanded upon very easily.

Another easy improvement is the addition of more nutrients to track. The nutrients we used were simply selected as they were the most commonly listed nutrients on food items. We also might want to add a simple feature where the user could select which nutrients they want to track. We could even add a feature where the user could point out if they have a deficiency in a certain nutrient so the application could adjust their suggested intake. Implementing these features wouldn't be that difficult if we used a database and user accounts to create unique experiences for all users.

Speaking of databases, a database could be used for storage instead of several CSV files. This would make things a bit easier memory wise and is a lot cleaner than our current implementation. Again, we have mentioned that using CSV files was purposeful to better showcase our applications functions, but if we were to be a bit more professional, we would use something like a MySQL database.

If we wanted to push this application to an actual app store, we could add an account system so users could access their information from any device via a database. This allows for more customization options and less volatile data. Plus, this account could store your specific food preferences and data and use it from any device. Using this app cross platform would make it much more convenient.

If we were to give this application accounts, and therefore connection to the internet, we could change our methods for assigning values to our rational variables. We could implement

something similar to a collaborative filtering algorithm that we mentioned in the “Research on Similar Implementations” section above. We could gather user data and see what food preferences users might share. Then, instead of making recommendations purely from the one user’s personal data, we could also make recommendations from all kinds of other users who had similar tastes.

Like other recommendation algorithms, we could ask the users questions based on their habits to confirm what they think is rational. For example, if the application sees a food that it thinks the user frequently eats, we could simply ask the user, “Do you eat this frequently?” This could help us get a gauge on what is considered frequent from other users. We could also ask users if they no longer eat a certain food instead of assuming they don’t if they haven’t eaten that item after a set amount of days. Depending on the responses to these questions, we could collect data to improve our rational variables so the application could make better guesses related to frequency and user preferences.

References

Liu, Feng, and Wei-wei Guo. “A Multi-Organ Nucleus Segmentation Challenge - IEEE Xplore.” *IEEExplore*, 2019, https://ieeexplore.ieee.org/document/8880654.

Goldentouch, Lev. “How Many Repetitions for Long Term Retention?” *Key To Study*, 27 Dec. 2014, https://www.keytostudy.com/many-repetitions-long-term-retention/.

Jones, Taylor. “7 Healthy Foods That Are High in Vitamin D.” *Healthline*, Healthline Media, 5 May 2022, https://www.healthline.com/nutrition/9-foods-high-in-vitamin-d.

Spritzler, Franziska. “12 Healthy Foods That Are High in Iron.” *Healthline*, Healthline Media, 27 Jan. 2020, https://www.healthline.com/nutrition/healthy-iron-rich-foods.

Jennings, Kerri-Ann. “Top 15 Calcium-Rich Foods (Many Are Nondairy).” *Healthline*, Healthline Media, 4 Nov. 2021, https://www.healthline.com/nutrition/15-calcium-rich-foods.

Lang, Ariane. “Potassium Rich Foods: 18 Foods High in Potassium.” *Healthline*, Healthline Media, 4 Feb. 2022, https://www.healthline.com/nutrition/foods-loaded-with-potassium.

Group Contributions

Affan Moyeed

Came up with the general idea for the AI implementation for the project. Gave some guidance on coding and equally contributed to the report. Created the Powerpoint presentation.

Matt Fackler

Came up with the general code structure, wrote about half the code and contributed equally to the report. Helped make the presentation.

Prashan Thapa

Wrote the other half of the code, equally contributed to writing the report, and helped make the presentation.