Cookin’ Not Cooked Final Report

Megan Zhang & Grace Liu

INTRODUCTION

As we will be off the university meal plan next year and will need to start cooking for ourselves, we wanted to find a way to plan meals that are healthy, budget-friendly, and easy to prepare. Our goals were to discover new recipes that support healthier eating habits while also keeping costs low. To address these needs, we created a diet optimizer using constraint programming. This tool helps generate meal plans that meet specific health goals and stay within a defined budget, making it easier for us to maintain a balanced diet without overspending.

IMPLEMENTATION

User Inputs

The constraint programming solver takes several user inputs that parameterize the optimization process. First, users provide their inventory of available ingredients through a text file, where each line contains an ingredient name and its available quantity. This data is processed and mapped to the system's ingredient database to create an accurate representation of the user's pantry. Users also specify their dietary preferences by listing ingredients they dislike and are allergic to in a separate file.

Additionally, users define non-ingredient constraints through a configuration file, including their budget limit, maximum calorie cap per meal, and the desired number of meals to plan.

Constraint Programming Solver

Initially, we considered using linear programming to build the diet optimizer. However, we realized that constraint programming offered greater flexibility, which made it better suited for representing the diverse and sometimes conflicting requirements of a realistic diet.

The implementation uses Google's OR-Tools constraint programming solver to optimize meal selection based on various constraints including budget limitations, ingredient availability, nutritional requirements, and personal preferences.

Setup and Preprocessing

The code begins by importing necessary libraries and defining preprocessing functions to read inventory data from a file and create a dictionary mapping ingredient IDs to their available

amounts and a dictionary mapping disliked ingredients to their respective IDs ensuring that the data is properly formatted for the constraint programming model.

Model Definition

The constraint programming model is defined in the `cp` function, which takes several parameters including `data` which is a JSON data file containing recipe and ingredient information, `budget` which represents the user-defined maximum spending limit, `calorie_cap` which represents the user-defined maximum calorie limit per recipe, `inventory` which encodes the user's available ingredients and their amounts, `disliked_ct` which is a count of disliked ingredients per recipe, `allergies` which is a set of user-inputted allergens, and `chosen_meals` which is the number of meals to select, defaulted to 5.

Variable Definition

The model defines two primary sets of decision variables, recipe selection variables (`x`) for each recipe and ingredient variables (`b`, `k`), one for each ingredient. The recipe selection variables are binary, indicating whether each recipe is selected (1) or not (0). The variable b_i represents the number of units of ingredient i that must be bought in order to make the recipes. The variables k_i are intermediate variables that are meant to constrain variables b so that they can only assume values of multiples of 100. The reasoning behind this came from the units of ingredients in the recipes, which were not all integers, posing an issue given that constraint programming requires values to be integers. To solve this issue, we multiplied all units by 100 to get two decimal places, where the value 100 equated with one unit. We also wanted to restrict the program to buy ingredients in whole units, since it is impossible to buy a fractional unit in real life, so we used k to help constrain b to only be in multiples of 100.

Constraints

The model implements the following constraints: meal count constraint, ingredient availability constraint, budget constraint, unit purchase constraint, allergy constraint and calorie cap constraint.

The meal count constraint ensures that the number of recipes chosen is exactly `chosen_meals` by defining the sum of all recipe selection variables to be `chosen_meals`.

The ingredient availability constraint maintains that for each ingredient, the total amount used across all selected recipes must not exceed the sum of the inventory amount plus any additional purchased amount. To do this, the program calculates the "needed" units of the recipe and subtracts the stored inventory units to get the "bought" units. It then sums up all the "bought" units and constrains it so that it is less than or equal to the budget.

The budget constraint ensures that the total cost of all purchased ingredients does not exceed the specified budget. Additionally, because in the real world ingredients can only be bought in specified unit sizes, the ingredient purchase variables are defined such that they can only take on values in increments of 100, where each increment represents purchasing 1 additional unit of the ingredient since the amount of each ingredient needed per recipe is defined as a proportion of 1 unit as sold by Kroger. The budget is thus scaled by 100*100 to accommodate the scaling of ingredient amounts and prices, and this constraint ensures that buying whole units of ingredients such that the amount bought is the ceiling of the amount needed across all recipes remains under the user's proposed budget.

The allergy constraint ensures that any recipes which use ingredients that the user is allergic to are not included in the output by automatically setting that recipe's selection variable to 0.

Finally, the calorie cap constraint ensures that each selected recipe does not exceed the specified calorie cap.

Objective Function

The objective function is a multi-criteria optimization that balances nutritional value and preference satisfaction by including variables to track the total protein and cholesterol across all selected recipes as well as an array which keeps track of how many disliked ingredients are in each of the selected recipes.

The final objective function to is:
obj_expr == (4*total_protein - total_chol - 50 * dislike_sum))

The goal is to maximize a weighted expression that favors high protein content, penalizes high cholesterol content, penalizes recipes with disliked ingredients which reflects the desire to find nutritionally balanced meals that respect user preferences, prioritizing protein while minimizing calories, cholesterol, and disliked ingredients.

Originally, we wanted to also minimize calories by adding in a negatively weighted total_calories variable in the objective function. However, we realized that this unintentionally led to getting stuck on the same total calorie count no matter what the inputted calorie cap was. For example, 500 calories/meal had the same total calorie count as 800 calories/meal. We decided that there might be situations in which people want recipes with more calories, especially if they are putting higher number calorie caps. Thus, we took out the calorie minimization in the objective function and relied on the calorie cap to constrain the number of calories consumed.

Solution Extraction and Reporting

After solving the model, the code extracts and reports the solution, including selected recipes, additional ingredients to be purchased and their associated cost, nutritional information and nutritional ratios (calories-to-protein, calories-to-cholesterol). The solution is only reported if the solver finds a feasible or optimal solution.

Data Pre-Processing

In order to run the constraint solver on large datasets that we don't have to create manually, we used a few different APIs: Spoonacular to get the recipes, Kroger to get ingredient prices and Gemini to do unit conversions. All of the information obtained from the above three APIs is then consolidated and passed into the constraint solver.

First, after asking the user for their available ingredients, Spoonacular is queried to return the desired number of recipes (defaulted to 100). This allows for the recipes to be somewhat relevant to the user's available ingredients so that the budgeting process doesn't always end up with needing to buy every ingredient in the recipe.

Then, the ingredients needed across all 100 recipes are cleaned by removing preceding adjectives and deduplication is attempted by checking for similarities across the ingredients using `difflib`. For example, all types of all specific types of tomatoes should be grouped together as a generic "tomato" ingredient since likely it does not matter which variety of tomato the user buys and buying multiple different types when they are all interchangeable would unnecessarily strain the budget. Then, the final condensed ingredient list is passed to Kroger's API to get the unit price of each ingredient and the unit size that Kroger sells it in.

Finally, because the units that each recipe requires the ingredients to be in vary from each other and the units that Kroger sells them in, we call Gemini to standardize the recipe units to Kroger's units and then ask it to determine what proportion of 1 Kroger unit each recipe calls for. This is so that in the unit purchasing constraint, the units to buy can be clearly encoded. For example, if a recipe needs 15 eggs, and Kroger sells them in dozens, Gemini would return that the recipe requires 1.25 Kroger units, which would be 125 when scaled for the CP solver. Then, the unit purchasing constraint would figure out that the amount to buy is 200, or 2 units.

EXPERIMENTATION

Our main experimentation came with finding the weights for the objective function for total_protein, total_chol, and the soft constraint.

We were informed during our check-in that calories, protein, and cholesterol have different units, which would affect how much each variable carried weight in the objective function. One

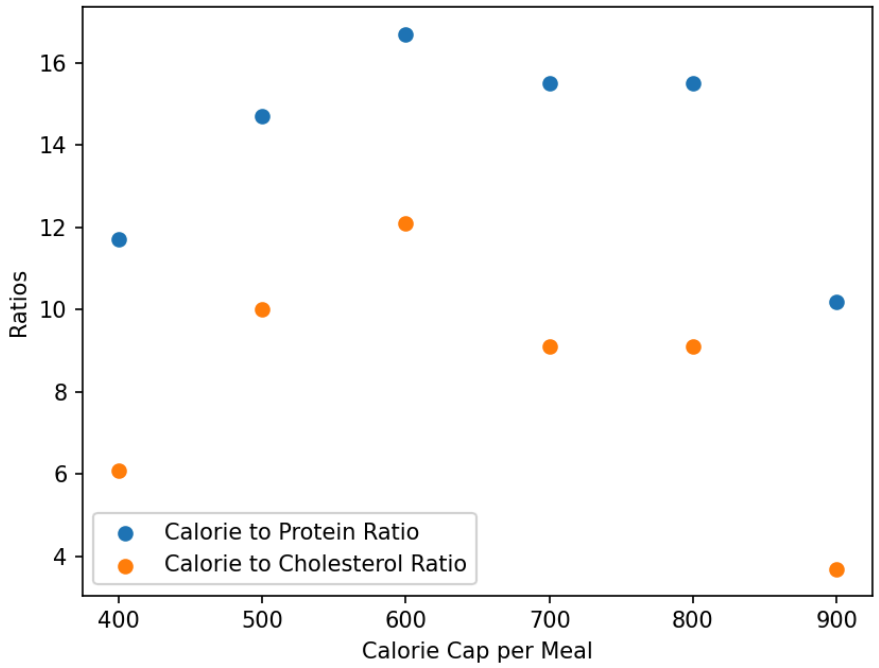
observation we had in regards to this issue was that there was not an easy way to convert between the units of calories, protein, and cholesterol. Thus, we researched an optimal ratio of calories to protein to cholesterol, and found that the ideal calories to protein ratio and the ideal calories to cholesterol ratio were 10. We made this the foundation of our experimentation.

To experiment on which weights would give us the closest to our ideal ratio, we did a sweep of all the possible weights. First, we knew that we wanted protein to carry more weight in the objective function, since that was the main concern of the maximization. From this, we assumed that the weight for total_protein would always be greater than the weight for total_chol, and made it so total_chol would always be weighted 1, and we would try to find the optimal weight for total_protein. We iterated through every possible protein weight from 1 to 100 and tested the objective function on the solver, and kept track of which weight produced ratios with the closest distance to 10. One issue was that the calorie-to-cholesterol ratio would end up dipping too low to 3 in favor of a calorie-to-protein ratio of 10, so we set a hard limit to ensure that the calorie-to-cholesterol ratio was at least 6. We did not have this concern with the protein ratio, since a higher-than-ten calories to protein ratio could be viewed in a positive light.

We discovered early on that the ratios would change when you kept the weights static but changed the calories per meal cap. Thus, we ran the sweep on 6 calorie caps, from 400 calories/meal to 900 calories/meal, and recorded the most optimal protein weight.

Calorie Cap	**Protein Weight**
400	3
500	4
600	4
700	4
800	4
900	4

Since the only differing protein weight was with a calorie cap of 400, we decided to choose a static protein weight of 4. We tested this weight and found the following results for the calorie-to-protein and calorie-to-cholesterol ratios:



(Note: we manually plotted this graph since the calorie cap is added to the model as a constraint, which we could not figure out how to remove/reset without restarting the entire model. Thus, we manually changed the calorie cap and ran the model before setting up the array to plot.)

As shown above, most ratios were close to 10, with a few high calorie-to-protein ratios (but as mentioned above, this could be viewed in a good light). The outlier is the 900 calories/meal cap. After experimenting with the protein weight, we realized we couldn't get a better answer than those ratios except for when we started changing the cholesterol weight too.

Thus, we decided to also sweep cholesterol weights from 1 to 5 and found that 900 cal/meal's ideal weights were 7 for protein and 2 for cholesterol. Testing on 1000 cal/meal and 1100 cal/meal also resulted in 7 for protein and 2 for cholesterol.

We wanted to test out making the weights dynamic according to the inputted calorie cap through a linear equation for the weights. Inputting the points into a linear regression, we got the following equations: $W_{protein} = 0.006190*C + 0.3571$ where C is calorie cap, and $W_{chol} = 0.001786*C + 0.03571$.

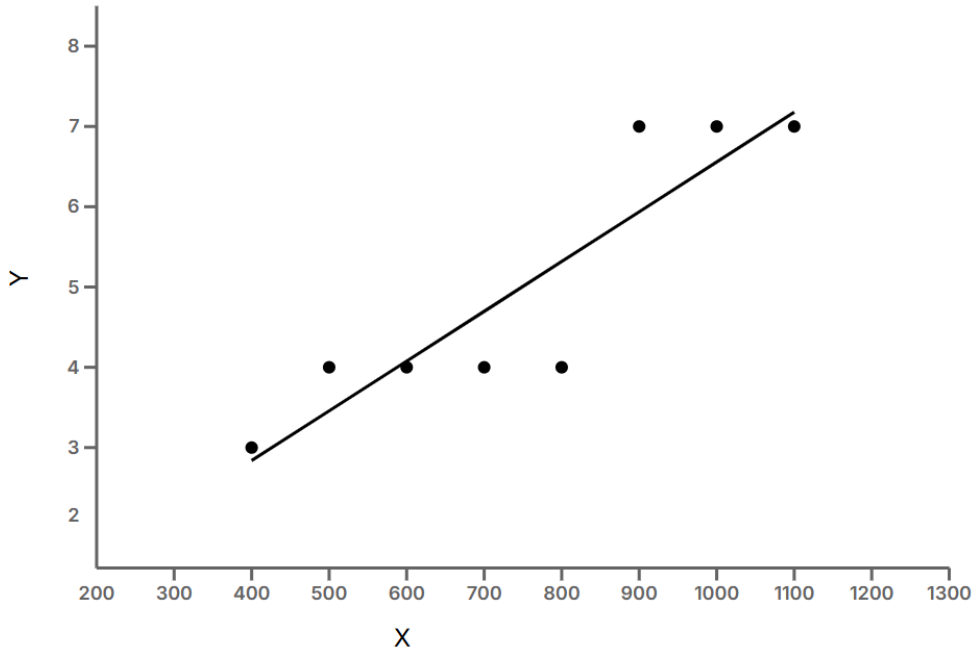


Figure. Linear regression for calorie cap vs protein weight

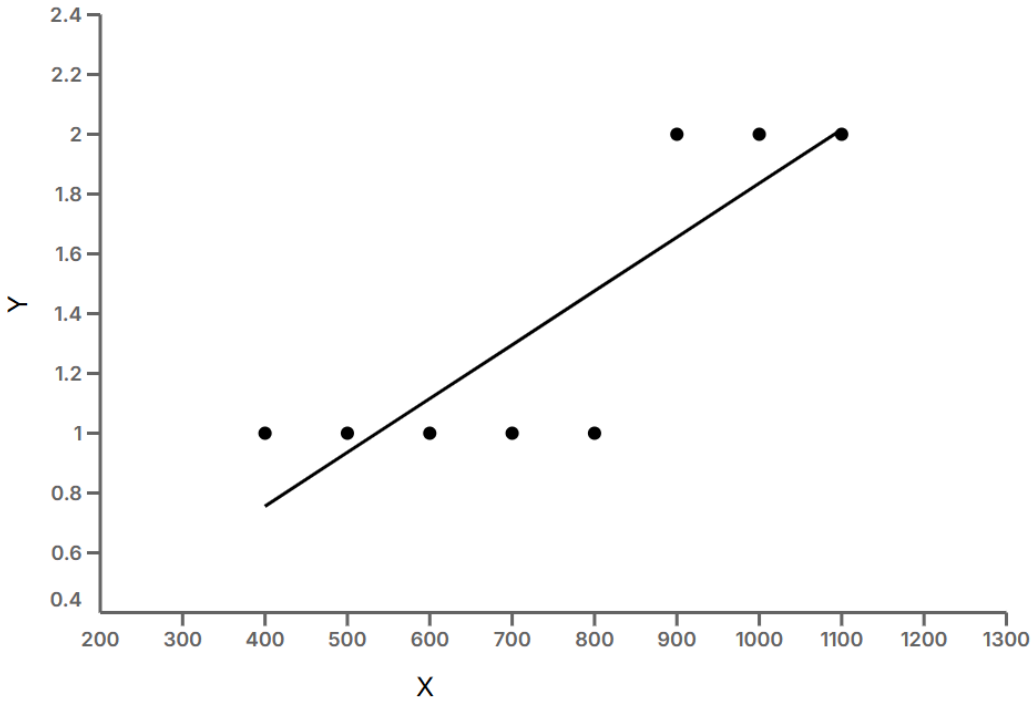


Figure. Linear regression for calorie cap vs cholesterol weight

Given the very stepwise nature of our points and the fact that cholesterol only had two weight values, we were skeptical about this approach.

Surprisingly, most of the calorie cap values did quite similar to our static 4, 1 weights, but there were a few hiccups which produced undesirable ratios, such as 500 and 850 producing ratios of 20 and 30. We decided to stick with our original weights of 4 and 1 as a result.

Another variable that we experimented on was the disliked ingredients soft constraint's coefficient. We wanted to make sure that if an ingredient was inputted as being disliked, it would not appear in any outputted recipes unless its contribution to the objective function was justifiably large. Therefore, we started with a high penalty to disliked ingredients at -500 per disliked ingredient in each recipe. However, we noticed that it would never output any recipes with any number of disliked ingredients which meant that this effectively became the same as the hard allergies constraint. This was likely because the magnitude overpowered all other coefficients in the objective function leading to the objective value always being extremely negative. Therefore, we had to lower the threshold, and at a coefficient of -200, we saw that sometimes a recipe containing a disliked ingredient would appear, but only if its nutritional value was disproportionately high, for example a beef stew contributing 100 grams of protein, which was about 4 times as much as any other recipe. We could have stopped at this threshold, but we noticed that when beef was then added as an allergy, the resulting list of recipes had a calories to protein ratio of closer to 40 which was much too high to be a reasonable diet, according to FDA metrics which says that a good ratio is around 10. Finally, we settled at a value of -50 which left the worst calories to protein ratio at around 19 which was acceptable to us since it was less than double the ideal. Pushing the coefficient further down would certainly lead to better metrics, but it would also severely diminish the point of having this soft constraint.

TESTING

To test our CP solver and verify that its performance was accurate, we hard-coded several test cases based on relatively small recipe and ingredient pools. This controlled testing approach allowed us to validate the solver's behavior against predetermined optimal solutions while ensuring that all constraints were properly enforced. We designed test cases with varying levels of complexity, specific constraint validation tests targeting each individual constraint, and edge case tests such as zero inventory scenarios and minimal budget situations. For each test case, we manually calculated the expected optimal solution by enumerating all possible recipe combinations that satisfied the hard constraints, computing the objective function value for each valid combination, and identifying the recipe set that maximized the objective function.

Our verification process focused on validating four key aspects of the solver's performance. First, we checked unit purchase integrity to ensure the solver correctly handled the discrete nature of ingredient purchases, such as buying 3 full units when 2 recipes required 1.5 and 1.2 units of an ingredient. Second, we verified hard constraint compliance, confirming that all selected meal plans strictly adhered to budget limitations, calorie caps, and allergy restrictions. Third, we

assessed soft constraint performance by tracking how well the solver balanced the preference for minimizing disliked ingredients against nutritional goals, confirming that the -50 weight in the objective function created an appropriate balance. Finally, we validated nutritional optimization by ensuring the solver correctly prioritized maximizing protein content and minimizing cholesterol levels across selected recipes.In all test cases, the solver successfully satisfied all hard constraints including budget, allergies, calorie cap, and unit purchasing, which validated our constraint formulations and scaling approaches. For each test case with a known optimal solution, the solver consistently identified the correct recipe combination that maximized the objective function, confirming its ability to effectively navigate the search space.

Performance testing showed that for small test cases with 10 or less recipes, the solver typically found optimal solutions in under 0.025 seconds, while the largest test cases with 100 recipes found solutions within 0.08 seconds. We hypothesize this is mostly just due to the solver trying to parse through all of the input data perhaps in conjunction with increased search space size since the increase in magnitude of the solution time was not correlated to an increase in the size of the input problem, so likely overhead costs increased the solution time. One interesting thing to note is that if the result was that there were no feasible solutions, then the CP solver could return a result within 0.001 seconds, indicating that the solver is very quick to determine the infeasibility of a problem. This is likely because it is very easy to check if there are for example not enough total recipes to pull from either if all recipes are too high in calories or too many recipes contain allergens. Another observation was that adding more restrictions did not significantly increase search time because the additional constraints actually reduced the search space by eliminating many potential solutions early in the solving process. When hard constraints like allergy restrictions or tight budget limits were added, the solver could likely quickly prune branches of the search tree that violated these constraints, focusing computational resources on exploring only viable solutions. This pruning effect could counterbalance the increased complexity from having more constraints to check, leading to similar or sometimes even improved performance compared to less constrained versions of the problem.