Optimization

Salaar Liaqat

Data Sciences Institute, UofT

Outline

- Setting up an Optimization Problem
- Dynamic Programming

Section 1

Setting up an Optimization Problem

Types of Optimization Problems

- *Optimization* refers to maximizing or minimizing a function with respect to its inputs
- Continuous optimization is when all the variables in the problem are continuous
- Discrete optimization occurs when some or all of the variables in the problem are discrete
    - Continuous: how many hours should workers in a factory work to maximize profits?
    - Discrete: how do I allocate TAs to teach within a department?

Autocorrect Example

- Autocorrect in an optimization algorithm. It has two parts
  - We need a list of known words and their use frequency
  - Classify errors are either: add a letter, remove a letter, substitute a letter, or switched two adjacent letters
- We quantify the error distance as the of errors in a string.
  - “ovon” -> “oven” is error distance 1
  - “ovvvn” -> “ovven” -> “oven” is error distance 2

Autocorrect Example

1. Check whether a word is in the dictionary
2. If the word is not in the dictionary, generate words that are error distance 1 or 2 from the given word
3. Rank the most likely correction given the error distance and use frequency

- "thene" could be "then" or "the," but but "the" is more common

Autocorrect Example

What are the steps to model the problem?

- We have the specification of possible inputs
  - Text, discrete
- The *objective function* is the function you are trying to maximize more minimize
  - Function with 2 variables: error distance and frequency
- Are we maximizing or minimizing the objective function
  - Minimize error distance and maximize frequency
- Identify the *constraints* in the problem
  - Only looking for words in the dictionary, only looking for words with error distance 1 or 2

Shortest Path in a Graph Example

- Finding the shortest path between two nodes on a graph is a discrete optimization problem
- The range of inputs are all possible paths from A to B
- The objective function is the length of the path
- We are minimizing the objective function
- And there are no constraints

Brute Force

Consider the following problems and proposed solutions

- You want to consume all necessary nutrients and calories at the lowest cost. So, you find all valid combinations of foods and find their cost.
  - If there are 10 foods, and 15 nutritional categories, then there are $2^{10\times 15} = 1.42 \times 10^{45}$ combinations to evaluate
  - We will fix this with *linear programming*
- You are robbing a store but the escape vent can only carry 4 kg of goods. To steal the maximum money's worth of goods, you calculate every set of goods and find the one giving the most value
  - If there are 3 goods in the store, then there are 8 combinations. But with 4 goods, there are 16 combinations. This solution is $O(2^n)$ time.
  - We will fix this with *dynamic programming*

Section 2

Linear Programming

Linear Programming

- Linear programming (LP) takes advantage of a program being linear. (what does that mean?)
- If we're considering a food that already fills one nutrition category, we can eliminate all other combinations that use the food
  - Sounds obvious, but brute forcing doesn't consider this!
- By this process of elimination, we make the problem much faster to solve.

Implementing LP in Python

Let's consider a very simple diet problem where the goal is to minimize the cost. There are 3 foods: apples ($3), bananas ($1), and oranges ($3). We want to meet 3 constraints: of vitamin A, a number of vitamin B, and a number of calories.

- Assume there is no upper limit on calories or vitamins
- The PuLP library is a popular linear programming library to do this in Python

```python
from pulp import LpProblem, LpMinimize, LpVariable, lpSum
```

Implementing LP in Python

```
diet_problem = LpProblem("Diet_Problem", LpMinimize)

# Define output variables
x1 = LpVariable("Apples", lowBound=0)
x2 = LpVariable("Bananas", lowBound=0)
x3 = LpVariable("Oranges", lowBound=0)

# Define objective function (minimize cost)
diet_problem += 3 * x1 + x2 + 3 * x3, "Total_Cost"

# Define nutritional constraints
diet_problem += 50 * x1 + 120 * x2 + 60 * x3 >= 2000, "Calorie
diet_problem += 2 * x1 + 3 * x2 + 5 * x3>= 40, "Vitamin A"
diet_problem += 12 * x1 + x2 + 2 * x3>= 50, "Vitamin B"

diet_problem.solve()
```

Implementing LP in Python

```
print("Optimal Diet:")
print(f"Apples: {round(x1.value(), 2)} units")
print(f"Bananas: {round(x2.value(), 2)} units")
print(f"Oranges: {round(x3.value(), 2)} units")
print(f"Total Cost: {round(diet_problem.objective.value(), 2)}
```

```
Optimal Diet:
Apples: 2.88 units
Bananas: 15.47 units
Oranges: 0.0 units
Total Cost: 24.1
```

Section 3

Dynamic Programming

Problem

The escape vent can carry only 4 kg of goods. The items are:

- Stereo: \$3000, 4 kg
- Laptop: \$2000, 3 kg
- Guitar: \$1500, 1 kg

We've established the brute force is not a valid general solution (although feasible in this case)

- The idea behind dynamic programming is that we'll solve subproblems that will lead to a solution to the big problem. We can pack items starting by considering smaller, sub backpacks

Guitar Row

- Each dynamic programming problem starts with a grid
- Each cell contains a list of items that can fit at that point
- For cell Guitar 1, a guitar will fit there. It will also fit in cell Guitar 2, 3, 4
- Sounds redundant, but let's keep going

	1	2	3	4
Guitar				
Stereo				
Laptop				

Stereo Row

- In the second row, we can steal the stereo or the guitar.
- At 1 kg, you can only steal the guitar, same as for every other cell until Stereo 4, at which point you can steal the stereo and only the stereo.

	1	2	3	4
Guitar				
Stereo				
Laptop				

Laptop Row

- Now we can steal all 3 items
- In the first two columns, we still can only steal the guitar. But in Laptop 3, we can steal the laptop
- Laptop 4 is the interesting step. We could steal only the stereo, or the laptop and something else for 1 kg. What is that 1 kg item?
- According to the above row, the max value for 1 kg is the guitar!

	1	2	3	4
Guitar				
Stereo				
Laptop				

Solution

- If we stole the guitar and laptop, the total value is 3500, which is greater than just stealing the stereo
- Thus, we should steal guitar and laptop

	1	2	3	4
Guitar				
Stereo				
Laptop				

Formula for each cell

- We skipped some very trivial steps in calculating cells aside from the last one
- Here's the explicit formula to calculate each cell's value

Let i be the row and j be the column.

$$\text{cell}[i][j] = \max \begin{cases} \text{the previous max at cell}[i-1][j] \\ \text{value of current item} + \text{value of remaining space} \end{cases}$$

The value of remaining space is cell[i-1][j-item's weight]

Python Implementation

```
def initialize_table(rows, cols):
  return [[0] * cols for _ in range(rows)]
```

Python Implementation

```python
def knapsack_dynamic_programming(values, weights, capacity):
  n = len(values)
  dp = initialize_table(n + 1, capacity + 1)

  # Fill the table using dynamic programming
  for i in range(1, n + 1):
    for w in range(capacity + 1):
      # Include the current item if it fits in the knapsack
      if weights[i - 1] <= w:
        dp[i][w] = max(dp[i - 1][w], \
        values[i - 1] + dp[i - 1][w - weights[i - 1]])
      else:
        dp[i][w] = dp[i - 1][w]

  selected_items = traceback(dp, values, weights, capacity)

  return dp[n][capacity], selected_items
```

Python Implementation

```
def traceback(dp, values, weights, capacity):
  selected_items = []
  i, w = len(dp) - 1, capacity

  while i > 0 and w > 0:
    if dp[i][w] != dp[i - 1][w]:
      selected_items.append(i - 1)
      w -= weights[i - 1]
      i -= 1

  selected_items.reverse()
  return selected_items
```

Python Implementation

```python
values = [3000, 2000, 1500]
weights = [4, 3, 1]
capacity = 4

max_value, selected_items = \
knapsack_dynamic_programming(values, weights, capacity)

print("Maximum value:", max_value)
print("Selected items:", selected_items)
```

```
Maximum value: 3500
Selected items: [1, 2]
```

Section 4

Recommended Problems and References

Recommended Problems and Readings

- Cormen (highly optional):
- Chapter 14, more advanced dynamic programming
- Chapter 29, more advanced linear programming
- Bhargava: Chapter 9 exercises
  - 9.1, 9.2
  - Read the knapsack problem FAQs on page 171
  - Follow the example about longest common substring on page 178

Recommended Problems

- Write the code to brute force the diet problem. Compare the run times using the `timeit` library.
- Modify the code from the slide such that there is an upper bound for calories and vitamins.
- Page 17 of Bhargava covered the travelling sales person problem. Is it possible to improve the proposed solution using any method we learned today?

References

- Bhargava, A. Y. (2016). *Grokking algorithms: An illustrated guide for programmers and other curious people.* Manning. Chapter 1.
- Cormen, T. H. (Ed.). (2009). *Introduction to algorithms* (3rd ed). MIT Press. Chapter 1 and 3.