Bishop's University
Department of Computer Science
CS316 - Artificial Intelligence - Fall 2024
Course Project
From November 8th to December 15th

Project: Solving the Knapsack Problem using Hill Climbing and K-NN

Introduction

The Knapsack Problem (KSP) is a classic optimization problem where the goal is to maximize the total value of items that can be carried in a knapsack without exceeding its weight capacity. This project combines Hill Climbing, a local search algorithm, with K-Nearest Neighbors (K-NN) regression to predict density values of items based on their weight, value, and volume. The density value serves as a criterion for deciding whether an item should be included in the knapsack.

Objective

The main objective of this project is to implement a solution to the KSP using Hill Climbing and K-NN to predict the density of items, allowing for informed decisions about item selection based on total weight and density values. By incorporating uncertainty into density calculations, the model aims to reflect real-world conditions more accurately.
The goal is to select a set of items that maximizes total value, while keeping the total weight within the knapsack's capacity and ensuring the average density remains below a threshold of 0.5.

Data Preparation

The dataset consists of two parts, both provided in two separate csv files:

1. **Training Dataset**: Used for training the K-NN model to predict density values based on the following attributes:
    - **Weight**: The weight of the item.
    - **Value**: The monetary value of the item.
    - **Volume**: The physical space occupied by the item.
    - **Density Value**: An additional computed attribute derived from the weight, value, and volume, incorporating a certain level of uncertainty to make predictions more realistic.

    The formula for density value is:

$$\text{Density} = \frac{\text{Value}}{\text{Weight} \times \text{Volume}} \times (1 + \text{Uncertainty})$$

where **uncertainty** is drawn from a normal distribution.

Interpretation

1) **Value per Volume and Weight**: This density measure represents the **value efficiency** of each item relative to both its weight and volume. Higher density values indicate items that deliver more value per unit of weight and volume, making them attractive for inclusion if they provide good value without consuming too much capacity.
2) **Uncertainty Term**: The **uncertainty** component is an adjustment factor to account for any variability or imprecision in the measurements of value, weight, or volume. This could represent unpredictable factors in real-world scenarios, like slight variations in item dimensions or weight.

How It Is Used in KSP

- **Selection Criterion**: This density measure can be used to prioritize items. Items with a higher density (more value per unit of weight and volume) are generally more efficient choices, helping to maximize the knapsack's total value within the given constraints.
- **Uncertainty Impact**: Adding the uncertainty term enables a **buffer for variability** in the properties of items. For instance, items with lower density but higher uncertainty might be deprioritized to avoid risk, helping to keep the knapsack within constraints even if item weights or volumes fluctuate.

Application with Constraints

In a Hill Climbing approach to the KSP:

- **Thresholding**: The density measure with added uncertainty could be used as a threshold criterion. For example, only items with density above a certain level (considering uncertainty) may be selected to ensure high efficiency.
- **Comparing Items**: This density allows for more nuanced comparison among items, balancing value, weight, volume, and the uncertainty factor. For instance, items that barely meet the density threshold may be excluded if they have high uncertainty, as this could put the knapsack solution at risk of exceeding capacity.

2. **Test Dataset**: A separate dataset that contains the same attributes (weight, value, and volume) but does not include the density value. This dataset will be used by the Hill Climbing algorithm to make selections based on predicted densities.

Implementation Steps

1. Load the training and testing datasets. Use pandas to load data into `DataFrame`.
2. **K-NN Regression**: Utilize the K-NN regressor algorithm to train a model on the training dataset.
3. **Hill Climbing Algorithm**: Implement the Hill Climbing algorithm for the KSP with the following tasks structured into separate functions:

3.1. **Generate Neighbors**: Create a function that generates neighboring solutions by either adding or removing items from the current solution.

- **Function Name**: `generate_neighbors(current_solution, items)`
- **Parameters**:
  - `current_solution`: The current selection of items in the knapsack.
  - `items`: A list of all available items.
- **Returns**: A list of neighboring solutions.

The `generate_neighbors()` function performs the following tasks:

1) Initializes an empty list to store neighbors.
2) Generate three neighbor solutions by modifying the current solution:
   - Randomly selects an item to add or remove from the solution.
3) Returns the list of generated neighbors.

3.2. **Evaluate Solutions**: Develop a function that evaluates each solution based on total value, total weight, and predicted density.

- **Function Name**: `evaluate_solution(solution, items, knn_model)`
- **Parameters**:
  - `solution`: The current selection of items.
  - `items`: A list of all available items.
  - `knn_model:` The knn model used to predict the density value.
- **Returns**: A tuple containing the total value, total weight, and predicted density of the solution.

The `evaluate_solution` function performs the following tasks:

1) **Initialize Totals**:
   - Sets `total_value` and `total_weight` to zero to keep track of the cumulative value and weight of the selected items in the solution.
2) **Sum Values and Weights of Selected Items**:

- Iterates through the indices in the provided solution to retrieve each item's corresponding value and weight from the items `DataFrame`, accumulating these values into `total_value` and `total_weight`.

3) **Predict Density Values**:
   - Initializes an empty list `predicted_density_values` to store predicted density values, checks if the solution is not empty, and then prepares input data (`weight, value, and volume`) for each item in the solution to predict densities using the K-NN model, appending the results to the list.
4) **Calculate Average Density**:
   - Computes the average density of the selected items from the `predicted_density_values`. If the list is empty, it assigns zero to `average_density` to avoid division by zero.
5) **Return Results**:
   - Returns a tuple containing the `total_value, total_weight`, and `average_density` of the selected items in the solution.

3.3. **Hill climbing main function**: Develop a function that solve the KSP using Hill climbing algorithm based on the items, capacity and knn model.

- **Function Name**: `Hill_climbing_knapsack(items, capacity, knn_model)`
- **Parameters**:
  - `items`: A list of all available items.
  - `Capacity`: Capacity maximum of the knapsack.
  - `knn_model:` The knn model used to predict the density value.

The climbing function performs the following tasks:

1) **Generate an Initial Random Solution**:
   - Create an empty list to represent the current solution.
   - Randomly decide how many items to include in the current solution.
   - Select random items from the available items list to populate the current solution, ensuring no duplicates.
2) **Evaluate the Current Solution**:
- Use the `evaluate_solution` function to calculate the total value, weight, and density of the current solution.
3) Loop Until No Better Neighbor is Found:
   - Inside the loop, generate neighboring solutions based on the current solution.
   - Initialize variables for tracking the best neighbor and best value found.
4) Evaluate Each Neighbor:
   - For each generated neighbor, evaluate its value, weight, and density using the `evaluate_solution` function.

- Check if the neighbor meets the constraints (weight within capacity and density above a threshold).
- Update the best neighbor and best value if the current neighbor is better.

5) Update Current Solution:
   - If a better neighbor is found, update the current solution to this neighbor; otherwise, exit the loop.
6) Return the Best Solution:

- After exiting the loop, return the best solution found along with its evaluation.

By structuring your code in this way, you will ensure clarity and modularity, making it easier to test and debug each part of the algorithm.