Entropy Analysis and Commit Owner Reconstruction

Algorithms and Complexity Assignment

by Kristi Shehaj, Frenki Selmani

Overview

This project consists of two distinct but computationally analytical parts:

1. **Entropy Analysis:** Measure and analyze the entropy of natural language text using n-gram models.
2. **Commit Owner Identification:** Reconstruct a list of contributors (employees) based on a continuous string of concatenated employee IDs (the “weld string”).

Both parts are implemented in Python and use fundamental principles from information theory and dynamic programming.

Part 1 - Entropy Analysis

1.1 Objective

The objective to compute the conditional entropy of cleaned natural language text using n-gram models (n=0 through n=10), identifying the most and least frequent n-grams and understanding the predictability of the language.

1.2 Methodology

- **Preprocessing**: The text is cleaned by removing punctuation, whitespace, digits, and preserving only letters (including Albanian characters ë and ç).
- **N-gram Generation**:
  - For n = 0, treat every character equally (simulate a uniform distribution).
  - For n ≥ 1, extract contiguous substrings of length n (sliding window).
- **Entropy Calculation**:
  - For n = 0, entropy is calculated using uniform probability over unique characters.
  - For n = 1, use standard Shannon entropy.
  - For n ≥ 2, compute **conditional entropy**: the uncertainty of the next character given the previous n-1 context.

1.3 Data Structures Used

- **String**: Holds the cleaned version of the input text.
- **List**: Stores n-grams.
- **Counter (collections module)**: Tracks frequencies of characters and n-grams efficiently.
- **Dictionary**: Used for mapping (n-1)-grams to a Counter of possible next characters.

1.4 Algorithm (Pseudocode)

```
function calculate_entropy(text, n):
  if n == 0:
    return -len(unique_chars) * p * log2(p)  // uniform distribution
  if n == 1:
    return -Σ p(char) * log2(p(char))
  else:
    for each (n-1)-gram context:
      for each next_char:
        compute conditional entropy
    return weighted average of all context entropies
```

1.5 Correctness Argument

- For $n=0$, the uniform entropy formula is mathematically valid.
- For $n=1$, Shannon entropy is calculated using observed frequencies.
- For $n \geq 2$, the algorithm computes entropy of next-character probabilities based on their context — consistent with the definition of conditional entropy.

1.6 Time Complexity

- **Text cleaning**: O(N)
- **N-gram generation**: O(N)
- **Entropy calculation**:
  - O(N) for $n = 0$ or 1
  - O(N) + O(V log V) for $n \geq 2$, where V = number of unique n-grams

1.7 Output Details

- Displays the calculated entropy.
- Number of total and unique n-grams.
- The five most and five least frequent n-grams.

1.8 Observations

- Entropy typically **decreases** with increasing n, reflecting the growing predictability with longer context windows.
- Clean and sufficiently long text is essential for meaningful entropy measurements.

Part 2 - Commit Owners

2.1 Objective

To reconstruct a sequence of employee IDs from a concatenated weld string that represents commit ownership. The goal is to identify the decomposition with the **most commits** (i.e., the most IDs).

2.2 Methodology

- **Input**:
  - A .txt file containing employee records (format: id,surname,name).
  - A numeric string (weld string) representing a chain of commits.
- **Decomposition Strategy**:
  - Use recursive backtracking to find all valid decompositions of the weld string.
  - Store and return the sequence with the highest number of IDs.
- **Output**:
  - The sequence of matched employee records corresponding to the best decomposition.

2.3 Data Structures Used

- **Dictionary**: Maps employee IDs to (surname, name)
- **List**: Used to keep the current path of decomposition
- **Recursive Call Stack**: Tracks position in weld string and backtracks if necessary

2.4 Algorithm (Pseudocode)

```
function find_all_decompositions(weld, pos, current):
  if pos == len(weld):
    return [current]
  for emp_id in employee_ids:
    if weld[pos:].startswith(emp_id):
      current.append(emp_id)
      recurse from pos + len(emp_id)
      current.pop()
return best decomposition with max len()
```

2.5 The correctness argument

- Algorithm explores **all possible decompositions** by trying every employee ID prefix at each position.
- Because it backtracks after each attempt, it guarantees **no valid path is missed**.
- Selecting the decomposition with the maximum number of commits ensures the optimal solution.

2.6 Time Complexity

- Worst-case: O(2^N) where N = length of weld string (due to all possible splits)
- Practical: Efficient for short welds or when IDs are not too numerous

2.7 Usage

main.bash

```bash
python commit_owners.py employees.txt 1234567890

# Ensure employees.txt is formatted correctly and IDs exist in weld.
```

2.8 Observations

- The longest valid decomposition doesn't necessarily use the longest IDs.
- Multiple decompositions are possible; the program returns the one with **most** splits (most commits).

Conclusion

This project demonstrates a practical application of entropy analysis and recursive search to solve language modeling and sequence reconstruction problems.

- Part 1 shows how information theory can quantify the structure of language.
- Part 2 illustrates how dynamic search strategies can recover information from compressed formats (like ID welds).

The scripts are clean, efficient, and well-commented to ensure clarity and extensibility.