Report: Assignment 5

Overview

This report describes the implementation of a product search application built using Flask. The application allows users to input a query and returns relevant product information based on the fuzziness of the search. The similarity between the query and product attributes (name, category, description) is calculated using the Levenshtein distance algorithm.

Features

- **Search Query:** The user can enter a search query to find relevant products.
- **Fuzziness Threshold:** A threshold can be set to determine the minimum acceptable similarity between the query and product attributes.
- **Ranking:** Products are ranked based on the similarity score between the query and their attributes.

Code Structure

Flask Application Setup

The application is initialized using the `Flask` class, and it serves a simple web interface for querying products.

```
from flask import Flask, render_template, request

app = Flask(__name__)
```

Helper Function: `calculate_string_distance`

A helper function is implemented to calculate the Levenshtein distance (edit distance) between two strings. This distance is used to determine the similarity between the search query and product attributes.

```
def calculate_string_distance(str1, str2):
    len_str1 = len(str1)
    len_str2 = len(str2)
    matrix = [[0] * (len_str2 + 1) for _ in range(len_str1 + 1)]

    for i in range(len_str1 + 1):
        matrix[i][0] = i
    for j in range(len_str2 + 1):
        matrix[0][j] = j

    for i in range(1, len_str1 + 1):
        for j in range(1, len_str2 + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else 1
            matrix[i][j] = min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
matrix[i - 1][j - 1] + cost)
```

```
    return 1 - matrix[len_str1][len_str2] / max(len_str1, len_str2)  # Returns
similarity ratio (0-1)
```

Product Data Loading: `load_product_data`

This function loads product information from a text file (`products.txt`). Each product is stored with its `id`, `name`, `category`, and `description`.

```
def load_product_data(file_path):
    """Load product data from a txt file."""
    products = []
    with open(file_path, 'r') as file:
        for line in file.readlines():
            product_info = line.strip().split('|')
            product = {
                'id': int(product_info[0].strip()),
                'name': product_info[1].strip(),
                'category': product_info[2].strip(),
                'description': product_info[3].strip()
            }
            products.append(product)
    return products
```

Membership Scoring: `calculate_membership_scores`

This function calculates the similarity score for a product by comparing the query with the product's `name`, `category`, and `description`. The highest similarity score is returned as the product's relevance score.

```
def calculate_membership_scores(query, product):
    """Calculate membership scores for a product based on query and product
attributes."""
    name_score = calculate_string_distance(query, product["name"])
    category_score = calculate_string_distance(query, product["category"])
    description_score = calculate_string_distance(query, product["description"])

    # Return the highest similarity score as product relevance
    return max(name_score, category_score, description_score)
```

Query Processing: `process_query`

The `process_query` function processes the user query by comparing it with all products. It filters products based on the fuzziness threshold and ranks the products by their relevance scores.

```python
def process_query(query, products, threshold=0.5):
    """Process the query and return ranked results based on fuzziness
threshold."""
    results = []
    for product in products:
        relevance = calculate_membership_scores(query, product)
        if relevance >= threshold:
            results.append({"product": product, "relevance": relevance})
    # Sort the results by relevance score
    return sorted(results, key=lambda x: x["relevance"], reverse=True)
```

Main Route: `index`

The main route (`/`) handles both GET and POST requests. On GET requests, it simply loads the product data. On POST requests, it processes the user query and returns the ranked results based on the threshold value.

```python
@app.route("/", methods=["GET", "POST"])
def index():
    products = load_product_data('products.txt')
    results = []
    threshold = 0.5  # default threshold

    if request.method == "POST":
        query = request.form.get('query')
        threshold = float(request.form.get('threshold', 0.5))  # fuzziness
threshold
        results = process_query(query, products, threshold)

    return render_template("index.html", results=results, threshold=threshold)
```

Running the Application

The application is run with the Flask development server in debug mode.

```python
if __name__ == "__main__":
    app.run(debug=True)
```

HTML Template: `index.html`

The `index.html` file renders the search interface and displays the search results. It contains an input field for the query and threshold, as well as a section to display the ranked products with their relevance scores.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Product Search</title>
</head>
<body>
```

```
    <h1>Product Search</h1>
    <form method="POST">
        <label for="query">Search Query:</label>
        <input type="text" name="query" id="query" required>
        <label for="threshold">Threshold (0-1):</label>
        <input type="number" name="threshold" id="threshold" value="{{ threshold
}}" step="0.1" min="0" max="1">
        <button type="submit">Search</button>
    </form>

    {% if results %}
        <h2>Search Results:</h2>
        <ul>
            {% for result in results %}
                <li>
                    <strong>{{ result.product.name }}</strong> (Relevance: {{
result.relevance }})<br>
                    Category: {{ result.product.category }}<br>
                    Description: {{ result.product.description }}
                </li>
            {% endfor %}
        </ul>
    {% endif %}
</body>
</html>
```

Conclusion

This Flask-based product search application allows users to query products with varying degrees of fuzziness. By leveraging the Levenshtein distance algorithm, the app compares product attributes to the search query and ranks them based on similarity. This simple yet effective tool provides a powerful way to search and discover products from a large dataset.

Future Improvements

- **Database Integration:** Replace text file storage with a database (e.g., SQLite or PostgreSQL) for better scalability.
- **Advanced Search Options:** Add more filters for users to refine their search results (e.g., by price, rating, etc.).
- **Performance Optimization:** For large datasets, consider using a more efficient text search algorithm or indexing mechanism (e.g., Elasticsearch).