# Matching Algorithm Documentation

## Algorithm Overview

### 1. Explanation of Matching Logic

The matching algorithm is designed to connect users based on compatibility by analyzing multiple factors such as hobbies, interests, location, and personality traits. Each factor contributes to a compatibility score calculated between two users.

- Key Steps in the Logic:  
 1. Collect user data (e.g., hobbies, interests, location, etc.).  
 2. Normalize and assign weights to the factors.  
 3. Compute individual factor scores between users.  
 4. Aggregate scores to calculate a final compatibility score.

### 2. Factors Considered and Their Weights

The algorithm takes the following factors into account:

|  |  |  |
| --- | --- | --- |
| Factor | Weight (%) | Description |
| Location | 25% | Proximity between users. |
| Hobbies | 20% | Overlap in recreational activities. |
| Interests | 20% | Alignment in areas of passion or engagement. |
| Personality Traits | 15% | Matching based on compatible personality types. |
| Education Level | 10% | Similarity in academic background. |
| Occupation | 10% | Compatibility in professional environments. |

### 3. Mathematical Formulas

Similarity Score for Each Factor:  
S\_f = Number of Matches in Factor / Total Elements in Factor  
  
Final Compatibility Score:  
C = Σ (W\_i \* S\_{f\_i})  
  
Where:  
- C = Compatibility score  
- W\_i = Weight assigned to factor i  
- S\_{f\_i} = Similarity score for factor i

## Technical Implementation

### 1. Code Structure

- Main Components:  
 - Data Collection: Handles user input and stores profiles.  
 - Normalization: Processes raw data for uniform comparisons.  
 - Matching Logic: Implements the weighted algorithm to calculate scores.  
 - Recommendation Engine: Returns a ranked list of compatible matches.

- Key Functions:  
 - normalize Data(data): Ensures data consistency across users.  
 - calculate Similarity (userA, userB): Computes factor-level similarity.  
 - compute Compatibility (userA, userB): Aggregates similarity scores with weights.

### 2. Libraries Used and Why

- NumPy: For efficient mathematical operations and data handling.  
- Pandas: For structured data processing and analysis.  
- Scikit-learn: For optional machine learning enhancements, such as clustering similar user groups.  
- Flask/Django (if applicable): Backend framework to handle API requests.

### 3. Performance Considerations

- Use caching (e.g., Redis) for frequently accessed matches.  
- Optimize database queries using indexing and partitioning.  
- Asynchronous task queues (e.g., Celery) for handling large-scale calculations.

## Future Improvements

### 1. What Would You Do Differently with More Time?

- Implement a machine learning model for dynamic weight assignment based on user behavior.  
- Add user feedback to refine matching logic over time.  
- Use NLP to analyze text-based user inputs for richer insights.

### 2. Scalability Considerations

- Transition to distributed databases like Cassandra or DynamoDB for scaling.  
- Employ microservices architecture to decouple components.  
- Use load balancers and auto-scaling groups for high traffic handling.

### 3. Additional Features

- Advanced Filtering: Allow users to prioritize certain factors.  
- Real-Time Updates: Notify users of new potential matches dynamically.  
- Social Graph Integration: Use graph databases to analyze and suggest matches based on friend-of-a-friend relationships.