Restaurant Finder:
Final Testing Report

Background:
As part of testing the application, several unit tests focused on testing the individual components of restaurant fetching and the algorithm computations have been conducted. These tests cover a range of scenarios, including missing data and highly divergent inputs. All testing functions can be found in test_algorithm.py

A. Sanity Test

Objective: Ensure the application can return a valid list of restaurants for a standard meeting.

- **Test ID:** `test_propose_restaurants_basic`
- **Input:** Valid meeting ID with multiple members and populated restaurant data
- **Expected Result:** List of recommended restaurants ordered by TOPSIS score.
- **Actual Result:** Passed – Restaurants returned with valid `ID`, `name`, and `rating`.
- **Mitigation Implemented:** Multiple checks and error messages have been integrated to identify whether the restaurants were fetched and written to the main database.

Meeting ID: 652234

Recommended Restaurants:

- Zaddy's by Kaplan (Rating: 4.70)
- Burgermeister Zoo (Rating: 4.30)
- McDonald's (Rating: 3.60)
- Risa Chicken (Rating: 4.50)
- AERA (Rating: 4.40)
- Goldie's Smashburger. (Rating: 4.30)
- Five Guys Berlin Ku'Damm (Rating: 4.00)
- KFC (Rating: 3.50)
- Luardi Cucina della mamma (Rating: 4.80)
- Upper Burger Grill (Rating: 4.70)
- Ming Dynastie West Berlin (Rating: 4.60)
- The Dawn (Rating: 4.30)
- Hard Rock Cafe | Berlin (Rating: 4.30)

B. Missing Data Scenario

No Restaurants Available

Objective: Validate the system's behavior when no matching restaurants exist (e.g., all are filtered out).

- **Test ID:** `test_propose_restaurants_no_candidates`
- **Input:** Meeting ID with strict filters or no inserted restaurant data.
- **Expected Result:** Return an empty list, or raise an informative exception.
- **Actual Result: Failed** – `ZeroDivisionError` when no members are present.
- **Mitigation Implemented:** Add a check in `Group.calculate_group_preferences()` for zero members and raise `ValueError`.

Case of One Restaurant

Objective: Handle the case where only one restaurant matches for a meeting ID, given the preferences entered.

- **Behavior in Code:** If both `start_price` and `end_price` are `None`, fall back to `price = 50` only if it's the *only* option.
- **Expected Result:** That single restaurant should still be processed.
- **Actual Result:** Passed- Restaurant returned with valid `ID`, `name`, and `rating`.

Meeting ID: 962276

Output: 1 Restaurant was recommended based on all preferences:
- My deli love
- Budget: 10 Euros
- Cash-only: Yes
- Serves Vegetarian Food: Yes

Edge Case: Null or Missing Fields

Objective: Ensure restaurant data gets written to the database even when attributes like rating, start_price, end_price, and description are missing after the API fetch.

- **Code Handling:** Safeguards such as `restaurant.rating, or 0,` and `try-except` for `description`.
- **Expected Result:** No crash, fallback scores applied.
- **Actual Result:** Passed

- **Mitigation Implemented:** the data pipeline has been pre-coded to ensure that all attribute column gets created regardless of their being present for all restaurants fetched from Google Maps. In such cases, the column is retained with all NA values in the final Restaurants Table.

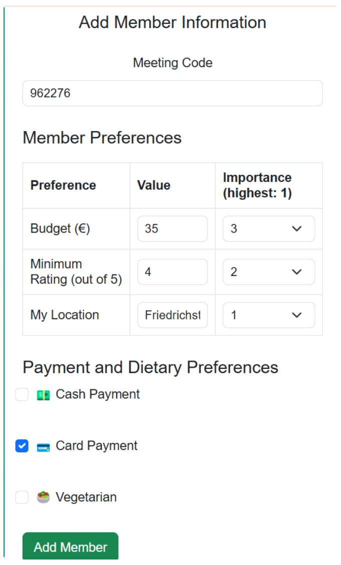
C. Handling extreme values

Case: Highly Conflicting Preferences Within Group

Objective: Evaluate behavior when group members have extreme and divergent preferences for budget, location, and ratings.

- **Expected Result**: Group preferences should average out and still produce stable recommendations.
- **Expectation:** May not be optimal if both preferences and the actual values of the parameters are too divergent
- **Actual Result**: Passed – TOPSIS algorithm still ranks restaurants based on the normalized score matrix.
- **Mitigation Implemented:** Because all features are normalized in the algorithm, extreme values from any one criterion are scaled down in influence. Please note that the user interface does not accept the member form until preferences for budget, rating, and location are different.

Inputs: The screenshots show the three divergent inputs used to test the case:


Add Member Information

Meeting Code

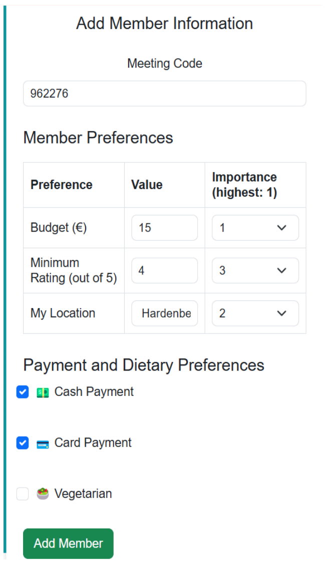
962276

Member Preferences

Preference	Value	Importance (highest: 1)
Budget (€)	35	3
Minimum Rating (out of 5)	4	2
My Location	Friedrichst	1

Payment and Dietary Preferences

- [ ] Cash Payment
- [x] Card Payment
- [ ] Vegetarian


Add Member Information

Meeting Code

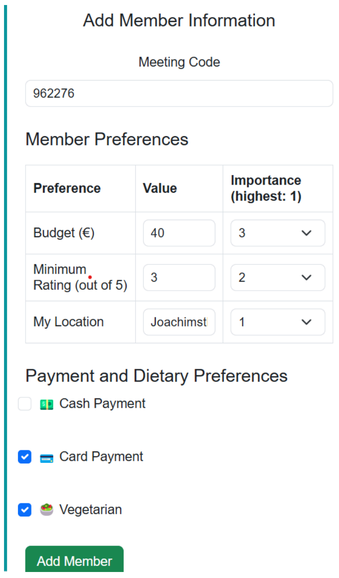
962276

Member Preferences

Preference	Value	Importance (highest: 1)
Budget (€)	15	1
Minimum Rating (out of 5)	4	3
My Location	Hardenbe	2

Payment and Dietary Preferences

- [x] Cash Payment
- [x] Card Payment
- [ ] Vegetarian


Add Member Information

Meeting Code

962276

Member Preferences

Preference	Value	Importance (highest: 1)
Budget (€)	40	3
Minimum Rating (out of 5)	3	2
My Location	Joachimst	1

Payment and Dietary Preferences

- [ ] Cash Payment
- [x] Card Payment
- [x] Vegetarian

Output: 1 Restaurant was recommended based on all preferences:

- My deli love
- Budget: 10 Euros

- Cash-only: Yes
- Serves Vegetarian Food: Yes

D. Filtering Options

Case: All Restaurant Prices exceed the Maximum Budget

Objective: Ensure restaurants are excluded if the group has `uses_card=True` and data indicates no card acceptance.

Handling Location: `data_pipeline.py` filters based on whether a restaurant accepts card payments

- **Expected Result:** Over-budget restaurants are excluded.
- **Actual Result:** Passed – Logic confirmed and validated via sample runs..
- **Mitigation:** Inform the user that all restaurants exceed the given maximum budget, but they are still being proposed after assessing based on other preferences

Case: Filtered based on availability of Cash/Card Payments

Objective: Ensure restaurants are excluded if they do not accept cash/card payment, but the user has given either of those.

Handling Location: `data_pipeline.py` filters based on whether a restaurant accepts card payments

- **Expected Result:** Restaurants missing card fields or not accepting cards are excluded from the list.
- **Actual Result:** Passed – Verified from logs and exclusions.
- **Mitigation:** Have included a print command within the data pipeline to track whether restaurants from the API fetched set were excluded because of cash/card payment being False.

Case: Filtered based on availability of Veg Options

Objective: Ensure restaurants are excluded if they do not offer vegetarian cuisine, even when one of the users in the group has specified it as a preference.

Handling Location: `data_pipeline.py` filters based on whether a restaurant accepts card payments

- **Expected Result**: Restaurants missing card fields or not accepting cards are excluded from the list.
- **Actual Result**: Passed – Verified from logs and exclusions.
- **Mitigation:** Have included a print command within the data pipeline to track whether restaurants from the API fetched set were excluded because of veg_options being False.