DS Cupid Technical Assessment Documentation

Repository Setup

The first step was to create a new repository, DS-Cupid, and clone it locally for development. A new Conda environment was set up using the command `conda create -n cupid python=3.11`. The choice of Python 3.11 was based on its stability and broad package support. To simplify the process, both datasets were downloaded and stored in the `Data` folder for easy access.

Data Analysis

The initial phase of the project involved studying the provided datasets in the notebook `01_data_analysis.ipynb`. From the analysis, it was observed that each dataset corresponded to a distinct supplier. The first supplier's dataset contained columns such as `hotel_id`, `lp_id`, `room_id`, and `room_name`, while the second supplier's dataset included `core_room_id`, `core_hotel_id`, `lp_id`, `supplier_room_id`, `supplier_name`, and `supplier_room_name`. A key finding was that the `lp_id` column serves as a unique identifier for a specific listing page associated with a hotel. Since this column was present in both datasets, it was used as the primary key to merge and match hotels from both suppliers.

However, not all hotels appeared in both datasets. Given that the assessment's objective was to automate the matching of hotels and rooms, any `lp_id` not found in both datasets was excluded from further analysis. After applying this filter, the dataset was reduced to 28,638 unique `lp_id`s.

Room Matching Implementation

To facilitate room matching, a second notebook, `02_reference_room_match.ipynb`, was created. This notebook focused on preprocessing room names and developing a method to match rooms efficiently. Normalization techniques were applied to create a more homogeneous representation of room names, which were then stored in a new variable called `processed_room_name`.

A function was developed to find matches between rooms by computing similarity scores. The function takes as input an `lp_id`, a reference provider dataset, an Expedia provider dataset, a room column, and a similarity threshold. Based on this information, it assigns scores to room name pairs and matches them if the score exceeds the defined threshold. The similarity scoring was implemented using **RapidFuzz**, a fast fuzzy string-matching

library. Several test cases were conducted to ensure that the function performed as expected.

API Development

For the API development, **FastAPI** was chosen due to its lightweight nature, ease of use, and widespread adoption in the industry. The first endpoint implemented was designed to replicate Cupid's reference room match API, which matches supplier room names to reference room names based on a given property ID. More details about the API usage can be found in the Swagger documentation.

Development Challenges

During development, one of the key challenges encountered was processing room names efficiently. Initially, a batch processing function, `preprocess_batch`, was created using **spaCy's `nlp.pipe()`**, which enables parallel text processing. However, processing all room names at once led to timeout errors due to excessive execution time. To address this, a simpler function, `preprocess_text`, was implemented. This function normalizes text by converting it to lowercase, removing numbers, punctuation, and stop words, and applying lemmatization. This approach significantly improved performance while maintaining comparable results.

Another challenge was optimizing data handling when merging rooms by hotel. Initially, the function `merge_rooms_by_hotel` processed both datasets in their entirety. However, since the API already received the supplier data in the request body, the function was modified to directly use this data, improving efficiency and reducing redundant computations.

Second API Endpoint: Room Match

The second API endpoint was implemented following Cupid's room match API specifications. This endpoint processes a list of supplier rooms and matches them against reference room names. To improve maintainability, the `ReferenceCatalog` class was modified to include `propertyName` and `referenceRoomInfo` as optional attributes. This allowed the `RoomData` class to be reused across both endpoints, ensuring consistency and reducing code duplication.

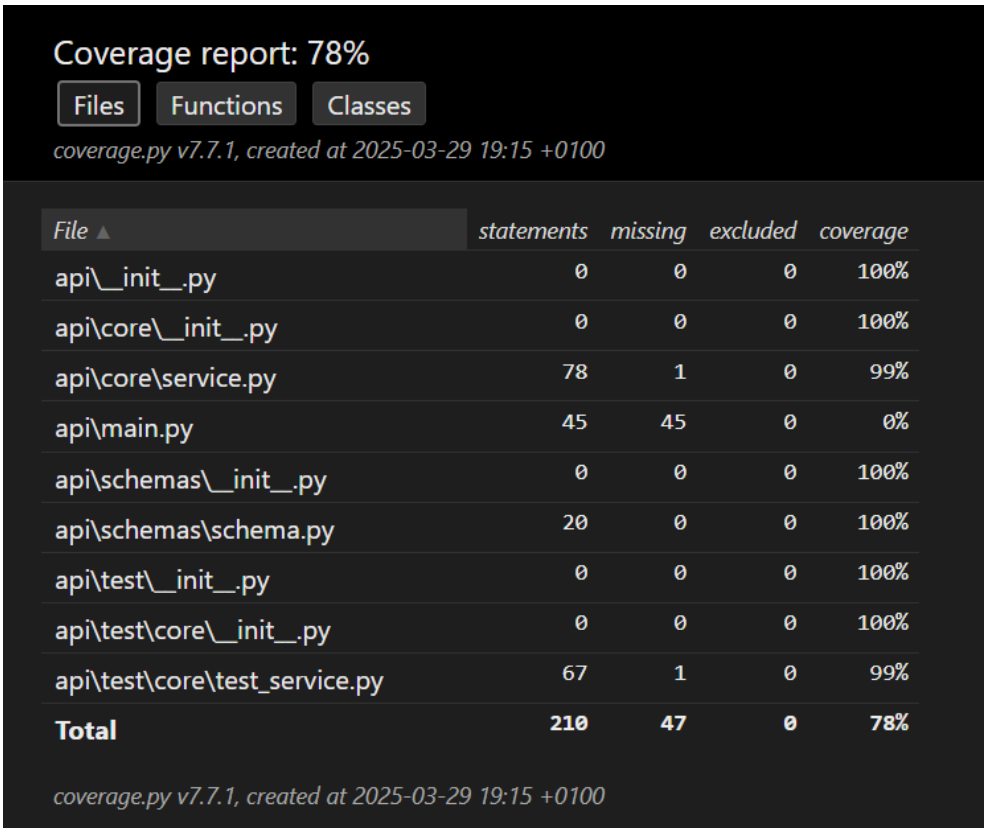
A further challenge arose when extracting matched and unmatched rooms. Initially, the function responsible for this task only returned two datasets: `mappedRooms` and `unmappedReferenceRooms`. To enhance clarity and maintainability, the logic was restructured into two separate functions: `get_matched_rooms` for matched room pairs and `get_unmatched_rooms` for unmatched rooms. This modular approach improved code readability, testability, and reusability.

Third API Endpoint: Bulk Room Match

The third and final endpoint implemented was designed to process multiple room match requests in batch mode, similar to Cupid's bulk room match API. Thanks to prior considerations in structuring reusable functions, implementing this endpoint was straightforward, requiring only an iterative application of existing functions.

Testing and Code Coverage

To validate the core logic, unit tests were written for functions within `service.py`. Tests were executed using the command `pytest api/test`, and all seven tests passed successfully. Additionally, test coverage was analyzed using the command `pytest --cov=api --cov-report=html`, revealing a **78% test coverage**.



Swagger documentation was configured to facilitate API exploration. The Swagger UI is available at `http://127.0.0.1:8000/docs` when running the application locally.

Final Steps

To ensure a smooth deployment process, a `requirements.txt` file was generated to track package dependencies. Additionally, a `README.md` file was created with instructions on setting up and using the API.

Next Steps

One of the next steps involves automating the preprocessing of the reference file by implementing a **Dagster** pipeline. This pipeline will monitor changes in the reference file and trigger preprocessing whenever an update is detected. A listener will be implemented to track file updates, and the processing logic will mirror the steps in `04_data_processing.ipynb`. By automating this process, room name normalization will be more efficient and reliable.

Another area for improvement is the enhancement of room name matching. One approach is to explore more advanced models to process room names in a more generalized manner, increasing the likelihood of correct matches. Alternatively, a precomputed dictionary containing all previously processed room names could be created. If a room name is found in this dictionary, its corresponding processed value can be retrieved instantly, reducing real-time computation needs. If a room name is not in the dictionary, it would be processed on the fly and added to the dictionary for future reference.

This documentation outlines the development process, encountered challenges, and potential future improvements, providing a comprehensive overview of the project.