Conversational Task Agent

Francisco Barreiras nº67208
Francisco Silva nº57824
Tomás Carvalho nº67209

NOVA SST

Abstract. This project involves developing a task-oriented bot designed to assist users through complex, multi-step tasks like baking a birthday cake or repairing a car scratch, with a unique capability to dynamically adjust instructions based on the user's available resources and tools. Should a user encounter a situation where they lack a necessary ingredient or tool, the taskbot is engineered to reconfigure the task plan on-the-fly, offering alternative solutions to ensure the task can still be completed successfully.

Keywords: task-oriented bot · dynamically adjust · alternative solutions.

1 Introduction

This project consists in the development of a task-oriented bot with the main goal of helping the user to perform a task like baking a cake, in case of adversity (like not having the required ingredients for cooking the recipe), it should provide alternative paths. For the implementation of such project, we have used OpenSearch due to its capabilities in realms of full-text search and analytics, and its ability to scale to billion size documents. The first phase consists of defining the structure of the index mappings and how the information is stored. The second phase consists of extending the previous phase with CLIP image and text embeddings, allowing searches by image and text and including a new module and interface to the application that communicates with the PlanLLM model to assist the user in the completion of the desired recipe.

2 Architecture

The base architecture of the system is comprised by 3 modules and a main python notebook, used to control the indexing ans search modules. On top of this, a simple command interface was built to interact with the search module as well as the PlanLLM model. The Indexing.py module's purpose is to handle the index operations, might these be to calculate the embeddings and interact with the pickle files used to store the embeddings, to create the index and mappings or to parse and store the recipes from the base file to OpenSearch. The Search Builder

API is meant to be used as a tool to easily and in an iterative manner build simple or complex search OpenSearch queries. This module avoids repetitive code, also as a way to ensure good application scaling and coding practices and maintaining a developer-friendly interface. The Search.py module is simply an abstraction of the SearchBuilder.py module, made to simplify the search operations for a user. It's easily scalable thanks to the SearchBuilder layer and was made to be used as a simple interface for the system.

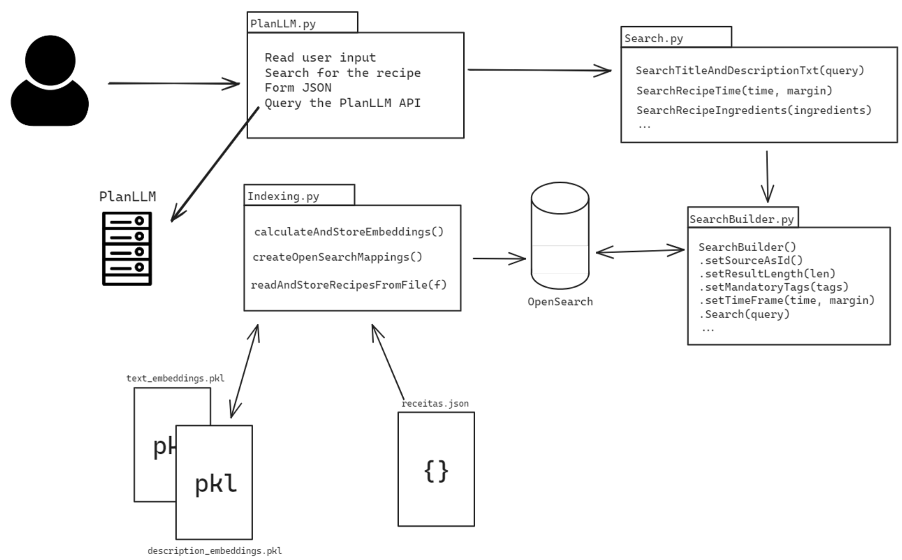

Fig. 1. Architecture of the system.

3 Algorithms and Implementation

3.1 Mappings and reasons behind them

Mappings define how data is stored, indexed, and searched within an index. They are essentially the schema definition, specifying the data types for each field in your documents and how those fields should be treated by OpenSearch. By specifying the data type for each field (such as text, keyword, date, integer, etc.), OpenSearch can store data more efficiently, improving both storage usage and search performance. In our work we implemented the following mappings:

– **doc_id: keyword -** used for fields that to be indexed as a whole, without breaking them down into separate searchable terms ,in this specific case, the id of the document.

- **tags: keyword -** used to tag each document according to its 'diets', 'courses' and 'cuisines'. Will be used to search these tags in a deterministic way, meaning the tag is to be exactly the ones used in the initial document, useful to filter recipes

- **title: text -** chosen for fields that contain full text, such as descriptions, article content, or any other form of unstructured text that benefits from full-text search capabilities.

- **title_embedding: knn vector -** performs searches to find documents that are "closest" to a given vector, created specifically for title embedding representations.

- **description: text -** enables complex search queries, including matching on individual terms within the field.

- **description_embedding: knn vector -** used to find the "closest" description embedding vectors to the given query.

- **clip_embedding: knn vector -** holds recipe images vector embeddings or the recipe caption vector embeddings if no image is available. These vectors were computed by a model that generates similar vectors for images and their captions.

- **ingredients: text -** used to perform searches by similarity, meaning a score will be calculated to match the documents in which its ingredients best match the given query.

- **instructions: nested -** used essentially to retrieve and parse information to be later used by the PlanLLM model, for recipe planning.

- **time: integer -** used to perform queries based on numeric ranges, such as finding all documents where a certain numeric value falls between two bounds. Used to find recipes that take a given time with a given range

- **difficultyLevel: keyword -** used to search for recipes that have a specific difficulty, its a keyword in order to be used as a filter.

- **contents: text -** a non-indexable field, used to store recipe information that is not to be searched by, but can be retrieved.

3.2 Supported Searches

OpenSearch excels in conducting searches within datasets, offering a robust set of features tailored for both simple and advanced search queries. This capability is foundational to its utility as a search and analytics engine.

We opted to join the recipe name and description when searching a recipe by query to have the best possible results, since we considered that some searches might not appear on a recipe's name, but rather be specified in the description.

Since we implemented the SearchBuilder module, it's possible to make a broad amount of queries just by assembling the builder as pleased, this means that any combination of parameters can be used to search for a recipe. In order to give use-case examples of our SearchBuilder as well as a straight forward way to test the system, we implemented 9 methods in the Search module that exemplify possible and useful queries for a searching system like the one implemented

– **SearchTitleAndDescriptionTxt** - provides the capability of searching for a recipe by title and description by text matching.

– **SearchTitleAndDescriptionTxtInstructions** - provides the capability of searching for a recipe by title and description by text matching, retrieving its id, name and instructions. Used with the PlanLLM.

– **SearchRecipeTime** - provides the capability of searching for recipe by its time within a range, not considering the name or description.

– **SearchRecipeNameTime** - provides the capability of searching for recipe by its time within a range, having in account text matching by the name and description.

– **SearchRecipeIngredients** - provides the capability of searching for a recipe by ingredients.

– **SearchRecipeExcludeIngredients** - provides the capability of searching for a recipe that does not have the given ingredients.

– **SearchRecipeDifficulty** - provides the capability of searching for a recipe by its difficulty.

– **SearchRecipeNameDifficulty** - provides the capability of searching for a recipe by its difficulty, matching the query to its name and description.

– **SearchTitleEmbeddings** - provides the capability of searching for a recipe by title embeddings.

– **SearchDescriptionEmbeddings** - provides the capability of searching for a recipe by description embedding.

– **SearchByImage -** provides the capability of searching for a recipe from an image. (also works on recipes without images).

3.3 Large Language Model

In order to facilitate the task completion by users, an interface was implemented. This interface, at first, serves as a searching point for the desired recipe, but after the recipe is chosen and the context is set, its job is to communicate with the PlanLLM, which is a Large Language Model, trained specifically to assist the user to complete a task and plan a strategy. The PlanLLM was made available by the professor through an API, which had a structured and an unstructured endpoint. We used the structured endpoint which makes it so that the conversation data can be sent as a JSON, which makes it convenient and easier to work with.

The JSON structure is as follows:

```
{
    "dialog_id": "1",
    "system_tone": "neutral",
    "task":{
       "recipe":{
          "displayName":"Chocolate Cake",
          "instructions":[
             {
                "stepText":"Preheat oven to 350 degrees"
             }
          ]
       }
    },
    "dialog":[
       {
          "current_step":0,
          "user":"Let's start the task",
          "system":"Hello, how can I help you today?"
       }
    ]
 }
```

Using this structure it is then possible to programmatically build a base JSON and update it as the conversation goes.

The PlanLLM.py module is divided in smaller functions that clearly state how the conversation flow is implemented, namely:

```
- Main
  - get_initial_user_input()
```

```
- search_Opensearch(recipeToSearch)
- generate_base_json(recipe_json)
- Loop
  - get_user_input("What to do now?\n")
  - add_to_json(...)
  - send_to_planllm(...)
  - add_to_json(...)
  - print(ai_response)
```

Annex 1 presents a demonstration of a conversation with some particular feats, namely, going some steps back, some steps to the front, getting the name of the recipe and a quit/stop/exit/end command to finish the conversation.

4 Dataset Description

This dataset is comprised of 993 elements (recipes), each one of them is constituted by several fields but in our work we will work only with the following:

- displayName - also known as the title of the recipe.
- description - the overview of the recipe and how you can use it.
- totalTimeMinutes - the time it takes to prepare it.
- ingredients - a list of the requirements for the recipe.
- instructions - the list of instructions to cook the recipe.
- difficultyLevel - how hard it is to prepare the dish.
- cuisines - a list of the kind of gastronomies in which this recipe is inserted in.
- courses - a list of the kind of meals in which it is usually eaten.
- diets - a list of the diets for which this recipe is fit for.

5 Contextual Embeddings and Self-attention

5.1 Contextual Word Embeddings

In Deep Neural Networks, input parameters are collected in nodes at the input layer and then mapped to output nodes. Hidden layers in between transform input embeddings to adjust input weightings for different tasks.

By plotting the contextual word embeddings on each layer we can observe how each hidden layer manipulates the vectors.

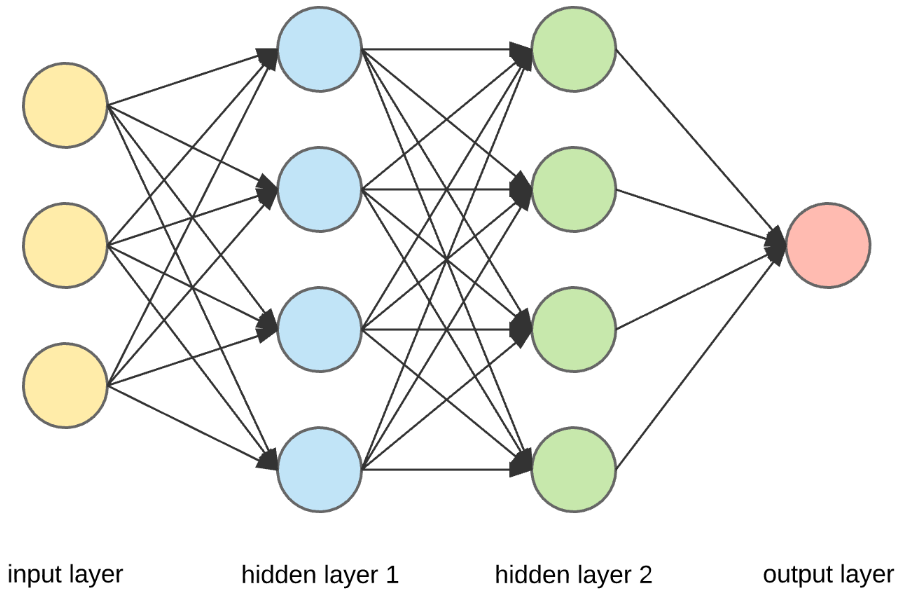

Fig. 2. Components of neural network.

Looking through the sequence of layers we could observe positional shifts between the contextual word embeddings, but mainly, how words cluster together differently in each layer.

For example for the text: "I love a good home made chicken recipe prusciato recipe". The 3rd layer has the "love" and "good" word vectors close (most likely because they have close meanings), but on the last layer, "good", "home", "chicken", "recipe" word vectors are clustered together and the "love" vector is far apart.

5.2 Positional Embeddings

Using a simple BERT encoder and inserting a sequence of repeated words like "pasta pasta pasta ..." outputs a sequence of with different embeddings even though they all represent the same word. **why?**

Theoretically all instances of the word "pasta" should have the same embedding. This is commonly known as the "bag of words" problem, where models fail to capture the semantic relationships and context between words in a document.

For this reason, many models like the BERT encoder use **Positional Embeddings** to understand sequences or order-dependant nuances in language to give each token, though textually identical a unique embedding.

5.3 Sentence Embeddings

Sentence embedding algorithms are techniques used to represent sentences as dense numerical vectors in a high-dimensional space, capturing their semantic meaning. Unlike word-level embeddings, sentence embeddings capture the context of the entire sentence, making them powerful for understanding the overall topic or intent.

The laboratory implementation of sentence embeddings describes a method for generating sentence embeddings with a k-nearest neighbors (KNN) vector approach. This implementation outputs 768-dimensional embeddings that stores the semantic meaning of the corresponding sentences in a high-dimensional space. It takes advantage of ANN techniques like HNSW for indexing and querying sentence embeddings because of it's scalability and speed in searching large datasets.

5.4 Self-Attention

Self-attention is essentially a mechanism that allows models to weigh the importance of different parts of an input sequence when processing it. It's a way for the model to focus on different words or tokens in the input sequence as it processes it, assigning higher importance to some tokens while disregarding others. In the code, its provided an implementation for the cross encoder as well as the dual encoder.

Transformer cross-encoder
A cross-encoder processes pairs of inputs simultaneously, applying self-attention across all tokens in the combined input sequence. This approach is particularly effective for tasks where the relationship between the inputs (such as question-answering or text matching) is crucial.

The self-attention mechanism in a transformer cross-encoder introduces a significant computational overhead due to the need to compute attention scores for all pairs of tokens in the input sequence, but it allows the model to capture complex dependencies between tokens in the input sequence, making it possible to have long-range dependencies and capture context across the sequence.

In the code implementation its possible to see just one self-attention graph that joins the inputs from both sequences, as explained, and its possible to see relations between tokens of the two sequences.

Dual encoder
In a dual encoder architecture, there are two separate encoders, each processing one of the input sequences. The self-attention mechanism is applied within each encoder independently.

Since each input sequence is processed independently without interaction between tokens from two sequences during the self-attention step, it limits the capture of interactions between tokens from the two sequences, potentially leading to information loss. The characteristic that makes the dual encoder a good architecture is that it allows for parallel processing of the input sequences, potentially leading to faster training and inference times when compared to cross-encoder architectures.

In the code implementation of the dual encoder self-attention matrix its possible to see that there are two generated self-attention matrices, that are calculated independently of the other as, explained.

Final analysis

It is now possible to express some considerations about the generated self-attention matrices. The cross encoder matrix's weights vary from 0 to around 0.5 because, when using a cross-encoder architecture, the interactions between tokens potentially lead to more evenly distributed attention, lowering the individual attention weights.

6 References

References

1. OpenSearch. "OpenSearch Compound Query DSL - Bool." https://opensearch.org/docs/latest/query-dsl/compound/bool/ (Accessed: April 8, 2024).
2. OpenSearch. "OpenSearch Full-text Query DSL - Match." https://opensearch.org/docs/latest/query-dsl/full-text/match/ (Accessed: April 8, 2024).
3. OpenSearch. "OpenSearch Full-text Query DSL - Multi-match." https://opensearch.org/docs/latest/query-dsl/full-text/multi-match/ (Accessed: April 8, 2024).
4. OpenSearch. "OpenSearch Term Query DSL - Range." https://opensearch.org/docs/latest/query-dsl/term/range/ (Accessed: April 8, 2024).
5. Python. "Python pickle — Python 3 documentation." https://docs.python.org/3/library/pickle.html (Accessed: April 8, 2024).
6. OpenSearch. "OpenSearch Field Types." https://opensearch.org/docs/latest/field-types/ (Accessed: April 8, 2024).

7 Annex 1 - Example Conversation

Hello! What recipe would you like to cook today?

chicken curry

What to do now?

start

Ready, set, go! Step 1: Prepare the Basmati Rice according to package's instructions.

What to do now?

next step

Step 2: Parboil Yukon Gold Potato and Sweet Potato. Parboil means to partially cook in boiling water. Season a pot of cold water with plenty of kosher salt (1/2 tsp) and cook the potatoes until they are partially cooked, but not yet fork-tender2̆00b.

What to do now?

give me the previous step

Let's go back to the previous step, Step 1: Prepare the Basmati Rice according to package's instructions.

What to do now?

jump to step 4

We're halfway there! Step 2: Parboil Yukon Gold Potato and Sweet Potato. Parboil means to partially cook in boiling water. Season a pot of cold water with plenty of kosher salt (1/2 tsp) and cook the potatoes until they are partially cooked, but not yet fork-tender2̆00b.

What to do now?

jump tp step 4

Step 3: Peel and devein the Medium Shrimp. Rinse them and set aside in the refrigerator until you need them.

What to do now?

go two steps forward

We are already in the middle of the steps.

What to do now?

what do i do now?

Step 4: In a food processor, puree the Garlic, Fresh Ginger, Onion, Red Bird's Eye Chili Pepper, Lime, Cumin Seeds, and Whole Coriander Seeds.

What to do now?

continue

Step 5: In a medium pan over medium-high heat, heat up Canola Oil. Fry pureed mixture for 2-3 minutes. Add Ground Turmeric and saute for an additional 1 minute.

What to do now?

what is the last step of the recipe

Step 6: Add Water, Brown Sugar, Lime Juice, Lemongrass, Fish Sauce, and Kosher Salt. Simmer for 20 minutes.

What to do now?

what was the step number 3 again?

Sure! Let's go back to the previous step, Step 3: Peel and devein the Medium Shrimp. Rinse them and set aside in the refrigerator until you need them.
What to do now?
continue
Step 4: In a food processor, puree the Garlic, Fresh Ginger, Onion, Red Bird's Eye Chili Pepper, Lime, Cumin Seeds, and Whole Coriander Seeds.
What to do now?
conitue
Step 5: In a medium pan over medium-high heat, heat up Canola Oil. Fry pureed mixture for 2-3 minutes. Add Ground Turmeric and saute for an additional 1 minute. What to do now?
continue
Step 6: Add Water, Brown Sugar, Lime Juice, Lemongrass, Fish Sauce, and Kosher Salt. Simmer for 20 minutes. What to do now?
continue
Step 7: Add the Coconut Milk and simmer for an additional 5 minutes.
What to do now?
what is the first step?
Let's go back to the beginning and start over. Step 1: Prepare the Basmati Rice according to package's instructions.
What to do now?
give me the eight step now
Step 8: Add raw shrimp, parboiled vegetables and Green Beans to curry sauce and cook over medium heat until just cooked through.
What to do now?
continue
Step 9: Garnish with Fresh Thai Basil and serve over rice.
What to do now?
what is the name of the recipe again?
The recipe is Yellow Curry Shrimp.
What to do now?
continue
The final touch! Step 10: Well done! Want to save this recipe to your SideChef cookbook or meal plan? Try ̆201cAlexa, send to phone ̆201d.
What to do now?
is it over?
I believe we have reached the end of the task. Remember, you can always come back to revisit an old recipe or start a new project. Have a wonderful day and happy cooking!
What to do now?
quit
Goodbye!