June 12, 2024

Web Search and Data Mining

Pratical Report of the Conversational Task Agent

Doobie Bot

Francisco Barreiras	Tomás Carvalho	Francisco Silva
67208	67209	57824
MEI	MEI	MIEI

Prof. João Magalhães

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.

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 provides 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.

In the final phase, we enhanced the bot's interactivity and usability by implementing advanced features to manage conversation flow, accurately identify user intents, extract specific information from user inputs, and process visual input. These improvements broadened the range of tasks the bot can assist with and significantly enhanced user experience through more dynamic and multimodal interaction capabilities.

Architecture

The base architecture of the system is comprised by 5 modules and a main python notebook, mainly used to test each module's features individually. On top of this, a command interface was built to interact with the conversational task agent.
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.

IntentDetector.py is a module to simply instantiate the intent detector and defines a function to do the main feature of the class.

The QuestionImage.py module's purpose is to instantiate a class instance that, via a ask function, can answer questions about images using ViLT.

At last, the SlotFilling class also only exposes a method, that given a context and a question, provides an answer given by the defined model.

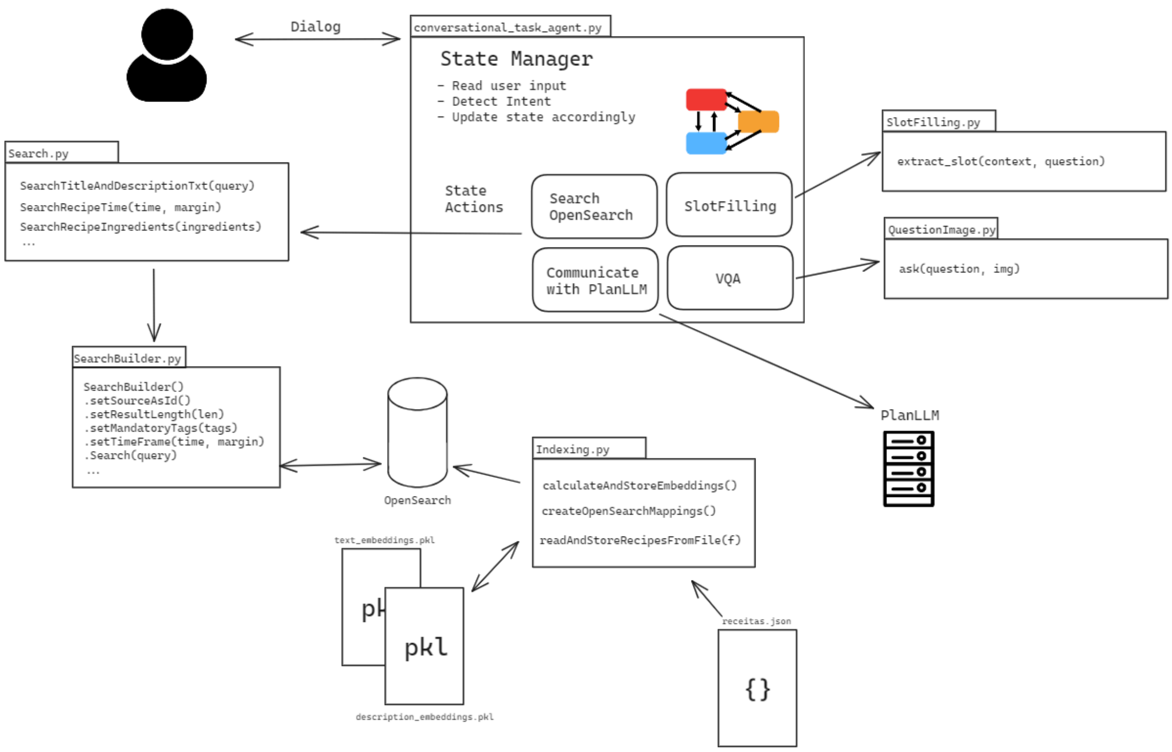

Figure 1: Architecture of the system.

Algorithms and Implementation

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.
- **images: nested** - list of image objects linked to a recipe. Each Image object contains the URL of the image.
- **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** - list of instruction steps containing each instructions info, image and step ID. 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.

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)

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.

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.
- images - the images belonging to the recipe.
- 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.

Contextual Embeddings and Self-attention

Contextual Word Embeddings

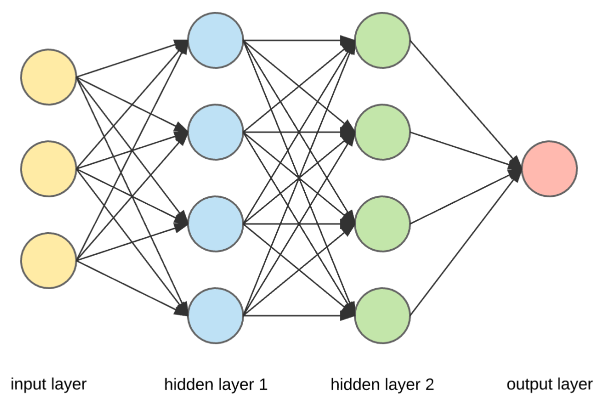

Figure 2: Components of neural network.

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.

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.

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.

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.

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.

Dialog Manager

In the third phase, we integrated a proper dialog manager to our conversational task agent. For this, we used the given and recommended Dialog Manager Framework that was implemented for TWIZ . This state machine fully controls the flow of the conversation between the user and the task agent, ensuring that the agent can handle user inputs effectively and transition between different states based on the detected intents.

State Machine Design

The state machine is composed of abstract base classes for events (AbstractEvent) and states (AbstractState). These base classes define the necessary methods for handling events and transitions. Specific states and events inherit from these base classes, implementing the required logic for entering and exiting states and validating events.

Components:

- **AbstractEvent and AbstractState**: These classes provide the structure for defining custom events and states. Each state can define its transitions and subflow transitions.
- **LaunchState and LaunchEvent**: These are the initial state and event that start the dialog. The LaunchEvent checks if the detected intent matches its ID and transitions to the LaunchState.
- **DialogManager**: This class manages the state transitions and maintains the history of states. It also handles triggering events and determining the next state based on the current state and event.

The DialogManager initializes with a starting state (LaunchState) and maintains a list of possible states and transitions. The trigger method processes incoming events, determines the next state, and manages the history and checkpoints for state transitions. Custom states like

ChooseTypeState and DisplayRecipeState implement specific logic for handling user interactions and transitioning between states.

The state machine's flow can be represented in a state diagram as follows.

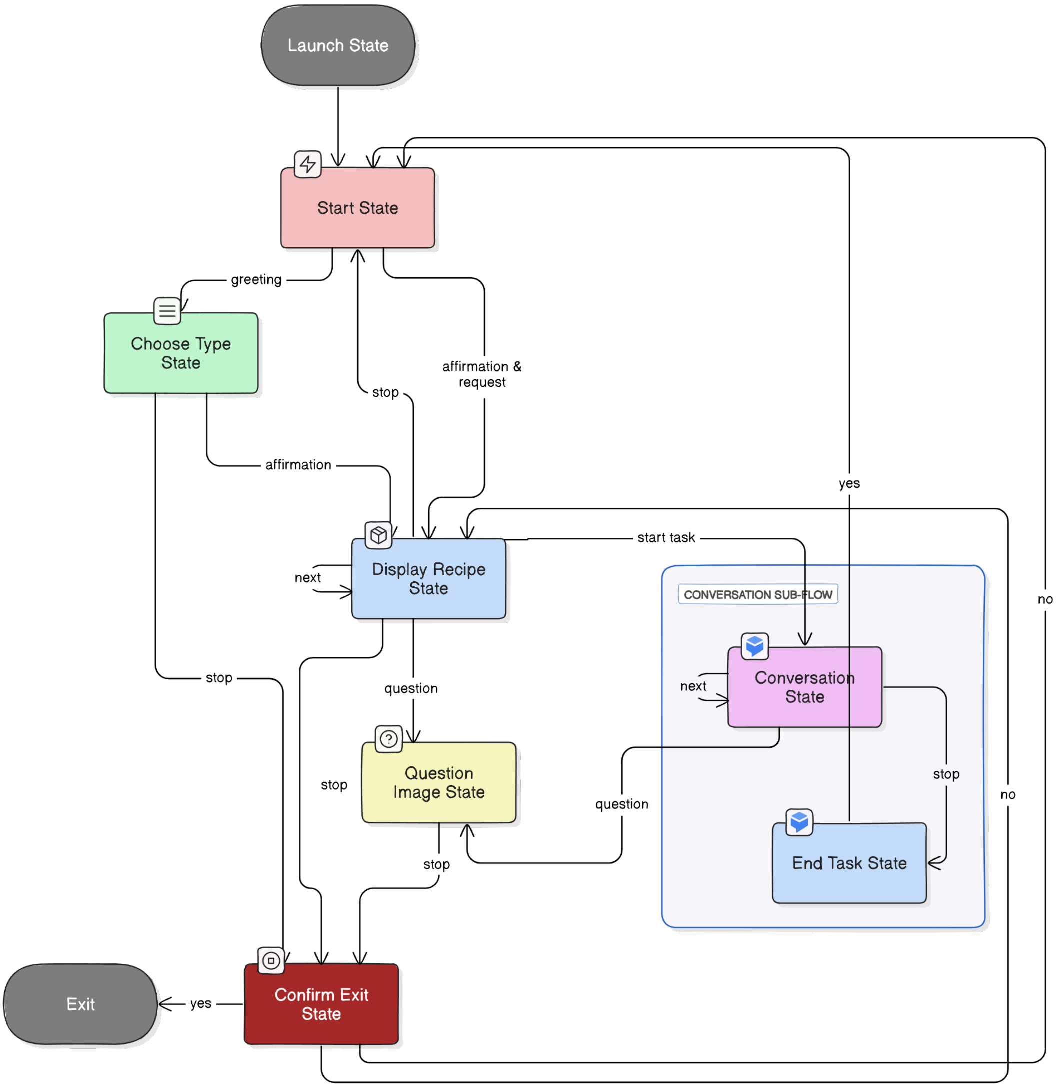

Figure 3: State machine flow diagram.

Dialog Intent Detector

The intent detection is implemented using a pre-trained model from the Hugging Face model hub [NOVA-vision-language/task-intent-detector](https://huggingface.co/NOVA-vision-language/task-intent-detector) . The detector uses the AutoModelForSequenceClassification and AutoTokenizer from the transformers library.

Since the intent detector was trained with a defined set of intents, we opted to use them, defined in the `all_intents.json` of the Hugging Face model page. To use these intents appropriately in our case, we created a function to map intents given by the intent detector, to events of our application.

It's worth noting the intent detection is working with a poor performance, a simple test "Greetings!", outputs a StopIntent from the intent detector model. This has proved to be a big stunt in the progress and conversation flow of the agent, since a lot of times the states do not progress as intended due to the wrong detected intent.

Because of this, the mappings between the intents and the events had to be broader and accept a wider range of possibilities.

The detect_intent method of the IntentDetector class tokenizes the input text, feeds it to the model, and retrieves the predicted intent based on the highest scoring class.

Slot Filling as QA

Slot-filling is implemented using a question-answering approach to extract specific information (slots) from the user's input. This method avoids the need for extensive annotation by leveraging pre-trained QA models. The slot-filling functionality relies on the deepset/roberta-base-squad2 model from the Hugging Face model hub, which is well-suited for extracting answers to specific questions from a given context.

The extract_slot method of the SlotFilling class formulates a QA input with the context and question, and uses the QA pipeline to extract the relevant slot value from the context.

How the components interact

In this section, the interactions between components will be described as well as the program general flow.

Each time the system requests input from the user, the system will try to find the user's intent behind the message, in order to choose which will be the next state. This will be done regularly until the conversation reaches the DisplayRecipeState via a ChooseType event, when this happens, the system needs to understand which recipe the user wants, and to do this, we apply slot filling by asking "What is the recipe?" and giving as context the user input. This is done to clean the uninteresting text given by the user and searching only by the wanted recipes. If the user input was not sanitized, the recipe search could be very compromised and in order to maintain a consistent conversation, we opted for this strategy.

Annex 1 presents a demonstration of a conversation with some particular features, namely, choosing a recipe from a list of recipes, starting a recipe, exiting that one and starting a new one, going some steps back, some steps to the front, getting the name of the recipe and a stop command to finish the conversation.

Group Specific Implementation

Question-Answering with Image (VQA)

Visual Question Answering (VQA) is the task of answering open-ended questions based on an image. The input to models supporting this task is typically a combination of an image and a question, and the output is an answer expressed in natural language.

We selected the [ViLT (Vision-and-Language Transformer)]() model from the Hugging Face hub which was fine-tuned specifically for VQA tasks because it's designed to handle tasks that require the integration of visual and textual information.

```
Doobie Bot: Hello, how can I help you search for a recipe?
You: Hello, I want to cook a devils food cake.
Doobie Bot: Here is the top suggested recipe: Devil's food cake. Feel free to ask for the next recipe.
```

```
You: start task.
```

Figure 4: Complimenting dialog with media

Visual Question-Answering features allow users to interact with media content by asking question's about its content. For **Doobie Bot**, a chat bot that helps users find recipes and execute them, it is crucial that users can engage with the media content found in recipes and instructional images.

```
Doobie Bot: Ready to dive in? Step 1: Preheat oven to 180. Place parts of the butter, espresso, honey, and
milk in a saucepan. Slowly heat until butter is melted.
```

```
You: What is being spilled inside the pan?
Doobie Bot: The answer is: milk. Feel free to ask me more questions.
```

Figure 5: Display of visual question answering

Baselines

In this section, we describe the algorithms and models used as baselines to address specific challenges in the project.

As stated before, we used external models and libraries, that were baselines for our project, in the first phase, we needed the transformers library, that allowed us to use a spectrum of tokenizers to encode text.

For the second phase, we used the CLIP model to calculate embeddings both for images and text, to enable comparisons between compatible tokens. The CLIPProcessor.

Building the Dialog Manager for the third phase of our project we used:

- A **Intent Detector** to capture the intentions behind the utterance of the user;
- **Slot Filling** to get the meaning of the user input through the given state context;
- A **Dialog Manager** to handle and control the flow of the conversation using a state machine system.

Finally we integrated a Visual Question Answering feature using **ViLT** to allow users to interact with the dialog media through questions and answers.

Results Discussion

The integration of various components led to a conversational task agent capable of assisting users with recipe-related tasks through both textual and visual interactions.

The dialogue management system, based on the TWIZ dialog manager framework, facilitated smooth conversation flow. The state machine design enabled the bot to handle user inputs effectively and transition between states based on detected intents and previous states.

Despite the advanced intent detection model, its performance was suboptimal, particularly with ambiguous or poorly phrased inputs. The model often misclassified intents, disrupting the conversation flow. This necessitated broader mappings between detected intents and corresponding events to mitigate misclassification impacts.

The system effectively handled various recipe-related queries such as navigating to the next or previous steps, retrieving the recipe name, and handling stop or exit commands. These capabilities are crucial for maintaining user engagement and ensuring the bot could assist users through multi-step tasks effectively.

Critical Discussion

The development and implementation of the task-oriented bot revealed several critical insights:

Strengths and Achievements

Multimodal Interaction

The successful integration of text and image-based search and QA capabilities provided a rich user interaction experience, making the bot versatile and user-friendly.

Dynamic Task Adjustment

The ability to dynamically adjust instructions based on available resources ensured that users could complete tasks even with missing ingredients or tools.

Structured Dialogue Management

The use of a state machine and structured JSON for managing conversations allowed for coherent and context-aware interactions, significantly enhancing user experience.

Limitations and Challenges

Searching

Even though we implemented not only a considerable amount of queries, but also a builder to virtually build any query, some of the search results by embeddings are not as exact as we expected.

Performance Optimization

The integration of advanced models for embedding search and VQA introduced computational overhead. Optimizing these processes to maintain real-time interaction capabilities is a big challenge and the response time overheads are to be considered.

References

OpenSearch. "OpenSearch Compound Query DSL - Bool." https://opensearch.org/docs/latest/query-dsl/compound/bool/ (Accessed: April 8, 2024).

OpenSearch. "OpenSearch Full-text Query DSL - Match." https://opensearch.org/docs/latest/query-dsl/full-text/match/ (Accessed: April 8, 2024).

OpenSearch. "OpenSearch Full-text Query DSL - Multi-match." https://opensearch.org/docs/latest/query-dsl/full-text/multi-match/ (Accessed: April 8, 2024).

OpenSearch. "OpenSearch Term Query DSL - Range." https://opensearch.org/docs/latest/query-dsl/term/range/ (Accessed: April 8, 2024).

Python. "Python pickle — Python 3 documentation." https://docs.python.org/3/library/pickle.html (Accessed: April 8, 2024).

OpenSearch. "OpenSearch Field Types." https://opensearch.org/docs/latest/field-types/ (Accessed: April 8, 2024).

Annex 1 - Example conversation

```
Doobie Bot: Hello, how can I help you search for a recipe?

You: Hello!
Doobie Bot: What type of recipe do you want?

You: i want to cook devil food cake please
Great option!
Doobie Bot: Here is the top suggested recipe:Devil's food cake. Feel free to ask
for the next recipe.

You: that sounds great! lets start the task
Doobie Bot: Let's get cooking! Step 1: Preheat oven to 180\u00b0C/355\u00b0F.
Place parts of the butter, espresso, honey, and milk in a saucepan. Slowly heat
until butter is melted.

You: lets proceed to the next step
Doobie Bot: Step 2: In a large bowl, whisk together cocoa powder, all-purpose
flour, sugar, baking powder, and seeds from the vanilla bean.

You: what is the tool the cook has in hand?
Doobie Bot: The answer is: whisker. Feel free to ask me more questions.

You: next step
Doobie Bot: Step 3: Beat eggs until slightly foamy. Add melted butter mixture.
Then slowly add in flour mixture and continue to beat until smooth.

You: next step
Doobie Bot: Step 4: Transfer dough to 2 springform pans. Bake in a preheated
oven at 180\u00b0C/355\u00b0F for approx. 25 min.

You: what is inside the glass bowl?
Doobie Bot: The answer is: bananas. Feel free to ask me more questions.

You: oh nice, thank you, can you tell me the next step?
Doobie Bot: Step 5: In the meantime, roughly chop chocolate. Heat heavy cream
in a saucepan. Next, combine hot heavy cream, chocolate, and remaining butter
in a large bowl. Stir until fully combined and set aside to cool for approx. 8
\u2013 10 min.

You: what is the cook doing?
Doobie Bot: The answer is: cutting cake. Feel free to ask me more questions.

You: take me to  the previous step
Doobie Bot: Alrighty! Here's the previous step again, Step 4: Transfer dough to
```

2 springform pans. Bake in a preheated oven at 180\u00b0C/355\u00b0F for approx.
25 min.

You: stop
Doobie Bot: Are you sure you want to stop the task?

You: yes