Conversational Shopping Agent - Web Search and Data Mining

Project Report

Research by:
Artur Stopa 65043
João Soares 57609
Ricardo Pereira 57912

Reviewed by:
João Magalhães (jmag@fct.unl.pt)

1. Introduction

The goal of this project was to use the UI from the code extension to receive input from the user interface and retrieve recommendations from the Farfetch database using the OpenSearch API and send the result back to the user.

1.1. Design

The code is organized in the following way:

- The **app.py** contains the Flask code to run the server and receive the HTTP request that are sent by the Chrome extension.
- The **controller.py** contains the methods that are required for search, which we discuss below.
- The **Encoder.py** is a class used by the controller.py in the cross-modal search method, and uses methods from the transformers and torch libraries. It's a class holding one instance of **CLIPModel** and **CLIPTokenizer** each used to get embeddings of a query.
- The **/conversation** folder contains code for the dialog manager, including methods that deal with dialog state tracking and GPT integration.

1.2. Additional commands

change_search_type
Use for changing the search type: change_search_type <search_type> (full_text, boolean_search, text_and_attrs, emb_search).

help
Prints some commands the user can use: change_search_type, search using boolean filtering, using text and attributes or using cross-modal spaces.

2. Functions and demonstrations

2.1 Text-based searches

2.1.1 search_products_with_text_and_attributes

In this search mode, the user can input a query for each product field. The fields and query are separated by a space, therefore the queries can only be one word long. The input should be like:

```
<field1> <query1> <field2> <query2> …
<fieldn> <queryn>
```

These are the fields that can be set by the user:

```
["product_id",        "product_family",
"product_category",
"product_sub_category",
"product_gender",
"product_main_colour",
"product_second_color",
"product_brand",   "product_materials",
"product_short_description",
"product_attributes",
"product_image_path",
"product_highlights",    "outfits_ids",
"outfits_products"]
```

Example 1: Single field query - A1.1

Example 2: Multiple field query - A1.2

2.1.2. search_products_full_text

This search method uses the text that was inserted by the user to search for products. It uses a simple query string to find the right

products in the online store. The logical operations are done using regex in the query, this includes the operands: AND, OR, NOT(and variants).
For example, if we were searching for “white or black jeans” the query used would be “white|black+jeans” and if we were searching for “not white shoes” the query would be “-white+shoes”. The results would be jeans that are white, black or both in the first input and shoes that aren’t white in the second one.

Example 1: Simple query without logical operands - A2.1
Queries “white sneakers” and “black jeans” return white sneakers and black jeans results, respectively. Basically the results need to have the color white **and** to be sneakers in the first case and to have the color black **and** to be jeans in the second case.

Example 2: Query with operands - A2.2
Query “not green shirt and white” returns shirts that do **not** have the green color but do **have** white in them. This happens because the “not” is replaced with “-”(negation) and the “and" is replaced with “+”(conjunction) making that the final be “-green+shirt+white”.

2.1.3. search_products_boolean

This search method processes a query using boolean filtering. In this query there is a **must** area, where there are things that need to be in the results, a **should** area, where there are things that should be in the results, a **must_not** area, where there are things that must not be in the results, and a **filter** area, where it’s possible to filter the results to include a certain thing without affecting the relevancy score.

Example - A3.1
In this example, we want to get a men’s watch where the main color is silver, the brand should be Rolex and not Seiko.

2.2. Embeddings-based search

2.2.1. text_embeddings_search

This search method computes embeddings of a query using **transformers.CLIPModel** and returns the most similar products using kNN algorithm. The OpenSearch index contains precomputed embeddings for each product and is set up in a way that allows a kNN search. Embedding raw user input isn’t a good solution, because queries “black tshirts white stripes” and “black tshirts with white stripes” yield different results while the user desires the same items. It would be possible to filter particular words or use a library like SpaCy to choose only desired tokens, but this solution isn’t robust enough and is just a hardcoded filter - there is no real understanding of text, which is the whole point of a conversational agent. This problem is illustrated by example 1.

Embeddings inherently have some sense of negation as shown in example 2.
There was an attempt to implement negation by querying the OpenSearch index twice - once for desired query terms and once for undesired ones. Top *size_of_query* items of desired query that are not in undesired query are returned. The distinction is computed using **SpaCy**. Examples 3-4 cover that.
Example 5 shows that embeddings have some capability to understand a mistyped query but it’s not perfect.

Example 1 - Embedding raw user query A4.1
Adding a non-informative word like “with” has an impact on query results. Not only are there different results, but also accuracy is much worse as it returns items that are not tshirts or don’t have stripes.

Example 2 - Inherent negation of embeddings - A4.2
Changing “with” to “without” in the above example to get black t-shirts without white stripes.

Example 3 - Incorrect behavior of negation - A4.3
The incorrect behavior is just the fact that it still shows items that should be excluded from the search. It is expectable, because the negation algorithm is just bad. Considering that database is huge *size_of_query* would have to also be big to remove all items that fit the undesired term. This means that the system will either have low recall or will be very inefficient computationally. Also, the

SpaCy-based detection of negated terms might be imperfect which might lower the performance even further.
It is possible to exclude the undesired item by appropriately adjusting the query, but it's definitely not the desired level of interpretation of text.

Example 4 - A typo - A4.4
Using embeddings a typo can sometimes be ignored and correct results are still returned. It doesn't always work. Typo correction might (tested on only a couple of queries) have better tolerance for description of an item rather than the item type.
For example both "black belts" and "blach belts" return black belts as results.

2.2.2 image_embeddings_search

This search method computes embeddings of an image using **transformers.CLIPModel** and returns the most similar products using kNN algorithm.
This search is not working properly due to a problem with the indices on the cluster. What happens is for every image used in the search, the same 3 images are returned (3 pairs of jeans), we've tried to find the error and we found out that the path in the response isn't a valid one, to reinforce this it doesn't make sense that the cross modal search works (image+text) but this one doesn't.
Here is a example that shows what's happening:

Example: Searching for a watch similar to a Rolex Submariner:
The logs showed that this was the response:
{'product_family': 'Clothing', 'product_materials': ['Spandex/Elastane', 'Cotton'], 'product_attributes': '[{"attribute_name":"Leg Length","attribute_values":["Regular"]},{"attribute_name":"Rise","attribute_values":["Mid Rise"]}]', 'product_image_path': '**../data/ProductsFarfetch/images/images/17/01/98/64/17019864.jpg**', 'product_sub_category': 'Wide-Leg Jeans', 'product_highlights': '[indigo blue, stretch-cotton, flared, logo patch to the rear, belt loops, front button and zip fastening, classic five pockets]', 'product_gender': 'WOMEN', 'product_main_colour': 'BLUE', 'product_second_color': 'N/D', 'outfits_products': [[16174797, 16250861, 16454859, 17018995, 17019864]], 'product_short_description': 'flared-leg jeans', 'product_id': 17019864, 'product_brand': "LEVI'S", 'product_category': 'Denim', 'outfits_ids': [281302]}

As we can see the path of the product is wrong, if by any case we needed that pair of jeans, the correct path would be:
https://large.novasearch.org/farfetch_products/images/17/01/98/64/17019864.jpg

Example 1 - Response for image of a of a Rolex Submariner - A5.1

2.2.3. cross_modal_search

This search method computes embeddings of a text and image query using **transformers.CLIPModel** and returns the most similar products using kNN algorithm.
In this type of search we first process the image and the text from our query, getting the embeddings from each one of those, after that we combine them and use them as one for our search.
Just like in text embeddings search, this search supports negation too with the same strategy, but it doesn't work properly every time, for example if I make a search with the rolex from image_embeddings_search, but with the text "not gold", it may return things that are not what I really wanted.

Example 1: Searching for a watch like the image of a Rolex Submariner, but with the color gold - A6.1

Example 2: Searching for a shirt but black and white, giving it an image of a blue Billabong shirt - A6.2

Example 3: Searching for a watch with the negation "not gold", giving it an image of a watch - A6.3

2.3. Statistics from Embeddings-based search

2.3.1. Accuracy

To calculate the accuracy of the search methods we manually prompted the program with a query and counted how many results match the given query and how many don't. To have a comparable set of results between the different search methods we created similar queries for all of them.

These are the queries that were used:

1. Blue pants 2. Red pants 3. White pants 4. Green pants 5. Black pants	6. Blue shirt 7. Red shirt 8. Green shirt 9. Black Shoes 10. White Shoes

And for the image and cross-modal search, here are the images we used - **A7.1**

We only considered matches for results that matched all given parameters in the query. For example, if the query was "Black Shoes" and the result is a pair of black pants, we don't consider it as a match. For the cross modal search, we used the color as the query and one of the images(for example, use the image of blue pants, with query "black"). Here are the results we got:

	text embeddings search	image embeddings search	cross modal search
Accurate	28	6	23
Not Accurate	2	24	7
Score(%)	93,33%	20%	76,67%

3. Contextual embeddings and self-attention

3.1. Contextual embeddings

The sentences are "This soup is great, how to cook it?" and "Just mix everything You have in the fridge.". Layer 0 has no context because the context is calculated based on the attention given to each token from the previous layer and the 0th layer has no previous ones. Higher layers tend to capture more abstract interaction between words. Notice how on the lower layers words are grouped together based on their general meanings/contexts - words like "just", "everything" or "you" and "have", "fridge", "soup", and "cook" are close to each other. Higher layers capture the sentence meanings, like on layers 7 and higher "everything" and "fridge" are next to each other, rather than to "just" and "soup" like previously. Similarly "have in the fridge" embeddings are all next to each other on higher layers but not on the lower layers. In layers 8 and higher one can clearly see clusters of phrases, e.g. "how to", "mix everything", "have in the fridge" and "this soup is great" etc.

This shows how the context of the sentence is being captured by the embeddings. In the last layer differences between all token embeddings are small, because those are the final embeddings of a sentence that is coherent and makes sense. Some layers pay more attention to neighboring words, while other layers pay more attention to words further away in the sentence. All of the layers are obviously different, because token embeddings are a weighted sum of multiplication of attention from previous layers and other embedding values. See **A8.1**.

Disregarding attention to [CLS] token layer 2 mostly attends to preceding and succeeding tokens, which is noticeable on the PCA plot - token embeddings changed their relative position less than after other layers, which is indicative or positional information importance. See **A8.2**.

3.2. Positional embeddings

The query was “cheese” repeated multiple times. Embeddings of each occurrence of this word are different because of positional embeddings. There is no context to be learned as the “sentence” itself is meaningless. Plots below show embeddings after layers 2, 4, 6 and 7 respectively. Distance between embeddings of different layers is preserved, which means that indeed no context has been learned and positional embeddings are the major part of those embeddings. Mirroring along an axis preserves distances between all points and the orientation of the “shape” in principal component analysis is meaningless. See **A8.3.**

Distance between token embeddings is proportional to distance between tokens within sentence, which is illustrated on subplot of layers 2 and 2. Distance between token embeddings also grows with the number of layers between embeddings, but the change is not that big. See **A8.4**.

3.3. Self-attention

Control tokens like [SEP] and [CLS] are special and for the purpose of semantic analysis will be ignored. Their attention values are unproportionally high compared to attention values of regular tokens. It's worth noting that on the last layer [CLS] token is the one aggregating values for classification task and thus will pay attention to all of the tokens. The last dot is associated with the [SEP] token that signifies the end of a sentence and that's why its attention values are also unproportionally high. See **A8.5** and **A8.6**.

Some heads look mostly at words right next to them - this is represented by near, but not exactly, vertical lines on the following plot. This was demonstrated in the contextual *embeddings* subsection. Layer 2, which mostly looks at words next to each other, did not influence relative position of token embeddings as much as other layers. This means those heads look at the relative position of tokens in a sentence, rather than their context. All words pay some attention to themselves regardless of the residual, but it’s mostly a small value of attention. While some heads may be similar, none are identical, which allows to capture context, and thus meaning, pretty well. Tokens that are close to each other on the PCA graph usually attend to each other. For example, on layer 9, the token “fridge” attends to “mix everything you have in the fridge” and “soup” attends to “this soup is great”, “cook it” and “just mix everything”. The latter also illustrates that self-attention mechanism can handle multiple contexts, in this case a description, question and answer. Tokens like “in” attend to tokens that they refer to, e.g. “in the fridge” or “have” to “you have” (and also to the contexts, so to “mix everything”, but this has already been covered). Some tokens also have broad attention, which means they attend to most, if not all, other tokens but with smaller weight.
See **A8.7.**

3.4. Sentence embeddings

Sentence embeddings are mean pooled embeddings of all tokens. Mean pooling is used because it’s a better representation of distribution of values than min or max pooling, which will be skewed by the outliers if any are present. Window size is based on input size, which makes output sentence embedding constant size regardless of number of input tokens. Embeddings are also being normalized within a sentence for stability of the algorithm.

4. Dialog Manager

4.1. Diagram

This diagram shows the dialog logic that our program follows, from the beginning of the conversation, to the end. See **A9.1.**

4.2. Dialog Intent Detector

Using the *‘bert-dsti-ff-new.ptbert-base-uncased’* model, we can detect from a given message, what the user wants to do, from a set of possible intents. See **A9.2**.

If the model detects that the user wants a product suggestion, it takes the key-value pairs from the model output and creates a query for the search models. For example, for the key-value pairs *{'category': 'suit', 'colour':'black'}*, it sends the query *"black colour category suit"*. We also experimented with using just the values in the query construction(for the previous example, it would be just "black suit" or "suit black"), but the accuracy of the queries wasn't affected.

4.3. GPT Integration

We used OpenAi GPT-3.5 for two purposes: for generating text responses with a set intent key that is detected by the model described in the previous section and for creating/detecting a new key that was not present in the model used for the intent detector.

4.3.1. Text generation

In our program, we only have hard coded responses for queries with the 'greeting', 'goodbye', 'who made you', 'who are you', 'are you a bot' and 'who do you work for' intent. For some examples of interactions using the text generation feature see **A9.3**.

4.3.1. Detect element value from input

We can also detect when the user wants to know more information about the last returned results. However, our model does not have a key for the element in the sequence that the user is referring to. For example, if the user asks "show me black suits" and then asks "what is the brand of the second suit?". The trained model has no way of understanding which result the user is talking about.

To solve this problem, we created a prompt for GPT-3.5:

get_elem_prompt = "I am building a dialog state tracking machine, and my model has a slot_key named \'element\'. \'element\' represents the position of the element in a given sequence. For example, \'what is the brand of the third product?\' will give me a value for \'element\' that is 3. If you can't find a value for \'element\', please set it as \"unknown\". If the user is referring to more than one position, set \'element\' as \"all\".
What would be the key-value pair for this phrase:
\'{input}\'
Please return the result inside curly brackets."

This way the result is always something similar to:
{element: 2}
We then use this to retrieve the referenced element from the last result set and we return the information that the user wants.

4.4. Translation

To implement translation, we tried a few different methodologies:

- using the langdetect and the translate libraries to translate user input to english and the output to the detected language;
- using the papluca/xlm-roberta-base-language-detection and t5-base models for language detection and translation.

Ultimately, we realized that existing language detection models/libraries struggle with short inputs. Since most inputs in a chatbot application are shorter than 20 characters, the program has trouble maintaining accuracy in detecting the preferred language of the user. For example, when the user sends "hi!", the language models can think this is Somali and not English. This problem is talked about in this article.

For this reason, we decided to not implement translation, as it's very hard to know what the user's intent is in regard to the preferred language.

4.5. Persisting information about previous queries

4.5.1 Asking for additional information

If a user's query lacks information about a necessary characteristic, the agent will ask about it, but remember what the user said

previously and add those characteristics to the new query. Necessary characteristics list contains only one element - “category”. If a user would like to narrow down the search he can use the following two features (4.5.2, 4.5.3) and if he is looking for inspiration he shouldn’t be forced into specifying anything. See **A9.3** for example use.

4.5.2 Remembering previous characteristics

The agent will remember previous characteristics and add newly provided ones to already known ones. It will forget all the characteristics everytime a necessary characteristic is changed by the user (unless the agent asked for additional information about a missing one) with an exception for “category_gender_name”. Gender is persisted until the user explicitly searches for items for the other gender.
This solution keeps the conversation flow natural, which would resemble a conversation with a human. Another approach would be to make the agent ask “is this a new search” everytime, but a need for constant reassurance for the agent would destroy the flow, and thus users' experience.
Example use case for this feature is user asking for a piece of clothing and then specifying more information about desired item, e.g. querying “from nike” after looking at the results of previous query “blue hoodie” will make the agent search for “blue hoodie nike”. See **A9.4** for example use.

4.5.3 Searching for similar

The agent is able to search for similar items to “n-th” or “last” item from the latest produced result. To understand which item the user wants it uses SpaCys numerizer extension for finding ordinal numbers in text or fuzzy matching for word “last”. It’s also possible to use GPT for this task in a very similar manner to VQA, but it hasn’t proven to be noticeably better. SpaCY was chosen because it doesn’t need to make requests to external APIs, which sometimes can be slow to respond. See **A9.5** for example use.

5. Conversational agent with VQA (additional feature)

Our idea for the additional, creative feature is the possibility to combine text and image querying. Users should be able to upload a picture with multiple clothing items and specify which item they want using text. In simplest form it would be able to find for example a shirt on a picture of a fully clothed person and returning similar items.

To do this, we used the Blip model to generate a caption for the image and use GPT to get the clothing items present in that caption. Then we find a partial match with the user intent and the clothing items. If there is a match, we use cross modal-based search with the match. See **A10.1** for an example graph. See **A10.2** to understand what the initial dialog logic diagram looks like with the VQA search implemented.

We tried both conditional(with partial text input as well as the image) and unconditional image captioning(no text input) and decided to stick with conditional due to the major speed difference.

Our solution works for pictures with a single person and multiple people. If there are multiple people wearing similar clothing items, the program will return the match that corresponds to the first element from the caption. For some examples of interactions using the VQA feature see **A10.3** and **A10.4**.

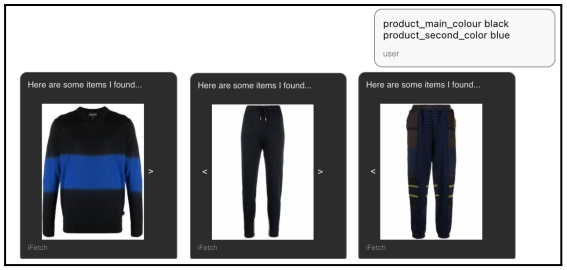
One thing to add, is that the user needs to be precise, in the sense that if the caption has “hat” and the user writes “cap”, it will not work. We tried to fix this using a pre-trained word2vec model, but none of us was able to run it, since our computers were freezing during the load process. With that we opted to just make the user be precise with his searches. This can be observed on **A10.5**.

7. Appendix

A1 - search_products_with_text_and_attributes

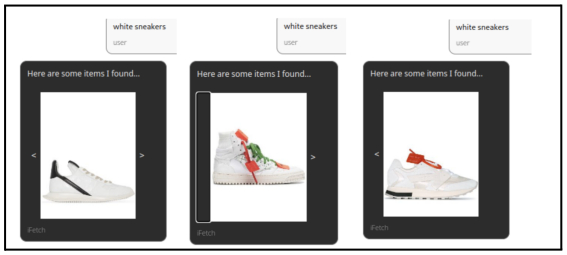
A1.1

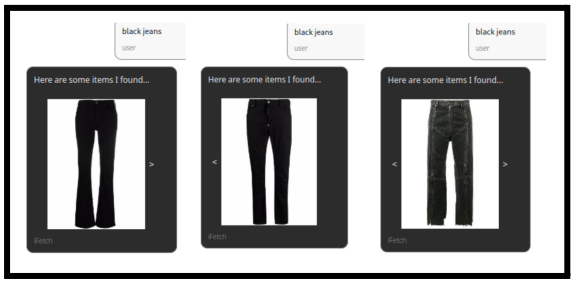
A1.2



A2 - search_products_full_text

A2.1

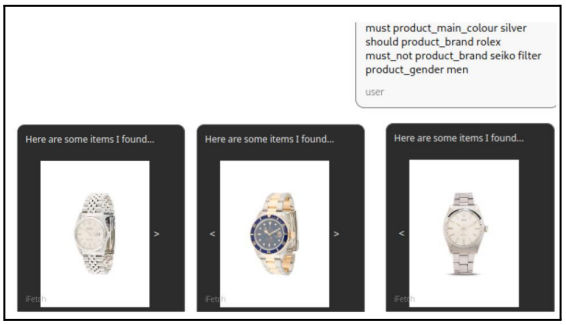




A2.2

A3 - search_products_boolean

A3.1

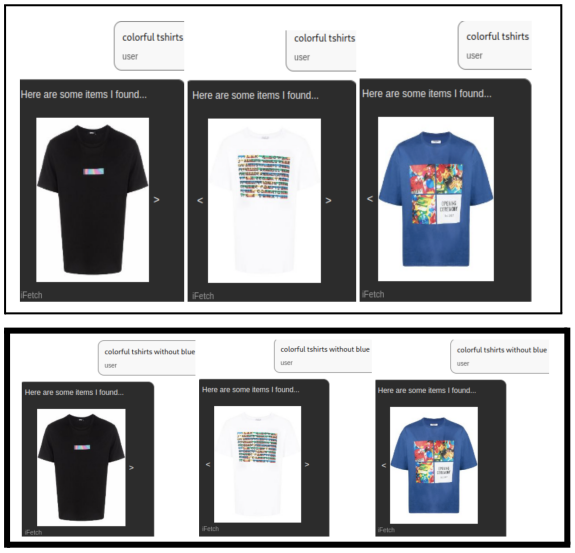


A4 - embeddings_search

A4.1

A4.2

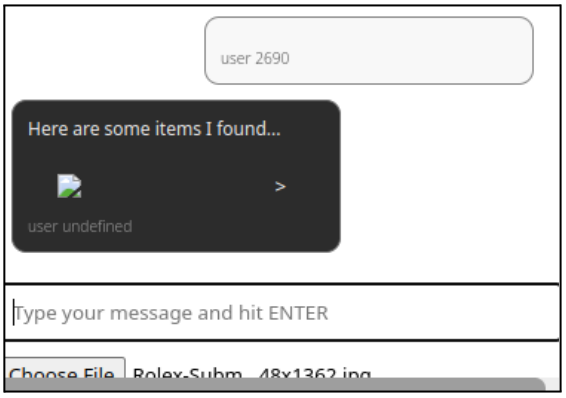
A4.3



A4.4

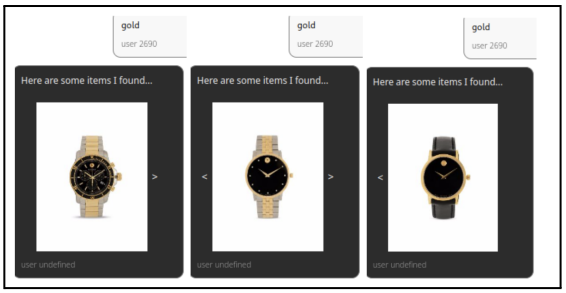
A5 - image_embeddings_search

A5.1

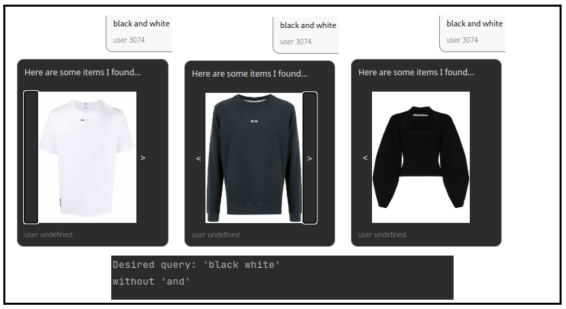


A6 - cross_modal_search

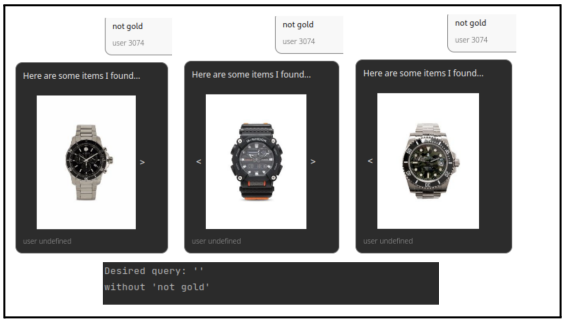
A6.1



A6.2

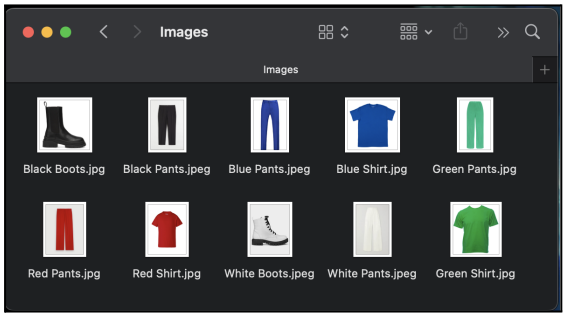


A6.3



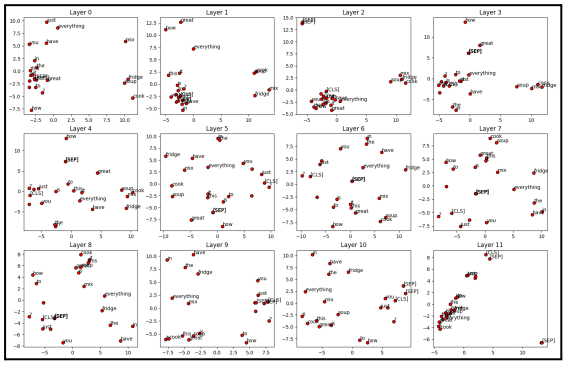
A7 - Statistics from Embeddings-based search

A7.1

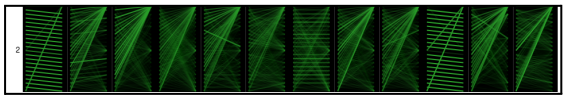


A8 - Contextual embeddings and self-attention

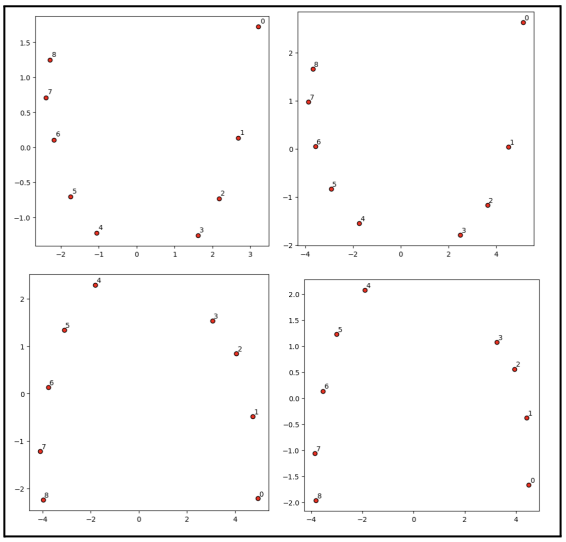
A8.1



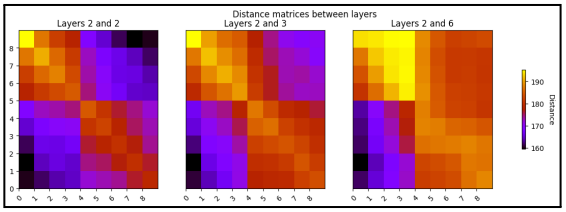
A8.2



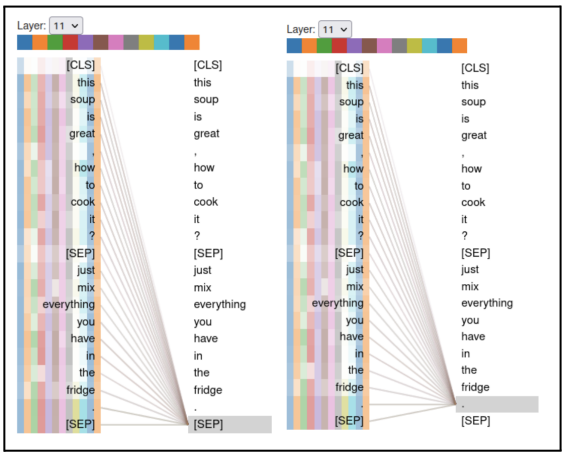
A8.3



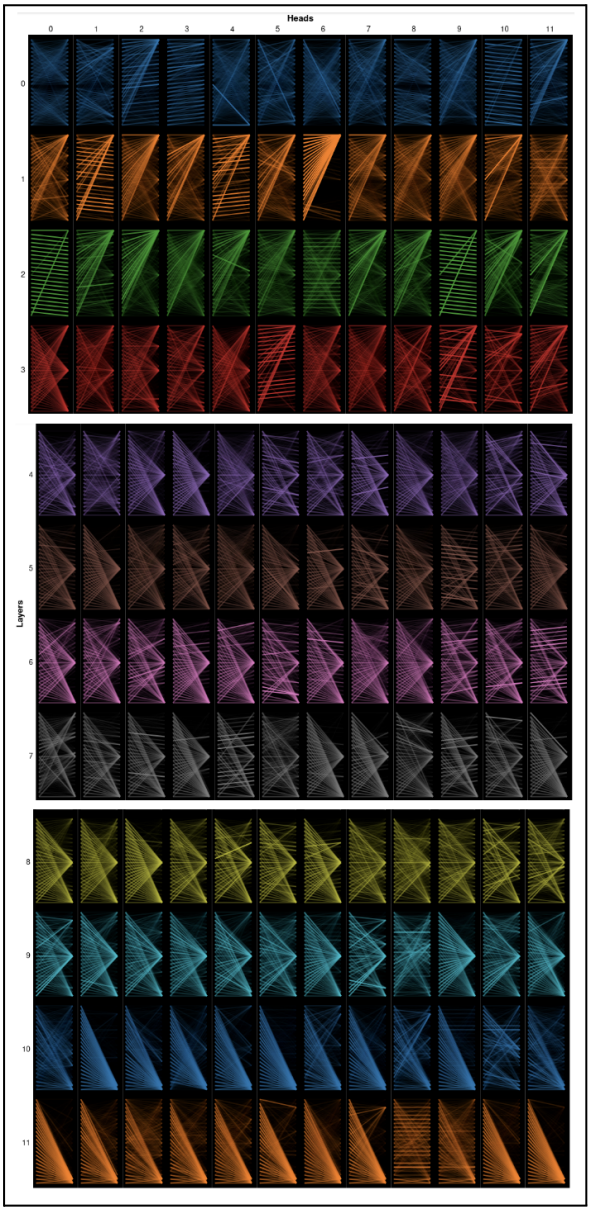
A8.4



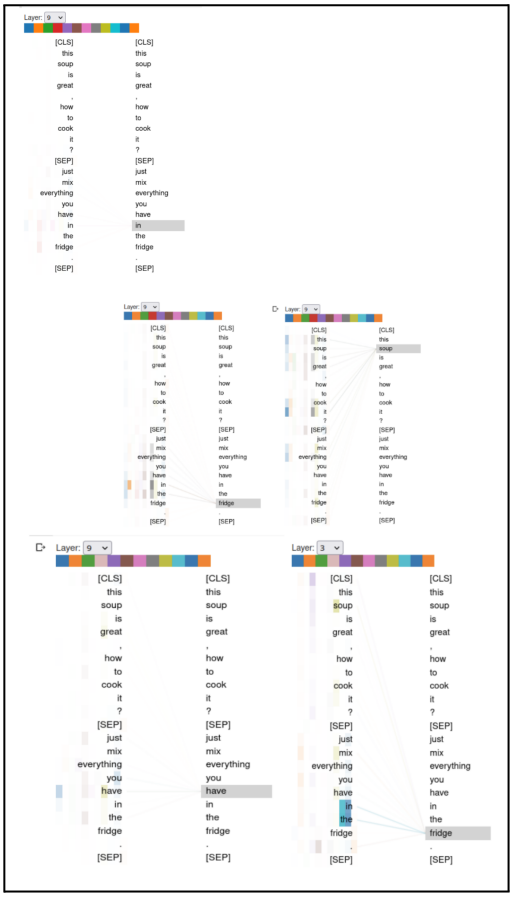
A8.5



A8.6

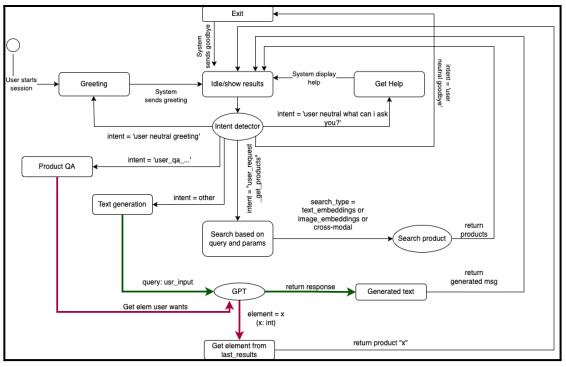


A8.7

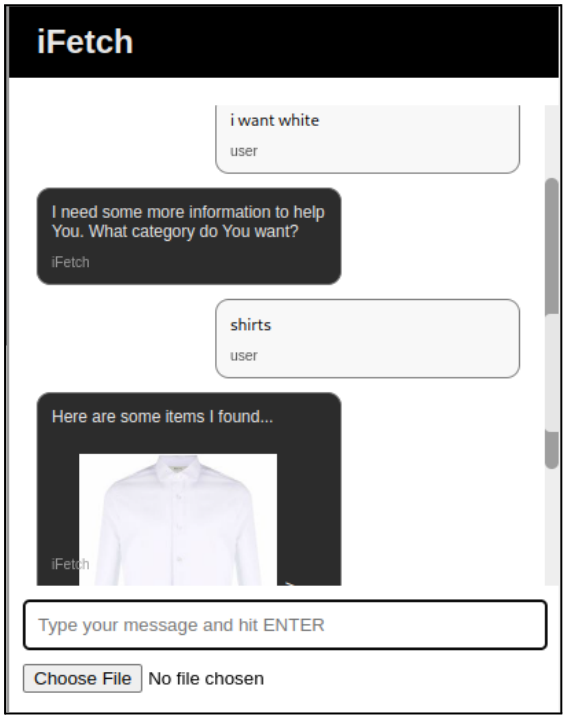


A9 - Dialog Manager

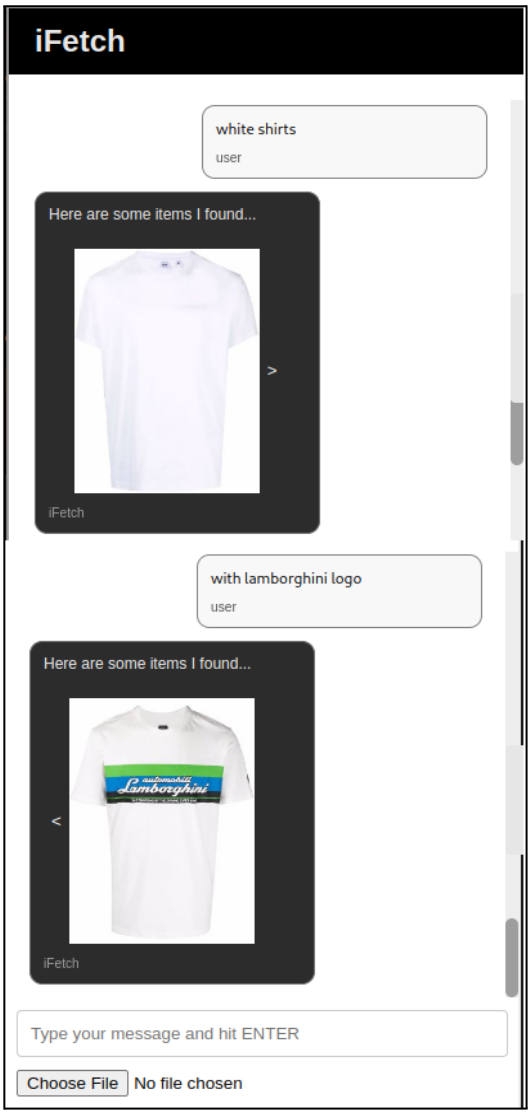
A9.1 - Dialog Manager Diagram



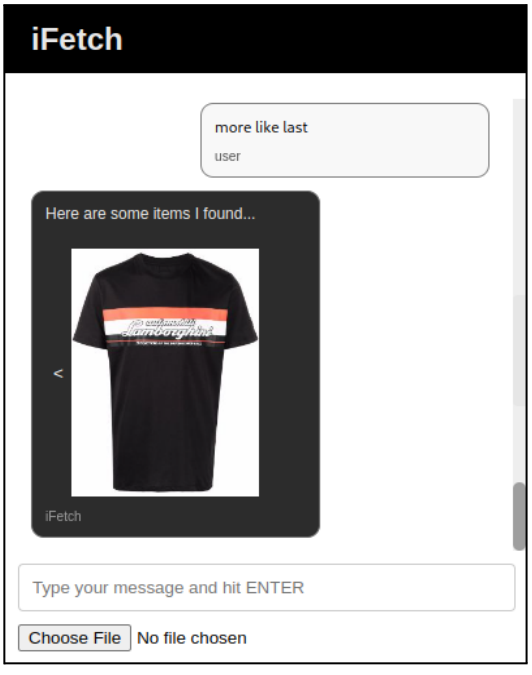
A9.3


iFetch
i want white
user
I need some more information to help You. What category do You want?
iFetch
shirts
user
Here are some items I found...
iFetch
Type your message and hit ENTER
Choose File
No file chosen

A9.4

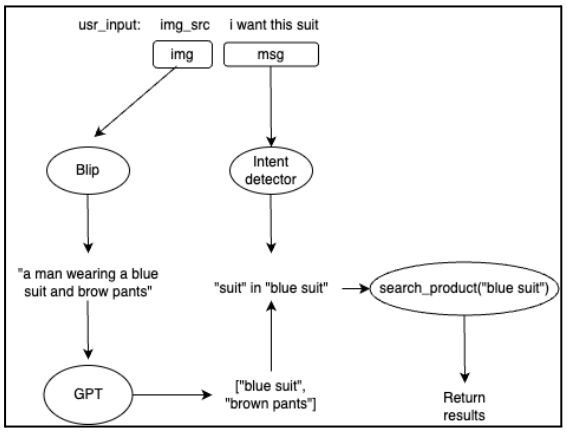

iFetch
white shirts
user
Here are some items I found...
iFetch
with lamborghini logo
user
Here are some items I found...
iFetch
Type your message and hit ENTER
Choose File
No file chosen

A9.5

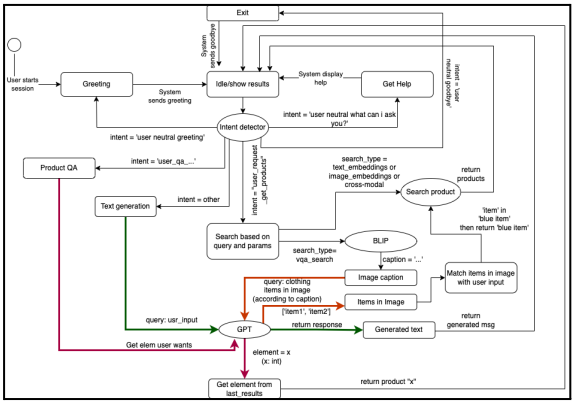

iFetch
more like last
user
Here are some items I found...
iFetch
Type your message and hit ENTER
Choose File
No file chosen

A10 - Conversational agent with VQA

A10.1



A10.2 - Dialog Manager Diagram with the additional feature implemented



A10.3

A10.4

A10.5