Conversational Task Agent

Diogo Rodrigues 56153, João Vargues 55185, and José Murta 55226

NOVA School of Science and Technology, Portugal

Abstract. This report has an introduction on the work done, an explanation about the chosen algorithms and implementation and an evaluation about the dataset description of the first and second phase for the Web Search and Data Mining course project.

1 Introduction

Conversational Task Agent have influenced the way many people engage with everyday activities and it is expected that in upcoming years, the use of the conversational paradigm will substitute the traditional search-and-browse paradigm[8].

In this report it will be discussed the implementation of a conversational assistant that helps the user to execute a task and provide the user with proper guidance throughout the task.

On the **first phase**, the main focus was on creating a searchable index of a recipe knowledge base that allowed the system to be able to solve a Question-Answer retrieval task.

In this **second phase**, the objectives were to illustrate all the steps of all recipes with images already provided using CLIP, understand the the self-attention mechanism for vision-language web related tasks and demonstrate the use of the Transformer architecture in different tasks.

2 Algorithms and Implementation

As explained in the introduction, the goal of the first phase of this project was to create a searchable index of recipe knowledge base implementing a Question-Answer search framework to answer Natural Questions. For that purpose we used the **OpenSearch framework**. Opensearch is a distributed search and analytics engine base on Apache Lucene, which organizes data into indices. Searches on the added data can be made using several types of queries with different features to be able to retrieve the desired data[2].

2.1 Mappings

To index documents using the OpenSearch framework specific settings and mappings have to be defined. In this project the documents provided are JSON files containing information about recipes. On our implementation we used 4 different types of data properties:

- to map the **recipe_id** and the **ingredients** we used the **keyword type**, since these index mappings are a string sequence of structured characters;
- to map the **title** and **description** of each recipe the **text type** was the property chosen, considering that they are string sequences of characters that represent full-text values
- to map the **title_embedding** and the **description_embedding** we used the **knn_vector type**;
- and finally, to map the **number of servings** that each recipe suggests we use the **Integer data property**.

To improve the index mappings on the properties of the text type we included two other parameters: **analyzer** and **similarity**.

The analyzer refers to the text processing method that we desire to use when running queries on that property. OpenSearch provides several built-in analyzers with each one of these using different character filters, tokenizers and token filters, and in our implementation, we decided to use the **standard analyzer**. The similarity, the second parameter, refers to the similarity module to be used. Similarity is the process of computing how similar two words, phrases or sentences are, which can be very helpful to tasks like question answering[1]. The **default similarity** module is the **BM25**. This module is an improvement over the TF_IDF algorithm, a widely used algorithm in natural language processing, because BM25 takes into account the field length and applies term frequency saturation functions, making the result more relevant to the user's query. Because of these reasons, we decided to go with this module.

2.2 K-Nearest Neighbor Vector mapping

As described above the embeddings of the title and the description use the type knn_vector so we are able to perform Approximate k-nn searches. Knn means **K-nearest neighbor and Approximate means** that for a given search, the neighbors returned are an estimate of the true k-nearest neighbors [4].

To index mappings of knn_vector type we need to detail other required properties, such as the **dimension** and the **method**. The dimension property indicates the dimensionality of the indexed vectors and the method property needs to be filled with other specific properties to the indexing method. In our case, the indexing method chosen was the **Hierarichical Navigable Small World**, or **hnsw**. This method is an Hierarchical proximity graph approach that does not require training to approximate k-NN searches. The properties specific to the indexing method are:

- **name**: name is hnsw as described above.
- **space_type**: indicates the similarity function containing the vector space used to calculate the distance between vectors, and we opt with the inner-product one.
- **engine**: the engine property indicates the approximate k-NN library to use for indexing and search and we chose the engine faiss.

– Finally the parameter **property**, is divided in two other:
  - **ef_construction**: the size of the dynamic list used during k-NN graph creation and the greater the value we choose here, the more accurate the graph and slower the indexing speed;
  - **'m'**:indicates the number of bidirectional links created for each new element, and the increasing of this value can have a large impact on memory[3].

  For a balanced result between accuracy and speed we decided to go with the value of 256 on ef_construction property and the value of 48 on the 'm' property.

The majority of the choices we made for the values of the properties described above were based on the examples and documentation provided by the tutor on the workshops.

To add the recipes on the index that we created, we iterated the json document and for each recipe we extracted the **title**, **description**, **ingredients** and **servings**. To be able to create an embedding for the title and description, we used the encode function using a Transformer encoder trained with the **MS-MARCO dataset**. After using the encode function on the title and description to create the embbedings, we added them on the index as well.

2.3 Queries description

To search and test our index we implemented multiple based searches: a **Text-based Search**, a **Boolean Text-based Search**, a **Boolean Search using a filter** and an **Embedding-based Search**. We created a new Python document to create all the search based functions. For the Text-based Search we search a result providing a query and the fields we want in the response. We then created two boolean search, one where we search with a query and given ingredients where the result must have the ingredients provided and should have the query. The other boolean we get a result using a filter for the ingridients and as before, we also should have the given query in the response. Finally we implemented a search using the embedding that we add to the index earlier. We can search using the title embedding or the description embedding.

2.4 CLIP

For the second phase of this project, the first objective that was done was the illustration of all the steps with images as mentioned in the introduction, using **Contrastive Language-Image Pre-training (CLIP)** that allowed us to compute the similarity between a given step and all provided images. CLIP is a neural network trained on a variety of pairs of type (image, text). It can be instructed in natural language to predict the best and most relevant image, given a text, or the other way around, without directly optimizing the task[9].

In order for us to implement and use CLIP functionalities to find the image that best describes each step, first we had to get all the images from recipes and steps that the provided json document had. In order to encode the images and save them, we used the method load of clip from **ViT-B/32** that returned the **model** and the **TorchVision** transform needed by the model to transform the images[9]. With the images opened, we applied the method tensor from the **torch library** that is a multi-dimensional matrix which contains elements of a single data type[10]. The values of that matrix, in our specific case, are the features of the images. The features were encoded using the model loaded previously, and normalized in order to be used in future procedures to calculate the similarity with the text features.

To choose the best image given a step information, we loaded a new model, a processor and a tokenizer from the pretrained model of CLIP and we defined three functions:

– **similarity -** calculates the similarity value between the text features and the image features;
– **bestImage -** that receives a text, encodes it into a numpy array, calculates the similarity using the function previously described and returns the index of the best image.

For visualization purposes, we also defined to other functions: **showStepsUrl** and **showStepsImage**. These functions have a similar functionality. Both receive an id of a recipe as parameter, and check if each of the recipe's instruction have an image, and if it does not, then the **bestImage** is used to calculate the image that better describes its description in order to help the user to cook the recipe step by step. The first function only shows the URL for the image, and the second one shows the actual image as output. We opt for using these two similar functions so we can choose the most suitable one for our final product, the Dialog Manager, on the next phase of this project.

Failure Analysis and Future Solution Our approach to address the problem of illustrating all the steps of all recipes with images may present some lack of accuracy on choosing the best image possible for each step of the recipe. However, a perfect match between every step and every image, would be almost impossible, since there is a high discrepancy between the number of instructions and the number of images on the dataset. In fact, there are 5425 instructions and there are only 1801 images from the instructions and the recipes. In order to improve the accuracy of our solution, it would be necessary to have more available images on the dataset.

In addition, on our implementation, every time an instruction does not have an image associated, we calculate the best image for that step, and if there is the need to retrieve the same step again, there would also be the need to calculate the best image again. Over time, this is not the most efficient approach. To improve the efficiency of our solution, we could save the image url on the appropriated attribute of the json file when the best image is calculated for a specific step. This way, we would only have to calculate the best image once for each step.

2.5 Questions phase 2: Contextual embeddings and self-attention

Contextual Embeddings Word embeddings are applied in a free environment, this is it does not have a sense of context in the sentence. In order to mitigate this problem it is used contextual embeddings. BERT calculates the embeddings of the words in a bidirectional way using both previous and next context around the word. This is particularly important for homonyms (words with different meanings but same spelling) because in word embeddings like word2vec with no context awareness this words would have the same values, BERT using 3 layers (positional, sentence and token) is capable of generating different embeddings for each meaning.

In the notebook provided as a solution for the phase 2, it is generated 12 plots to visualize the contextual word embeddings of all 12 layers and it can be observed that for each layer words with similar meanings are in each layer quite close to each other. For example, the word "cancer" is used in the both words with the same meaning and they have almost the same embeddings, also the words "throat" and "lung", both connected by the context "cancer" in the sentence, have similar embeddings and because of that are always close on each layer's plot.

Positional Embeddings The position and order of words are very important since they are the ones that define the grammar and thus the actual semantics of a sentence[11]. Positional Embeddings are introduced in order to recover the information about position, since information about "order" for example is not shown in the embeddings. This is because the "order" is not automatically considered by the model, and in Transformer, this information is not present[12].

In order to test and study positional embeddings, we used a simple BERT encoder. We provided a sequence with the word dog repeated 20 times, and then we encoded it.

Looking at the plots that were generated on the notebook that we provided for positional embeddings, we can observe that on every plot, the words dog are always grouped or very close to each other. This means that, although there can be some very small discrepancy in the visualization, the words are very similar and that makes sense since the words are all the same.

Sentence Embeddings The representation of the meaning of a sentence is important for many tasks. It allows us to understand the intention of the sentence without calculating individually the embeddings of the words.

To demonstrate our work on sentence embeddings, we utilized **Sentence-Transformers**, a python framework used for sentence, text and image embeddings. The model we chose is the **msmarco-distilbert-base-v2** which maps sentences and paragraphs to a 768 dimensional dense vector space [13]. The tokenizer is also from the **msmarco-distilbert-base-v2** transformer. We based our implementation on the examples and documentation provided by the tutor.

In order to compute the sentence embedding, firstly the sentence has to be tokenized and after that the token embeddings of the sentence have to be calculate. Then, a mean pooling function on the token embeddings calculates the average of all the tokens and finally the pooled embeddigs are normalized.

To visualize the similarities between two sentences, we defined a function that, given two sentence embeddings, calculates the cosine similarity between both. This gives a similarity score between them, and this way we are able to observe how similar two sentences are.

Self-Attention Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence [14]. In simpler words, self-attention allows each word in the sentence to look at other words to better know which word contribute for the current word.

In order to compute the self attention, it is firstly necessary to do the encoding of the desired words of one or more sentences. To do that, we used the AutoTokenizer and the AutoModel from **bert-base-uncased** model. Then, with the encode tokens, we are able to access their output that contains the **attentions** for all tokens. This return the attention weights after the attention softmax, used to compute the weighted average in the self-attention heads.

To visualize our results, we decided to present a Multi-head Self-attention heatmap matrix of the layer one of the attentions, where we are able to verify the different results for the attention calculation on the different heads of the Multi-head Attention. On the first visualization, we used the BERT Cross-Encoder to compute the two different sentences simultaneously, and while using the BERT as a Dual-Encoder with the two desired sentences encoded separately. While using the cross encoder, we can verify that there is an higher correlation between the words that can have a similar context.

Vision and Language Self-Attention Visual Bert is a powerful pre-trainable generic representation for visual and language tasks that can be tuned to various down-stream visual linguistic tasks such as Visual Question Answering [15], that is the one we are using on our provided solution as an example. In a simplistic manner Vilt directly inputs image patch projections into the multi-modal Transformer in order to achieve the fastest inference speed with the lightest VLP architecture[16].

In order to visualize the self-attention with vision and language data we first had to encode a image with a question using the Vilt processor. Then we got the outputs of the encoded image and question and visualized the Predict answer, that in our specific case of study made sense, since we had an image of rice, and the answer to our question was rice.

To visualize our results of Multi-head self-attentions, we used a heatmap matrix of the layer one of the attentions, where we were able to verify different results along the attention calculations of the different heads of the Multi-head Attention. Using the Visual Bert, we can see that the features of the image that are more useful to the final result appear more frequently with brighter colors in the different layers, while zones that do not make sense to the interpretation almost always appear in darker colors, meaning that they have no correlation with the image and the question provided.

3 Evaluation

3.1 Dataset Description

Open Search is an engine for analysis and search of data. The data is stored in indices that are collections of JSON documents. In order to initialize it, the indices contains *settings* and *mappings*. The settings give information about the index name, creation date, number of shards and replicas and the mappings describe the collections of fields for the JSON documents that will be indexed [2].

Our index settings has four shards, zero replicas of each shard and a refresh interval of minus one (meaning that the refresh is disabled).

Our choice for the set of index mappings to index the recipes' data are:

- recipe_id *keyword*
- title *text*
- title_embedding *knn_vector*
- description *text*
- description_embedding *knn_vector*
- ingredients *keyword*
- servings *integer*

Recipe_id and ingredients are of type keyword because they are structured sequences of characters, the first is an unique id for each document indexed and the other is a list with all of the ingredients for that receipt. The title and description are both of type text because they represent full-text values that describe in the most generic and simple way the content of the receipt For this type text we used the standard analyzer and the similarity module BM25 that their goal are explained in section 2. The servings is of type integer because it is a 32-bit number that describes the number of people who can enjoy that receipt. The last two, description_embedding and title_embedding, are of type knn_vector (a custom type of data to allow knn-search on OpenSearch, better explained in section 2) and they represent the embeddings of both title and description calculated with Dual-Encoders.

This mappings' choice is directly connected to the natural questions that are going to be asked to OpenSearch. This is, we chose the mappings fields accordingly to the most important information that will most probably be asked in the questions. For example, in the query "How to cook chicken" is of our main interest to search on the title, descriptions and ingredients of the receipt and all of this attributes are indexed on the mappings of the documents. Servings and ingredients are a good addition to use in boolean queries to the data where it can be specified an ingredient or the amount of servings that are needed.On the other side the steps is not a information we need to keep indexed on Open Search because there is no interest in search on that data.

References

1. Author, Dan Jurafsky., Author, James H. Martin.: Speech and Language Processing. 3rd ed.draft. Chapter 6.
2. OpenSearch Documentation - Introduction to OpenSearch https://opensearch.org/docs/latest/opensearch/index/ Last accessed 7 April 2022.
3. OpenSearch Documentation - k-NN Index, https://opensearch.org/docs/latest/search-plugins/knn/knn-index/ Last accessed 8 April 2022
4. OpenSearch Documentation - Approximate k-NN search, https://opensearch.org/docs/latest/search-plugins/knn/approximate-knn/ Last accessed 9 April 2022
5. OpenSearch Documentation - Full-text queries, https://opensearch.org/docs/latest/opensearch/query-dsl/full-text/ Last accessed 8 April 2022
6. OpenSearch Documentation - Boolean queries, https://opensearch.org/docs/latest/opensearch/query-dsl/bool/ Last accessed 9 April 2022
7. OpenSearch Documentation - Term-level queries, https://opensearch.org/docs/latest/opensearch/query-dsl/term/ Last accessed 9 April 2022
8. Chatbots: History, technology, and applications, https://www.sciencedirect.com/science/article/pii/S2666827020300062/ Last accessed 10 April 2022
9. CLIP, https://github.com/openai/CLIP Last accessed 8 May 2022
10. TORCH.TENSOR, https://pytorch.org/docs/stable/tensors.html Last accessed 8 May 2022
11. Transformer Architecture: The Positional Encoding, https://kazemnejad.com/blog/transformer_architecture _positional_encoding/ Last accessed 9 May 2022
12. Positional Embeddings, https://medium.com/nlp-trend-and-review-en/positional-embeddings-7b168da36605 Last accessed 9 May 2022
13. Sentence Transformers- msmarco-distilbert-base-v2, https://huggingface.co/sentence-transformers/msmarco-distilbert-base-v2 Last accessed 9 May 2022
14. The Transformer Attention Mechanism, https://machinelearningmastery.com/the-transformer-attention-mechanism/ Last accessed 9 May 2022
15. VL-BERT-jackroos, https://github.com/jackroos/VL-BERT Last accessed 9 May 2022
16. Probing Inter-modality: Visual Parsing with Self-Attention for Vision-Language Pre-training, https://proceedings.neurips.cc/paper/2021/file/23fa71cc32babb7b91130824466d25a5-Paper.pdf Last accessed 9 May 2022