Grant Agreement: 101016453

D5.7 – Deep Learning Toolbox – Version 2

Deliverable D5.7		
Authors and institution	INFAI, AIRC	
Date	M30	
Dissemination level		
PU	Public, fully open, e.g., web	
CO	Confidential, restricted under conditions set out in Model Grant Agreement	CO
CI	Classified, information as referred to in Commission Decision 2001/844/EC	

Document change history			
Date	Version	Authors	Description
03.05.2023	V0.1	Giulio Napolitano (INFAI)	Reproduced from Version 1
22.06.2023	V0.2	Giulio Napolitano (INFAI)	Introduction
08.07.2023	V0.3	Mirza Mohtasim (INFAI)	Main test drafted
13.07.2023	V0.4	Giulio Napolitano (INFAI)	Most text finalised
24.07.2023	V0.5	Giulio Napolitano (INFAI) and Kim Kyoungsook (AIST)	Final draft for internal review
27.07.2023	V0.6	Michael McTear (IXP)	Internal review
28.07.2023	V1.0	Rainer Wieching (USI)	Finalization

Disclaimer

e-VITA – European-Japanese Virtual Coach for Smart Ageing

e-VITA (EU PROJECT NUMBER 101016453)

Work-package 5 – Trustworthy AI, Data Analytics & NLP

D5.7 Deep Learning toolbox – Version 2

Editors: Giulio Napolitano (INFAI) and Kim Kyoungsook (AIRC)

Work-package leader: AIRC, INFAI

Executive Summary

This document describes the deep learning techniques used in the dialogue components of the e- VITA prototype. It briefly states the purpose of the methodology of deep learning and language models and presents a short overview of the technology used in the subtasks of the NLP pipeline. Since the project adopted the Rasa Open-Source Conversational AI system, much of the toolbox is already provided within the platform itself. Here we describe the updated version of the NLU system, which we have extended with advanced intent classifiers and semantic search. We also illustrate the latest addition to the system, that is the exploitation of the recent advancements in dialogue generation introduced by large language models.

Table of Contents

Lists of Figures and Tables

Acronyms and Abbreviations

Acronym/Abbreviation	Explanation
Conversational AI	Interactive system using artificial intelligence techniques
Rasa	Open-source Conversational AI Framework
KB, Knowledge base	Structured knowledge that encodes information necessary for the system's interaction operation
KG, Knowledge graph	Particular type of data representation consisting of nodes and connecting edges.
NLP	Natural language processing
NLU	Natural language understanding

1 Introduction

The Deep Learning Toolbox comprises a set of tools and techniques that can be used in the annotation, segmentation, analysis, and processing of language data. In the e-VITA project, the tools are related to the Rasa open-source conversational AI platform, which offers a wide variety of state-of-the-art tools to experiment with different pipelines and parameters.

In the previous version of this document, we explored the use of language models – specifically the T5 Text-to-Text Transfer Transformer [1] – for various natural language processing (NLP) tasks such as translation, question answering, and classification. We also delved into the concept of multitasking within a single model and the use of the RASA platform for dialogue systems. Our use of Named Entity Recognition, Entity Linking, and Relation Linking, which are crucial for understanding the finer points of a phrase and connecting text data to structured information, was also discussed.

In this revised version, we shift our focus to the Natural Language Understanding (NLU) component, which is integral to the processing and interpretation of human language. We aim to provide a comprehensive understanding of the NLU component and its critical role in enhancing the capabilities of the e-VITA dialogue system.

A chatbot, or a conversational agent, should excel in two critical functions: first, it needs to comprehend what the user communicates, and second, it needs to respond suitably. The initial process involves Natural Language Understanding (NLU), while the latter involves Response Generation. Here we provide an overview of our research and implementations, focussing on enhancing the NLU capabilities of the system, and justifying the enhancements we have made to the RASA NLU component. We also share experimental outcomes, in which our NLU system's performance is compared with several other top-performing NLUs.

Notably, our tools now incorporate OpenAI modules into the dialogue manager. We have leveraged OpenAI language models to respond to queries that do not align with any predefined utterances within the dialogue system. These models have been successfully employed particularly where the conventional dialogue manager falls short, due to an insufficiency in the structured narratives. We utilize custom-selected content as a resource, enabling the dialogue manager to address queries that only pertain to the source material. In doing so, we assist the dialogue manager in adopting a form of guided text generation.

2 RASA as a Toolbox for Dialogue System

The default NLU pipeline in Rasa, which was detailed in the previous version of this Deliverable, is depicted in Figure 1. It consists of four basic components: 1) Tokenizers, 2) Featurizers, 3) Intent Classifiers, and 4) Entity Extractors.

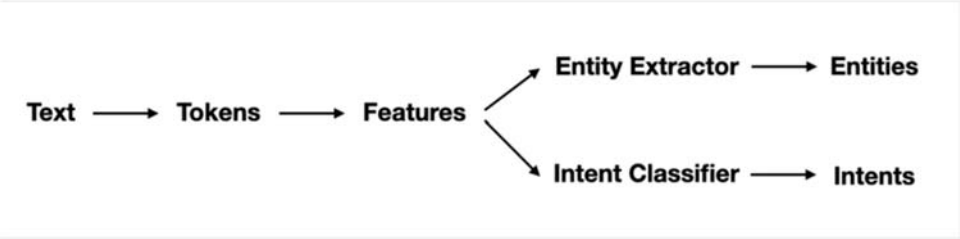

Figure 1 – Basic NLU pipeline for RASA

That can be considered as the "base" RASA system. In our project we need to handle a high number of intents across several domains and, in order to classify the intents well, we have built custom components which are directly used by the dialogue manager. Our new pipeline of the RASA system is depicted in Figure 2.

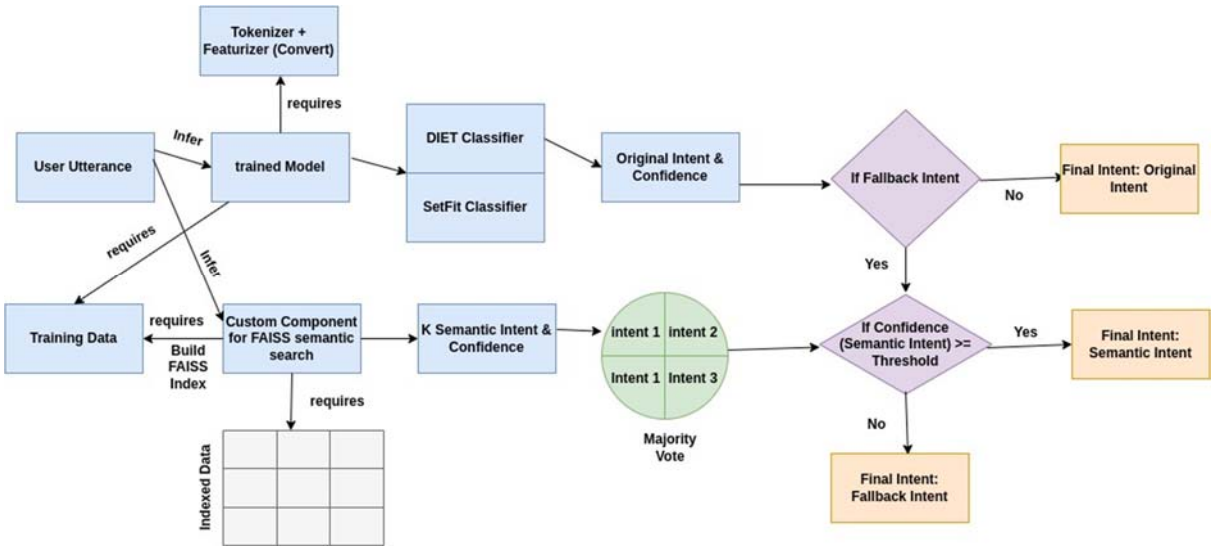

Figure 2 – Our RASA NLU Pipeline

2.1 Language Handler

Our chatbot is engineered to accommodate multiple languages, enabling users to interact with it in their language of choice. Our dialogue system incorporates the Rasa NLU and Core components, alongside the DeepL[1] language handler for machine translation. Furthermore, we performed training and evaluation procedures for the system, which include the use of custom datasets in various languages. Our findings validate the efficacy of our approach in crafting a multilingual chatbot capable

[1] http://www.deepl.com/

of precisely understanding and responding to user inputs across multiple languages. Figure 3 depicts how our system is currently using the translation layer in the language handler.

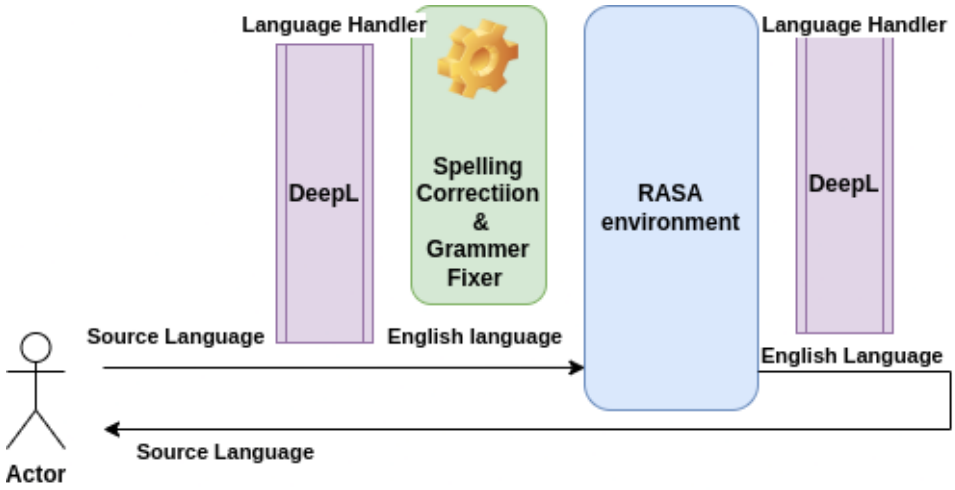

Figure 3 – Our RASA Language Handler component

2.2 Tokenizers

Tokenizers take the text input and segment it into suitable chunks. In natural dialogue processing the suitable chunks are usually words, and the output is a list of words. The Tokenizer also provides separate tokens for punctuation marks. The default tokenizer for English is the WhiteSpaceTokenizer[2] but for languages other than English it is possible to use other tokenizers.

The advantages of using a white space tokenizer are manifold, including high-speed processing and efficiency, making it an ideal choice for real-time chat applications. Additionally, its ability to be language-independent enables it to be employed effectively across an array of languages.

2.3 Featurizer

In the latest version of our pipeline, we make use of the ConveRT featurizer [2], a pretrained feature extraction model created by the RASA team. This model is uniquely crafted to draw out relevant features from conversational text and translate them into dense vector formats, which are then leveraged by Rasa's machine learning algorithms. We have selected this featurizer because of its capability to increase the precision of NLU by offering a detailed portrayal of the conversational context. The ConveRT featurizer implements a transformer-based structure to process the input text and generate dense embeddings, which are subsequently used in the training of the NLU model. Moreover, the ConveRT featurizer excels in situations with few training samples per intent, thus further increasing its suitability for our application.

2.4 Intent Classifiers

2.4.1 DIET Classifier

The crucial part of dialogue modelling is to correctly classify the user's intent. For intent classification, Rasa provides its own DIET model (Dual Intent and Entity Transformer) which handles both intent classification and entity extraction. This was the classifier we used in the previous version of the system.

[2] https://rasa.com/docs/rasa/components/#whitespacetokenizer/

2.4.2 SetFit Classifier

Along with the DIET classifier, we have now adopted the SetFit classifier and FAISS Semantic search with majority voting. As a crucial element, the SetFit Classifier is integrated into our NLU pipeline. It adds value to the RASA NLU framework by facilitating rule-based detection of specific phrases or patterns from the user input. This feature is particularly beneficial in situations where the DIET classifier might find it challenging to accurately pinpoint and classify the user utterance.

The significance of the SetFit classifier becomes evident when dealing with unique situations for which the DIET classifier hasn't been specifically trained. For example, if a user asks a specific question that falls outside the training data scope of the DIET classifier, the SetFit classifier is still capable of identifying the question and providing a suitable response. The SetFit classifier is configured and utilized in a straightforward manner, enabling developers to quickly incorporate new rules for pattern recognition in user input without the need for extensive training data or deep machine learning expertise.

Moreover, by incorporating the SetFit classifier into the RASA pipeline in conjunction with the DIET classifier, the overall precision of the chatbot is enhanced. This combination harnesses the strengths of both classifiers, leading to more accurate and dependable responses to user input. For each intent, we acquire the sample utterance and create a dataframe to feed to the SetFit classifier. The algorithm in Figure 4 illustrates how we have incorporated the SetFit classifier into RASA as a custom component.

Algorithm 1 SetFitClassifier

```
Require:
    training_data:  TrainingData
Ensure:
    resources:  Resource
function TRAIN(training_data)
    labels ← get labels from training_data
    texts ← get texts from training_data
    intent_df_train ← create dataframe from texts and
                      labels
    train_ds ← create dataset from intent_df_train
    model ← load pretrained SetFit model
    trainer ← initialize SetFitTrainer with model,
train_ds, and other configuration parameters
    trainer.train(max_length=max_length)
    persist trained model
    return resources
function PERSIST
    model_dir ← get model directory from resources
    path ← join (model_dir and 'modelname')
    save pretrained SetFit model to path
function PROCESS(messages)
    for each message in messages do
      text ← get text from message
      label ← setfit_pretrained([text])
      confidence ← max(
                      setfit_pretrained.predict_proba([text]))
      rasa_intent ← create intent object with label and
                    confidence
      set rasa_intent in message as output intent
    return messages
function LOAD(config, model_storage, resource,
              execution_context, ...)
    component ← create instance of SetFitClassifier
                with config, model_storage, and
                resource
    model_dir ← get model directory from resources
    path ← join model_dir and 'setfit'
    component.setfit_pretrained ← load pretrained
                                  SetFit model from path
    return component
```

Figure 4 – Integration of the SetFit classifier

2.4.3 FAISS sematic search

For cases in which the SetFit classifier's confidence in classifying an intent is not sufficient, we have the FAISS[3] semantic search. FAISS, or Facebook AI Similarity Search, is an open-source platform developed by Facebook AI Research, intended to simplify the search and clustering of vectors with high dimensions. This library facilitates FAISS semantic search, allowing users to seek out vectors similar in meaning to a specified query rather than depending solely on precise value correlations. By harnessing deep learning methods, including neural networks, FAISS achieves majority voting and is especially adept at managing vast datasets comprising millions or billions of vectors. This is made possible by leveraging index structures and endorsing approximate nearest neighbour searches, ensuring efficient and successful similarity retrieval. By using FAISS on the training data, we were able to generate indexed data. For this purpose, we have developed a custom component within the RASA framework. The indexed data is then employed for contrasting against incoming user expressions. It presents us with the K nearest neighbour intents according to similarity between the user expression and the existing labelled instances in the training dataset. A majority voting is then conducted to decide on the final intent.

The algorithms in Figure 5 illustrate this procedure. The algorithm introduces the Semantic Intent Classifier, which is intended for natural language understanding (NLU) tasks. It uses a DenseFeaturizer component and the FAISS index to perform efficient nearest neighbour searches.

Algorithm 2 Do Majority Vote

```
procedure DO_MAJORITY_VOTE(intents, scores)
    intents ← array(intents)
    scores ← array(scores)
    unique_intents, intent_counts ← unique(intents,
return_counts=True)
    counts_zero ← intent_counts[0]
    filtered_intent_idx ← [0]
    for i = 1 to len(intent_counts) − 1 do
        if intent_counts[i] < counts_zero then
            break
        append i to filtered_intent_idx
    combined_scores ← []
    for i in filtered_intent_idx do
        filtered_intent ← unique_intents[i]
        filtered_score ← mean(scores[intents =
filtered_intent])
        append filtered_score to combined_scores
    conf_max_idx ← argmax(combined_scores)
    intent ← unique_intents[conf_max_idx]
    score ← combined_scores[conf_max_idx]
    return intent, score
```

[3] https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

Algorithm 3 SemanticIntentClassifier

```
Require:
    training_data: TrainingData
Ensure:
    resource: Resource
function TRAIN(training_data)
    hf_dataset ← prepare_index(training_data)
    persist()
    return resource
function PERSIST
    model_dir ← get model directory from resource
    path ← (join model_dir and 'hf_dataset_index.hf')
    save hf_dataset to disk at path
function PREPARE_INDEX(training_data)
    labels ← get intent labels from training_data
    training_examples ← filter out examples without
                        text features from training_data
    X ← dense features of training examples
    data_dict ← {'intent': [], 'embeddings': []}
    for i in range(len(labels)) do
        append labels[i] to data_dict['intent']
        X_norm ← X[i] normalized by its norm
        append X_norm to data_dict['embeddings']
    data_frame ← create dataframe from data_dict
    hf_dataset ← create dataset from data_frame
    return hf_dataset
function PROCESS(messages)
    for each message in messages do
        current_intent ← get current intent from message
        if current_intent['name'] is "nlu_fallback" then
            if hf_dataset exists & message not empty then
                message_features ← dense features of
                                   message
                message_features ← normalize
                                   message_features by
                                   its norm
                scores, samples ← find K nearest examples
                                  in hf_dataset to message_features
                if K = 1 then
                    intent, score ← intent and score of the
                                    single nearest example
                else
                    intent, score ← do majority vote among
                                    intents and scores
                rasa_intent ← {text['name']: intent,
                               text['confidence']: score}
                if rasa_intent['confidence'] > threshold then
                    set rasa_intent as output intent in
                        message
    return messages
function LOAD(config, model_storage, resource,
execution_context, ...)
    component ← create instance of
                SemanticIntentClassifier with config,
                model_storage, and resource
    model_dir ← get model directory from resource
    path ← join model_dir and 'hf_dataset_index.hf'
    component.hf_dataset ← load dataset from path
    add FAISS index to component.hf_dataset for column
                'embeddings'
    return component
```

Figure 5 – Majority voting scheme used by FAISS, and its integration

The algorithm includes several essential functions. The *train* function accepts training data and forms an indexed high-feature (HF) dataset. This indexed dataset is then conserved for future use. The *process* function processes incoming messages and conducts intent classification based on the trained model. It fetches the text message, extracts dense features, and uses the FAISS index to locate the nearest instances. The algorithm utilizes a majority vote approach to determine the final intent and confidence score. If the confidence score exceeds a certain threshold, the intent is allocated to the message.

The algorithm additionally incorporates functions for maintaining and loading the trained model. The *persist* function commits the high-feature (HF) dataset index to disk storage, whereas the *load* function retrieves the model and initializes the FAISS index.

In summary, the Semantic Intent Classifier algorithm offers an effective and precise methodology for intent classification in NLU tasks. It capitalizes on dense featurization techniques and employs FAISS indexing to facilitate quick nearest neighbour search for intent correlation.

2.5 NLU Pipeline

We have significantly overhauled the pipeline of the previous prototype, to accommodate new components such as FAISS and majority voting. Alongside these alterations, we undertook a comprehensive system upgrade, transitioning our dialogue manager from RASA 2.X to 3.X. This substantial modification required us to adjust our system in accordance with the latest advancements in RASA.

```
pipeline:
- name: custom_components.language.LanguageHandler
- name: WhitespaceTokenizer
- name: "ConveRTFeaturizer"
# Remote URL/Local directory of model files(Required)
  model_url: "./pretrained_featurizers/convert_tf2"
- name: RegexFeaturizer
- name: RegexEntityExtractor
  use_lookup_tables: True
  use_regexes: True
- name: LexicalSyntacticFeaturizer
- name: custom_components.sentiment_analyser_hf.SentimentAnalyzerHF
  model: 'j-hartmann/emotion-english-distilroberta-base'
- name: DIETClassifier
  epochs: 100
  use_masked_language_model: True
- name: EntitySynonymMapper
- name: ResponseSelector
  epochs: 100
  retrieval_intent: chitchat
- name: FallbackClassifier
  threshold: 0.8
  ambiguity_threshold: 0.1
- name: custom_components.semantic_intent_classifier.SemanticIntentClassifier
  K: 5
  threshold: 0.4
```

In the DIET classifier we are using masked language modelling (MLM)[4] in order to better classify the intents. The most important components newly integrated include the ConveRT featurizer and the semantic intent classifier by FAISS. We've also updated the majority voting criteria, which is a hyperparameter. At present, we are using nearest neighbour for majority voting with K set to 5, and a confidence threshold of 0.4 for semantic Intent classification. The SetFit classifier is still under the evaluation phase. We plan to integrate them after complete testing in the evaluation. One new component that we have optionally added is a sentiment analyser which uses a Hugging Face library by

[4] https://huggingface.co/docs/transformers/main/tasks/masked_language_modeling

Jochen Hartmann named "Emotion English DistilRoBERTa-base. [5] This has been integrated experimentally to detect emotion from input utterances and can be used in the real dialogues.

The policy has also been updated due to the recent development of RASA technology. Many of the items are now simplified in the RASA configuration.

```
policies:
  - name: MemoizationPolicy
  - name: TEDPolicy
    max_history: 10
    epochs: 100
  - name: RulePolicy
```

[5] https://huggingface.co/j-hartmann/emotion-english-distilroberta-base

3 Question Answering over Wikipedia

3.1 Prototype 1

In the Version 1 of this Deliverable, we already described our employment of Haystack,[6] an open-source framework for building intelligent search systems over large document collections. In summary, we built an extractive question answering (QA) system using Haystack, where the goal is to return a text phrase from a passage within one or more Wikipedia articles in response to a natural language question.

The data used for the system is obtained from Wikipedia dumps, converted to a lightweight JSON format with just the article titles and plain text. The indexing process involves preprocessing each Wikipedia article. This includes cleansing and normalization tasks such as removing white spaces and splitting articles into smaller pieces to optimize retrieval. During the indexing phase, the system computes embeddings for each text using one of the language models provided by Haystack. These embeddings are stored in a vector database called Milvus, while the corresponding documents are stored in SQLite.

For search functionality, the system utilizes a pipeline. In this case, an extractive QA mechanism is used, which involves searching through a large collection of documents to find a span of text that answers a question. The pipeline combines a Retriever and a Reader component. The Retriever searches the database and retrieves the most relevant documents, while the Reader selects a text span from those documents as the answer to the query.

The output of the pipeline is a Python dictionary that contains a list of Answer objects stored under the "answers" key. These Answer objects provide additional information, such as the context from which the answer was extracted and the model's confidence in the accuracy of the answer.

The main components and core concepts of the general Haystack setup are depicted in Figure 6.

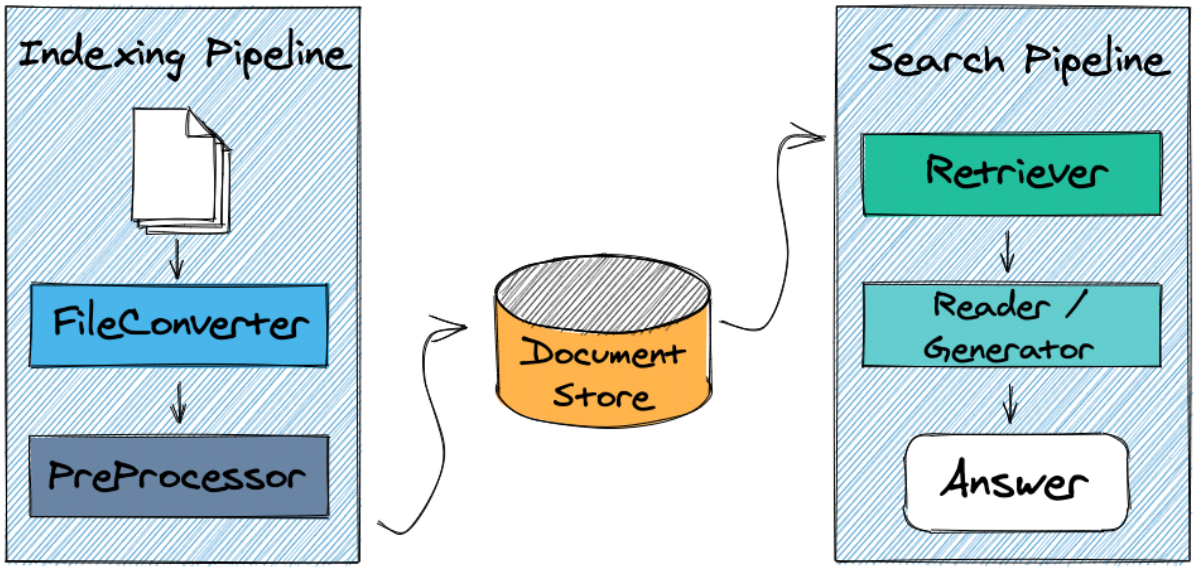

Figure 6 – Haystack architecture reference

[6] https://github.com/deepset-ai/haystack

3.2 Prototype 2

3.2.1 Modification to the module

In the prototype 2, a complete retraining has been performed with the updated Wikipedia corpus. Previously, in the prototype 1, the system providing answers from the Haystack API took full questions such as “Ask wiki about <who is the president of USA>”, where the text within “<…>” represents the question as a query. However, the system failed to answer simple questions, e.g., “ask wiki about football”. Thus, we have updated our system to use media wiki[7] API endpoint on such queries. Additionally, previously no context was provided by the system (e.g. from which source the questions were answered): now the system is updated to provide the context for the answer as well.

3.2.2 Summary of the Deployment Components

The following components and language models are still being used, unmodified from the previous version:

Haystack framework:
    v1.1.0
Language Models:
    facebook/dpr-question_encoder-single-nq-base
    facebook/dpr-ctx_encoder-single-nq-base
    deepset/roberta-base-squad2
Document store:
    Milvus v1.1.1 for storage and retrieval of the passage embedding vectors
    SQLite for storage of the passages
PreProcessor:
    clean_empty_lines=True,
    clean_whitespace=True,
    clean_header_footer=True,
    split_by="word",
    split_length=100,
    split_respect_sentence_boundary=True,
    split_overlap=0,
Retriever:
    DensePassageRetriever
        query_embedding_model="facebook/dpr-question_encoder-single-nq-base"
        passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base"
Reader:
    FARMReader
        model="deepset/roberta-base-squad2"
        use_gpu=True

[7] https://www.mediawiki.org/wiki/MediaWiki

4 Dialogues supported by GPT-3

The e-VITA project could not ignore the recent and fast-moving advancements in the state of the art in conversational AI, driven by Large Language Models (LLMs). For this, we added the use of the OpenAI API, combined with several documents generated by the Content Group. These documents form the context in which the API-accessed service frames its responses, eliminating the risk of so called "hallucinations", that is the production by the system of responses which are sometimes out of scope or untrue. This is due to the extremely large and unrestricted data set on which the language model has been trained.

The use of the OpenAI API is illustrated in Figure 7.

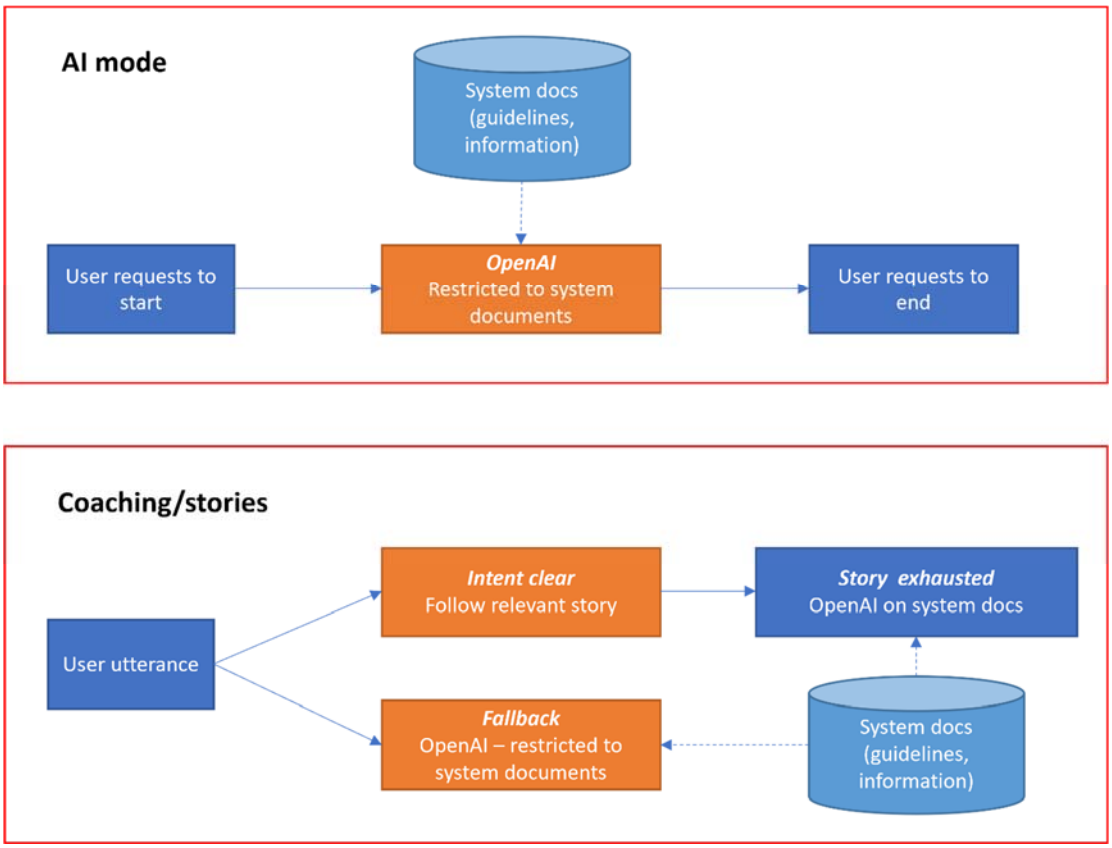

Figure 7 – Diagram of OpenAI use in the e-VITA dialogue system

In the AI modality, the OpenAI API is used to provide a human-like experience in casual dialogues about the domains covered by the system, without following a scripted dialogue.

In the coaching modality, the OpenAI API is only invoked when the user's utterance is not clear in its intent or the user follows up on a dialogue which has reached its conclusion in our main system. In this modality, the OpenAI API will again only respond on the basis of the documentation we provide, ensuring that the answers are in scope and accurate.

We are using langchain[8] and the FAISS vector store to encode the text from the documents providing the context. This is performed by the function *actionDocumentStore*, with which we also encode the history of the user dialogue with the system, see Figure 8. First we process all the documents in a

[8] https://langchain.com/

specific folder, extract information from them and create chunks of information (this is done to reduce the token size, as there is a limit for tokens for QA tasks), see

```
class ActionDocumentStore(Action):
    def name(self) -> Text:
        return "action_document_store"

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict[Text, Any]]:
        language_name = tracker.get_slot("language")
        if (Globals.openAI_fallback == False) & (
            Globals.bert_fallback == False
        ):
            utter = "I couldn't understand your query, would you please rephrase it?"
        else:
            if Globals.openAI_fallback:
                chat_history = ""
                for event in tracker.events:
                    if event.get("event") == "bot":
                        chat_history += f"ChatGPT: {event.get('text')}\n"
                    elif event.get("event") == "user":
                        chat_history += f"User: {event.get('text')}\n"
                query = tracker.latest_message["text"]
                utter = provide_fallback_dialogue(
                    query, chat_history, llm="openAI"
                )
```

Figure 8 – Using the chat history to provide context to the OpenAI API call

The *openAI_setup()* function (Figure 9) is an integral part of the project, which primarily focuses on setting up OpenAI embeddings. It is designed to automate the process of initializing and configuring essential components required to use OpenAI's language models, specifically the Ada text embeddings.

The function begins by configuring the OpenAI API key, which is crucial for enabling the interaction with OpenAI services. The API key should be kept confidential to prevent unauthorized access. In this function, the key is stored as an environment variable for security purposes.

Next, the function initiates the Ada text embeddings from OpenAI. These embeddings serve as the foundation for language models, converting text into numerical representations that can be processed by machine learning algorithms.

The function then proceeds to establish the FAISS document store. If the FAISS document store does not already exist at the designated path, it processes all PDF documents at the specified document store path and creates a new FAISS document store using the processed texts and the initialized Ada embeddings. If the FAISS document store already exists, it loads the document store from the local path.

With the FAISS document store in place, the function establishes a retriever for information retrieval from the document store. It also sets up a conversation buffer memory that tracks the chat history.

Then, the function sets up a *ConversationalRetrievalChain* from OpenAI's GPT-3.5-turbo model. This retrieval chain uses the configured retriever, memory, and specific prompts to formulate responses to user questions. These prompts include a condensation prompt to rephrase follow-up questions into standalone questions and a QA prompt to answer user queries using the provided context.

Finally, the function returns a *fallback_qa* object. This object serves as a safety net for the dialogue system, allowing it to answer queries even when there is no perfect match in the system's database. This object uses the *ConversationalRetrievalChain* and can handle queries using the Langchain context in OpenAI, thereby enhancing the versatility and robustness of the dialogue system.

```
def openAI_setup():
    ---
    sets up openAI key and other required stuffs for obtaining openAI embeddings
    @param returns: fallback_qa object where we can do qa with langchain with context for openAI
    ---
    os.environ[
        "OPENAI_API_KEY"
    ] = "[redacted]"
    faiss_docstore_path = "data/document_store/faiss_index"
    docstore_path = "data/document_store"

    # # Create a completion
    # llm = OpenAI()
    # initialize the embeddibgs using openAI ada text embedding library
    embeddings = OpenAIEmbeddings()

    # initialize the FAISS document store using the preprocessed text and initialized embeddings
    if not os.path.exists(faiss_docstore_path):
        texts = process_all_pdfs(docstore_path, preprocess_langchain=True)
        docsearch = FAISS.from_texts(texts, embeddings)
    else:
        docsearch = FAISS.load_local(faiss_docstore_path, embeddings)

    retriever = docsearch.as_retriever()
    memory = ConversationBufferMemory(
        memory_key="chat_history", return_messages=True
    )
```

Figure 9 – Setting up OpenAI embeddings

We have designed the prompt to utilize documents from the document store for pinpointing specific answers, taking into account a few shot memory as well. The prompt engineering is such that it produces answers within 3-4 sentences to ensure succinctness. Moreover, to avoid generating fictitious or 'hallucinated' responses, we've explicitly instructed it not to fabricate any answers. Details are in Figure 10.

```
CONDENSE_PROMPT = """Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question.

Chat History:
{chat_history}
Follow Up Input: {question}
Standalone question:"""
condense_prompt = PromptTemplate(
    input_variables=["chat_history", "question"], template=CONDENSE_PROMPT
)
QA_PROMPT_DOCUMENT_CHAT = """You are a helpful AI assistant. Use the following pieces of context to answer the question at the end.
Make sure the answer is between 2-3 sentences.
If the question is not related to the context, just say Sorry for this question my AI has no answer.
If you don't know the answer, just say Sorry for this question my AI has no answer . DO NOT try to make up an answer.

{context}

User: {question}
System: """
qa_prompt = PromptTemplate(
    input_variables=["context", "question"],
    template=QA_PROMPT_DOCUMENT_CHAT,
)
fallback_qa = ConversationalRetrievalChain.from_llm(
    OpenAI(temperature=0, model_name="gpt-3.5-turbo-16k"),
    retriever=retriever,
    memory=memory,
    condense_question_prompt=condense_prompt,
    combine_docs_chain_kwargs={"prompt": qa_prompt},
)

return fallback_qa
```

Figure 10 – OpenAI prompt engineering details

The *provide_fallback_dialogue()* function, in Figure 11, is designed to generate a fallback dialogue for a given query and chat history. This function serves a vital role in enhancing the dialogue system's resilience, ensuring that it can provide a response even when an exact match for a user's query isn't found within the existing dialogue history or document database.

This function supports two language models for generating fallback dialogues: OpenAI and BERT. The model to be used is determined by the *llm* parameter. The function starts by setting a default response to be used if the model isn't properly initialized.

When OpenAI is specified as the language model, the function uses a globally defined OpenAI model (*Globals.fallback_qa*) to generate the fallback dialogue. It passes the user's query and chat history as parameters to this model and then extracts the answer from the generated result.

Alternatively, when BERT is selected as the language model, the function generates fallback dialogues for each chunk of data available in *Globals.chunks*. It decodes each chunk, combines it with the chat history, and feeds this combined text into the BERT model (*Globals.model_BERT*) to generate a response. Each response is then appended to a list of responses. The function ultimately combines all the responses into one result.

Finally, the function returns the generated fallback dialogue. This ensures that the system is capable of providing a response under all circumstances, enhancing the user experience.

```python
def provide_fallback_dialogue(query, chat_history, llm="openAI"):
    # Default response if the model is not initialized properly
    result = "fallback model is not initialized properly"

    if llm == "openAI":  # Using OpenAI for fallback dialogue
        # Generate fallback dialogue using the global OpenAI model
        result = Globals.fallback_qa({"question": query, "chat_history":
            chat_history})
        # Extract the answer from the generated result
        result = result["answer"]

    elif llm == "bert":  # Using BERT for fallback dialogue
        responses = []
        for chunk in Globals.chunks:
            # Decode the chunk
            chunk_text = Globals.token_BERT.decode(chunk,
                clean_up_tokenization_spaces=True)
            # Combine the chunk text and the conversation
            combined_text = "\n".join([f"{role}: {text}" for role, text in
                chat_history] + [chunk_text])
            # Use the question-answering pipeline
            query = "Question: " + query
            response = Globals.model_BERT(question=query, context=combined_text
                )
            # Append the response to the list of responses
            responses.append(response["answer"])
```

Figure 11 – Fallback function to handle unclear or unmatched queries

We should note that we are using the initialized *fallback_qa* for question answering for the document store.

```python
def provide_dialogue_chatgpt(chat_history, user_input):
    """
    This function generates a response using OpenAI's ChatGPT model for a given user input and chat history
        .
    Parameters:
    chat_history (str): The history of the chat conversation. It is a string where each turn in the
        conversation
                        is separated by a newline.
    user_input (str): The user's current input or question that needs a response.
    Returns:
    str: The generated response from the ChatGPT model.
    Example:
    chat_history = "User: Hello\nChatGPT: Hi! How can I assist you today?"
    user_input = "Tell me about OpenAI."
    response = provide_dialogue_chatgpt(chat_history, user_input)
    print(response)  # "OpenAI is an artificial intelligence research lab..."
    """
    openai.api_key = ""
    # Construct the prompt with the chat history and user input
    prompt = f"{chat_history}\nChatGPT:"
    # Generate the response using the OpenAI API
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=prompt,
        max_tokens=200,
        temperature=0.7,
        n=1,
        stop=None,
        timeout=10,
    )
    # Return the generated response, stripping any leading/trailing whitespace
    return response.choices[0].text.strip()
```

This function generates a dialogue response using OpenAI's ChatGPT model. It takes the chat history and the user's input as parameters and returns the AI's response. It starts by setting the OpenAI API key and then constructs a prompt using the chat history and the user's input.

The function calls the OpenAI API to generate a response using the constructed prompt and specific parameters such as the engine (text-davinci-003), maximum token limit (200), temperature (0.7), number of responses to generate (1), and the timeout period (10 seconds). Finally, it returns the generated response after stripping any leading or trailing whitespace.

In addition to the AI capabilities, we have also incorporated other chatbots into our system - namely, MSBot from Microsoft and Blenderbot from Facebook. The selection of these chatbots is contingent upon the parameters passed, enabling us to tailor their usage according to specific requirements.

Here is the screenshot for the dialogue generation from other bots:

```
def provide_dialogue_msbot(tokenizer, model, msg, step, chat_history_encoded=None):
    """
    This function generates a response using Microsoft's MSBot model for a given user input and chat history.

    Parameters:
    tokenizer: The tokenizer used for tokenizing the input message.
    model: The pretrained MSBot model.
    msg (str): The user's input message.
    step (int): The current step of the conversation.
    chat_history_encoded (torch.Tensor, optional): The encoded chat history.

    Returns:
    chat_history_encoded (torch.Tensor): The encoded chat history including the current input message.
    response (str): The generated response from MSBot.
    """
    # Ensure no gradient is computed, to save memory
    with torch.no_grad():
        # Encode the current message, adding the EOS token at the end
        current_msg_encoded = tokenizer.encode(msg + tokenizer.eos_token, return_tensors="pt")

        # Append the new user input tokens to the chat history
        bot_input_encoded = (torch.cat([chat_history_encoded, current_msg_encoded], dim=-1)
                             if step > 0
                             else current_msg_encoded)
        bot_input_encoded = modify_history(bot_input_encoded)

        # Generate a response from the MSBot model
        chat_history_encoded = model.generate(bot_input_encoded, max_length=1000, pad_token_id=tokenizer
            .eos_token_id)

        # Decode the response
        response = tokenizer.decode(chat_history_encoded[:, bot_input_encoded.shape[-1]:][0], skip_special_tokens
            =True)

    return chat_history_encoded, response  # Return the encoded chat history and the response
```

In this function, the user's input message is first tokenized and encoded. If there are previous messages in the chat history, these are concatenated with the current message to form the full input for the MSBot model. The *modify_history()* function is called to preprocess the encoded input. The model then generates a response which is decoded and returned along with the updated encoded chat history. This allows the chat history to be used in subsequent calls to this function, enabling multi-turn conversations with MSBot.

```python
def provide_dialogue_blenderbot(tokenizer, model, msg, step=0, chat_history_encoded=None):
    """
    This function generates a response using Facebook's Blenderbot model for a given user input and chat
        history.

    Parameters:
    tokenizer: The tokenizer used for tokenizing the input message.
    model: The pretrained Blenderbot model.
    msg (str): The user's input message.
    step (int, optional): The current step of the conversation. Defaults to 0.
    chat_history_encoded (torch.Tensor, optional): The encoded chat history. Defaults to None.

    Returns:
    chat_history_encoded (torch.Tensor): The encoded chat history including the current input message.
    response (str): The generated response from Blenderbot.
    """
    # Ensure no gradient is computed, to save memory
    with torch.no_grad():
        # Encode the current message
        current_msg_encoded = tokenizer([msg], return_tensors="pt")

        # Generate a response from the Blenderbot model
        chat_history_encoded = model.generate(**current_msg_encoded)

        # Decode the response
        response = tokenizer.batch_decode(chat_history_encoded, skip_special_tokens=True)

    return chat_history_encoded, response[0]  # Return the encoded chat history and the response
```

In this function, the input message from the user is tokenized and encoded. The Blenderbot model generates a response from this input, which is then decoded to form a text string. This function then returns the encoded chat history along with the decoded response from the model. The encoded chat history can be used for subsequent turns in the conversation.

4.1 Japan-specific implementation

Compared to the EU system which has a restriction on generating dialogue responses sourced from the document store, the Japanese system fully leverages the OpenAI GPT-3.5 capability to generate responses. The difference is due to a less restrictive policy, which is allowed by the ethical approval of the use of generative AI on the Japan side. Figure 12 shows the diagram of the OpenAI use in the Japanese dialogue system.

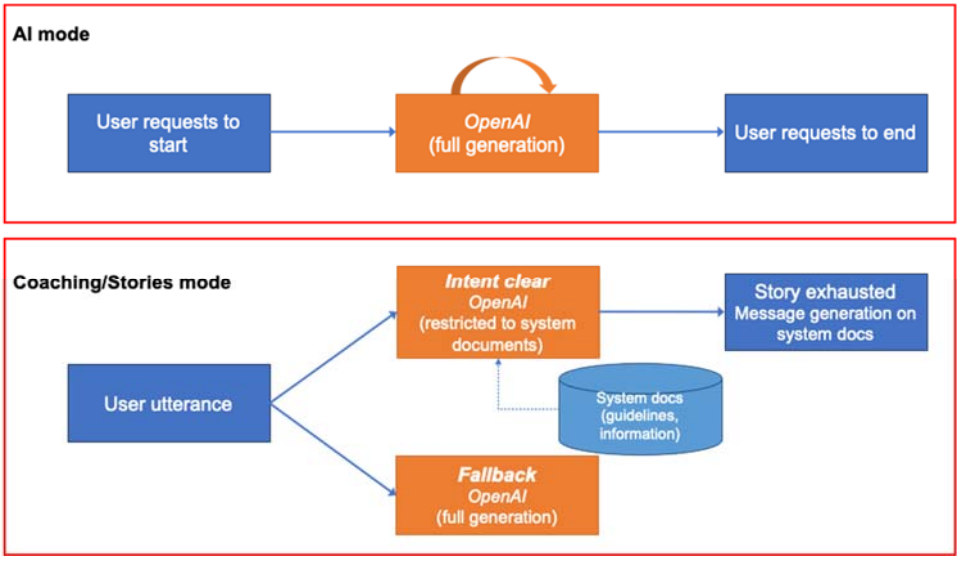

Figure 12 – Diagram of OpenAI use in the Japanese dialogue system

For example, the following story, shown in Figure 13, uses OpenAI's Langchain context to generate responses to user utterances representing specific intents in the cognitive, health, and nutrition domains. Figure 14 shows the fallback to using OpenAI GPT-3.5 for the AI modality.

```
stories:
- story: S03
  steps:
  - or:
    - intent: langchain_for_cognitive
    - intent: langchain_for_disease
    - intent: langchain_for_meal
  - action: action_langchain
```

```
- intent: langchain_for_cognitive
  examples: |
    - I'm worried about falls.
    - I'm worried about dementia.
    - I'm scared of getting cancer.
    - Cancer is very scary.
    - I'm concerned about disease.
    - How can I prevent dementia?
    - How can I protect against cancer?
    - Any ways to prevent cognitive decline?
    - I want to delay dementia.
    - How can I avoid falling over?
    - How can I improve my memory?
    - I want to increase my fitness.
    - I want to boost my mobility.
    - I want to increase my concentration.
    - I would like to improve my stamina.
    - I wish I was more flexible.
    - I'd like to be more fit.
    - What are types of brain training?
    - What is cognitive training?
    - What brain training should I do?
    - I want to know more details.
    - What exercise is good for me?
    - What are some types of exercise?
    - What exercises can I do indoors?
    - What are the recommended exercises for older adults?
    - What exercise do you recommend we do in the house?
    - I'd like to exercise outside. What do you think?
    - What do you recommend for exercise to do outside?
    - How many days a week should I exercise?
```

Figure 13 – Example of Stories and NLU with OpenAI use by using Langchain

```
def provide_dialogue_chatgpt(chat_history, user_input):
    openai.api_key = "XXXX...."
    print(f"chat_history: {chat_history}")

    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system",
             "content": "I am an older adult and you are my health coach. We are having a friendly chat, and we have talked about " +
                        f"{chat_history}" + " already. Limit your reply to whatever I say to two sentences. DO NOT ask a follow-up question."
            }
        ],
        max_tokens=200
    )
    return response.choices[0]["message"]["content"].strip()
```

Figure 14 – Fallback function to generate dialog messages by using OpenAI GPT-3.5

5 Conclusion

This document briefly described the tools used for the e-VITA Prototype 2, as they have evolved from Prototype 1. The tools consist of state-of-the-art machine-learning tools for the management of verbal interactions between the e-VITA coach and its users. The tools are highly configurable and extensible, and they ensure that the user experience is as close to a “natural” dialogue interaction as the current state of the art, and as well as safety considerations permit.

6 References

[1] Raffel, Colin, et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." Journal of Machine Learning Research 21 (2020): 1-67.

[2] Henderson, M., Casanueva, I., Mrkšić, N., Su, P. H., Wen, T. H., & Vulić, I. (2019). ConveRT: Efficient and accurate conversational representations from transformers. arXiv preprint arXiv:1911.03688.