UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MAURÍCIO BIASI DO MONTE CARMELO

An Experiment Environment for Question Answering Research

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Dante Augusto Couto Barone
Coadvisor: Prof. Ms. Eduardo Gabriel Cortes

Porto Alegre
May 2022

ACKNOWLEDGMENT

First of all, I thank my parents, Jackson and Luciana, for the education, affection, and love you have given me throughout my life. Without your unconditional support I would not have gotten this far. I thank my brothers, Fernando and Felipe, and cousins, Cleber and Hugo. Without your friendship, I don't know what I would do. I thank my uncle Emerson, who gave me valuable advice on college and studies and, without blinking, bought the first plane ticket that took me to Rio Grande do Sul for the first time.

I would also like to thank all the friends I made and who participated, in some way, in this adventure that started at FURG and ends now at UFRGS.

Finally, thanks to my advisor and professor Dante Barone, and co-advisor Eduardo Cortes for all the help that guided me in these first steps I took in academic production.

ABSTRACT

Building tests and experiments for Question Answering (QA) tends to be a hardworking task due to the fact that a lot of time is spent in the production of repeated code and tasks that could be automated. For example, there is a great number of datasets available and each of them organizes and structures information differently. Also, there are several QA benchmarks, which makes the task of testing an approach in different databases laborious. This work proposes the creation of a system that facilitates researchers to implement new techniques through the easy availability of different datasets, tasks, and pre-implemented techniques. Furthermore, the system architecture has been developed in such a way that having knowledge of the Python programming language is enough to implement and test new techniques. We have a functional prototype that performs the reading of different datasets and implements different techniques related to Question Processing, Information Retrieval, and Answer Processing phases. Empirical tests have shown that the system facilitates the implementation of techniques for the stages of question processing, information retrieval and answer processing.

Keywords: Natural Language Processing. Question Answering. Question Processing. Information Retrieval. Answer Processing. Experiment Environment.

LIST OF ABBREVIATIONS AND ACRONYMS

IR	Information Retrieval
CSV	Comma-separated values
ML	Machine Learning
MMQA	Multi-domain Multi-lingual Question-Answer
openQA	Open Question Answering
QA	Question Answering
SVC	Support Vector Classification
SVM	Support Vector Machines

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Question Answering (QA) systems are systems that aim to respond, autonomously and precisely, questions proposed in natural language. They differ from standard Search Engines, that receive a set of keywords and return to the users a list of relevant classified sources (DIMITRAKIS; SGONTZOS; TZITZIKAS, 2020). Question Answering is a field of study of Information Retrieval and Natural Language Processing.

The architecture of a QA system is usually divided in three parts: question processing, information retrieval, and answer processing. The first is responsible for interpreting and enrich the question with new information that might assist the next steps; the second, that resemble search engines, intent to rank relevant information from a database; the third step is responsible for processing and returning the final answer (CORTES et al., 2020).

In this context, developing new techniques, improving existing techniques, and even performing comparative tests are laborious and monotonous tasks. The availability of several datasets and techniques for each of the three parts of the architecture creates an enormous quantity of non-reusable code and, many times, diverts the attention of the scientist to tasks that can be automated. For example, a scientist that is working on a new technique of named entity recognition and desires to verify the impact of his/her study on the overall QA system will have to apply this new technique along with other tasks, and he/she will have the arduous work of configuring (or implementing) several other techniques.

The objective of this work consists in the development of a system base model [1] that facilitates the development of new techniques in the QA field. This system model have the main datasets for different tasks, which facilitates the execution and performance of the techniques. As result, there is a functional environment that abstracts from the user the implementation of multiple datasets, facilitates the implementation and usage of several QA techniques with dynamic modules that can be created and have its order defined in the pipeline. Moreover, this work contains two case studies that aim to identify the efficiency of the developed system, and an analysis of two related systems.

The next part of this text is divided in the following way. In chapter 2, a background is given, including a theoretical overview of QA systems, with details in regards to some used techniques in each one of the three steps (question processing, information

[1] https://github.com/MauricioCarmelo/QuestionAnsweringSystem

retrieval, answer processing). Besides that, the reader will be introduced to a few datasets and how the data are usually structured. Chapter 3 discusses about the simulation system that has been developed, with an explanation about what the user can expect, including technical details of the implementation. Chapter 4 presents two experiments that show what the user can expect while implementing new techniques and collecting metrics from them. Lastly, chapter 5 concludes the article and give insights on the limitations of the tool and future work.

2 BACKGROUND

This chapter provides the necessary theoretical background. It is presented in more detail what a QA system is and three fundamental steps that might constitute them. It also discusses some of the datasets that can be found in the literature and how the information is generally structured, and there is a brief explanation of some work that is related to the system proposed by this paper.

2.1 QA Systems

The QA field can be considered an advanced form of the Information Retrieval field (CAO et al., 2011). A QA system must have the capacity of interpreting and answering a great variety of complex and robust questions, generally divided among factoid and non-factoid. Factoid questions are identified as having immediate and accurate answers ("**Who** wrote the book Frankenstein?", "**How** many calories are in a fried egg?", "**What** is the average age of people with diabetes?", "**Where** is Mount Everest?". Non-factoid questions are open-ended questions that require robust answers, such as opinions, explanations, and descriptions (YANG et al., 2016).

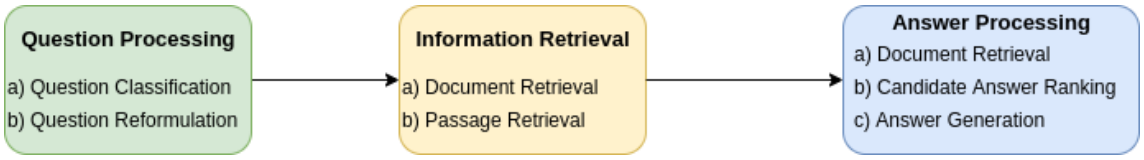
In addition to question type, QA systems can also be classified between close domain and open domain. Close domain are systems that focuses on answering questions related to a specific domain such as History and Medical. Meanwhile, open domain are systems whose goal is to be generic and aims to answers questions related to "anything". It is also possible to classify QA systems according to the type of knowledge base available for search in the Information Retrieval step, such as document and linked data (DIMITRAKIS; SGONTZOS; TZITZIKAS, 2020).

The medical use of QA systems, for example, requires a system that is capable of interpreting and classifying a question, seeking a set of satisfactory answers, and preparing an answer in such a way that a system exclusively based on pre-defined templates cannot do (CAO et al., 2011). To achieve the ability to answer questions in an automated way, QA systems typically consist of stages present in more detail in the next section.

2.2 General Architecture

The architecture of the QA system is fundamentally composed of three parts, as shown in Figure 2.1: 1) Question Processing, grouping tasks of Question Classification and Question Reformulation, 2) Information Retrieval, with tasks related to Document Retrieval and Passage Retrieval, and 3) Answer Processing, with three other task types, such as Candidate Answer Extraction, Candidate Answer Ranking, and Answer Generation (CORTES et al., 2022). This section presents each of these parts in detail and, in general terms, some of the techniques used.

Figure 2.1: General architecture of a QA system.



Source: The Authors

The next paragraphs present each of these parts and, in general lines, some of the techniques used.

2.2.1 Question Processing

The Question Processing step, which works directly with the question that was entered by the user in natural language, aims to interpret and enrich the question with information such as the domain of the question (weather, health, sports, etc.) and the type of information expected in the response (date, description, explanation, location, name, measure, object, organization, etc.) (CORTES et al., 2022).

The question can therefore go through a reformulation process to prepare it to increase the chances of finding a satisfactory answer in the database. An example of question reformulation task in the Portuguese language is to remove all stop words, such as the determinants (o, a, os, as, um, uma, uns, umas, este, esse, etc.) and the prepositions (desta, no, neste, nesta, nesse, nessa, etc.).

Question classification tasks play an important role in the overall process of finding the correct answer for an open-domain question. Classify a question in a semantic category not only reduces the search area in the next steps, but may also propose the

usage of different processing techniques (ZHANGL D., 2003). Techniques based on Support Vector Machines (SVMs), for example, are widely used to classify tasks due to its overall performance. A study proposed by Huang Zhiheng and Qin. (2008) achieved an accuracy of approximately 90% in a dataset arrangement with 50 question categories.

2.2.2 Information Retrieval

The Information Retrieval (IR) part is responsible for seeking relevant data to assist in formulating a final answer. A technique implemented for this purpose is heavily dependent on the available source of information. The source of information is a collection of objects (text files, databases, documents, video or web pages) that comprises the information available to the system for obtaining answers.

There are different techniques, algorithms, and frameworks related to information retrieval and extraction (SOARES; PARREIRAS, 2020). Usually, the search is done in two steps: a) search for relevant documents and, for reducing the amount of text, b) filtering passages (CORTES et al., 2022). Typically, an information retrieval module is straightforward: it receives a query as input and returns a list of relevant objects, ranked according to the plausibility of containing the correct answer to the proposed question (KOLOMIYETS; MOENS, 2011). The ranked objects can be documents or text passages, depending on the technique applied. The final response is expected to be contained within at least one of these objects returned by the technique.

2.2.3 Answer Processing

The Answer Processing step is responsible for structuring the final response that the user receives. All the information interpreted and extracted by the techniques in the previous parts is used. The preparation of the answer is performed in three parts: a) extraction of possible answers, b) classification of answers, to put in order the best answer options, and c) generation and selection of the answer (CORTES et al., 2022).

2.3 Datasets

QA experiments are heavily dependent on the quality of the dataset. One of the most labor-intensive steps to prepare a QA simulation is adapting the code to support the processing of different datasets, due to the fact that they are structured distinctly and may have files with different formats and names.

As this work makes extensive use of several previously prepared, consolidated, and widely used collections in the field of QA, some of these datasets were listed below with a brief explanation of how the information is structured in each of them.

2.3.1 WikiPassageQA

A collection of thousands of questions and answers based on Wikipedia articles. This dataset targets the field of non-factoid questions used for training deep learning models and collecting benchmarks. It is divided into 3332 training, 417 prediction and 416 testing questions, totaling 4165 questions.

Training, prediction and test questions are available in different files. In addition, a *.json* file is provided with excerpts referring to the relevant passages, that is, passages where the answer can be found (COHEN; YANG; CROFT, 2018).

2.3.2 SQuAD - Stanford Question Answering Dataset

With a set of more than 100,000 questions based on Wikipedia articles proposed by various people, where the answer to each of the questions is a text segment of a passage that can be found in one of these articles.

This collection contains exactly 107,785 question-answer pairs based on 536 articles (RAJPURKAR et al., 2016).

2.3.3 Antique

Hashemi H. and Croft (2020) created a dataset called Antique with a total of 2,626 non-factoid questions based on users' search at *Yahoo!*. These questions are divided in training and test sets with 2,426 and 200 questions, respectively.

2.3.4 MS MARCO - A Human Generated MAchine Reading COmprehension Dataset

Extensive collection of 1,010,916 questions collected from the search logs of the *Bing* search engine. Of all these questions, 182,669 answers were rewritten by humans. In addition, the dataset contains more than 8 million passages extracted from more than 3.5 million documents. It is important to note that some of the questions in this collection may have none or more than one answer (BAJAJ, 2016).

2.3.5 DuReader: a Chinese Machine Reading Comprehension Dataset from Real-world Applications

This collection of questions and answers is based on the *Baidu* search engine. It contains 200,000 questions, 420,000 answers and 1,000,000 documents. It is the largest Chinese MRC (*Machine Reading Comprehension)* dataset to date (HE KAI LIU, 2018).

It is important to highlight the fact that the answers present in this dataset were made manually.

2.3.6 Other Collections

In addition to the collections described above, others can be used for experiments in the QA field. Table 2.1 shows the relationship between the name of some of these datasets and the number of questions available.

Table 2.1: The number of questions of each dataset.

Dataset Name	**Number of Questions**
SQuAD	100,000
WikiPassageQA	4,165
Antique	2,626
MS-MARCO	1,000,000
DuReader	200,000
NarrativeQA	46,765
SearchQA	140,000

2.4 Related Work

There are similar literature studies, like Gupta D. (2018) and Marx E. (2018), that works towards the support in the research and development of QA systems. However, its main goal does not address the implementation and integration of new techniques.

Furthermore, there are open-source implementations of QA systems with particular strategies in each of the main steps (Question Processing, Information Retrieval, and Answer Processing) [1], and frameworks that share the objective of facilitating experiments in the area [2] [3].

Table 2.2 highlights the key differences between these two systems and the system presented in this work in relation to the initial objectives that led to the creation of this project.

Table 2.2: Key differences between the systems.

	WikiQA	**Qanary**	**Our system**
Easy to use	Yes	No	Yes
Easy to implement another component	Not possible	No	Yes
Support Question Classification	No	Yes	Yes
Support Information Retrieval	Yes	Yes	Yes
Support Answer Processing	No	Yes	Yes

2.4.1 WikiQA

This small sized application that focuses on retrieving information from models trained with *Wikipedia* articles. This system works alongside Wikipedia's search engine, as it queries the question in the search engine and then utilizes the output of the search as input for the QA model, which returns the extracted passages from the text.

This application is fundamentally different from the application presented in this work as the objective is not to facilitate the implementation of techniques and modules, but rather to allow a quick test on top of techniques that are already implemented.

[1] https://paperswithcode.com/task/question-answering
[2] https://github.com/cloudera/CML_AMP_Question_Answering
[3] https://github.com/WDAqua/Qanary-question-answering-components

2.4.2 Qanary Question Answering Components

This framework can be used to integrate QA components developed in the *Java* programming language, enabling rapid development of QA systems. It has a wide variety of components that can be reused by the community to build a *Qanary Pipeline* and execute QA approaches. Unlike our system, the Qanary framework is used to create real QA applications rather than to facilitate the creation and testing of new QA techniques.

This system has been developed in the Java programming language. Although this language is largely used by many programmers around the world, it is much less friendly than Python and it is not as used as Python by scientists in the field of QA. Also, in order to create a new component and integrate it with existing ones, the users need to understand about *Maven* and *Spring Boot*, frameworks that require a considerable amount of study and learning time. For these reasons, we are inclined to believe that this might not be attractive for a researcher that wants to quickly test and evaluate a QA approach under research and development.

Nevertheless, this framework is more robust than the one presented in this paper and it currently counts with a group of three core developers [4]. It also provides a Web UI that facilitates the integration of components.

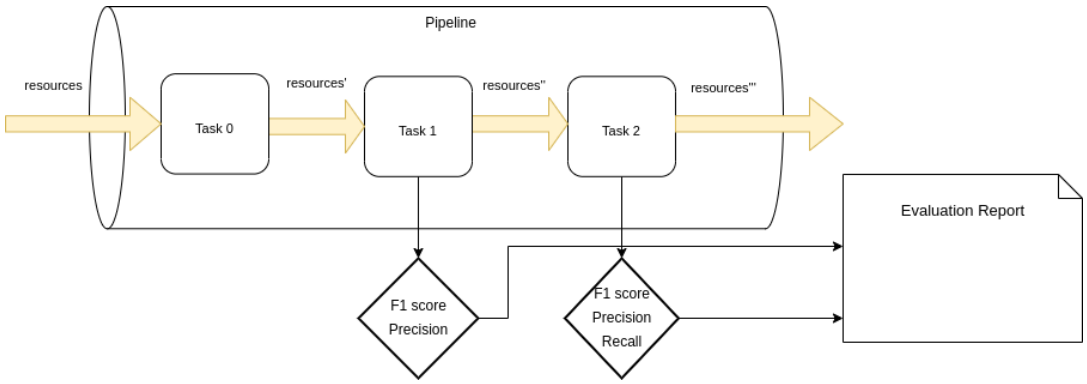
[4] https://github.com/WDAqua/Qanary/wiki/Who-do-I-talk-to%3F

3 TOOL FOR TESTING QUESTION ANSWERING TECHNIQUES

The proposed tool aims to support the research and development of the QA area through a system that facilitates QA experiments combining techniques and tasks in the QA pipeline to evaluate it with different datasets. The system allows users to create a pipeline of previously implemented tasks and collect benchmarks related to multiple datasets. Also, it is possible to implement new tasks, techniques, and evaluation metrics to employ them in a QA pipeline and evaluate them using different datasets. Once a new task is implemented it is possible to include this module anywhere in the execution pipeline. For example, in case the user wants the question domain information, they can simply include the question classification (section 2.2.1) task in the beginning of the pipeline and, as a consequence, the next tasks will have access to that information within the resource entries.

The system executes a simulation that generates an evaluation report. The simulation comprises a group of **Resources**, a **Pipeline**, and a chain of **Tasks** (Figure 3.1). A Resource (Figure 3.2) is an entity that encapsulates the information from a dataset, incorporating it into a generic structure. A Task is an agent that manipulates Resources by reading its data and may override or add new information into it. The Pipeline is a sequence of Tasks.

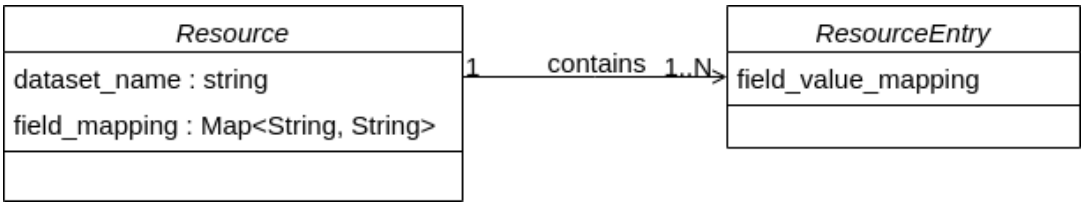
Figure 3.1: Pipeline of tasks.



Source: The Authors

Moreover, it is possible to evaluate each task with different metrics. For example, *Task 1* is evaluated by metrics *F1 score* and *precision* while *Task 2* is evaluated by the same metrics in addition to the metric *recall*. It is also possible to configure a task to not be evaluated. This scenario is used if the applied technique only assembles some data for

Figure 3.2: *Resource* and *ResourceEntry* class diagram.



Source: The Authors

the next tasks and does not require the evaluation of its generated results.

The rest of this chapter describes in more detail some technical aspects of the simulation execution flow, how the system loads information from a dataset and uses it within the pipeline of tasks, and how the result report is generated based on the metrics that the user expects to collect. It is also shown insights on how to configure a simulation through the configuration files.

3.1 Characteristics of the Tool

The main characteristic of this tool is to facilitate the implementation of techniques derived from the types of tasks mentioned on Figure 2.1 (Question Classification, Question Reformulation, Document Retrieval, Passage Retrieval, Candidate Answer Extraction, Candidate Answer Ranking, and Answer Generation).

The tool architecture has been conceived in such a way that it is easy to implement new techniques, new classes to read information from other datasets, and new evaluation methods.

With this tool it is possible to:

- use the output of a technique as input for other techniques in the pipeline.
- quickly configure a simulation and collect concrete results with the techniques that different users implemented.
- easily implement a new dataset reader to parse information from a not (yet) supported dataset.
- remove entries by filtering undesirable lines of the dataset according to any parameter. This can be achieved without implementing a single line of code, as it can be configured in a *.yaml* file.

– execute a simulation with multiple datasets for a single pipeline of tasks, collecting more results and facilitating the comparison.

– calculate various metrics, including *F1 score*, *precision*, *accuracy* and *recall*.

– easily add a new metric to the tool and use it to evaluate any existing technique.

3.2 Simulation Execution Flow

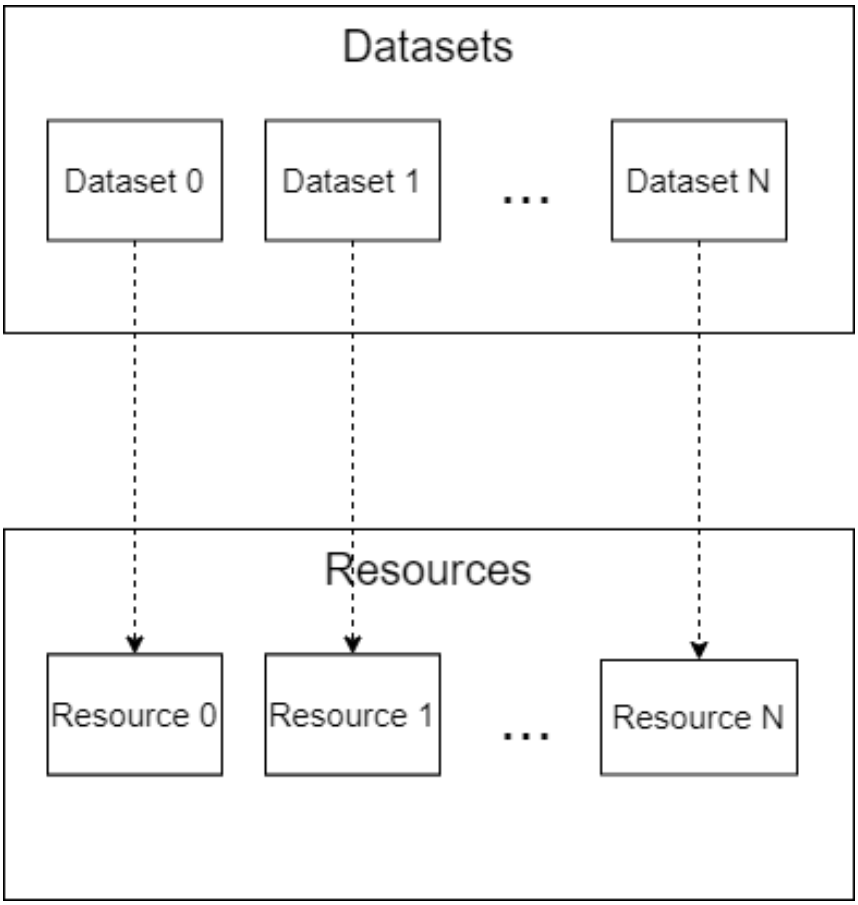
The user has access to two configuration files: *datasets.yaml*, used to set up the datasets that might be used during the simulation, and *pipeline.yaml*, used to configure the chain of tasks in the pipeline. These files are easy to understand, and there is a documentation in the tool's official repository.

This section presents an overview of the simulation execution flow. Starting by loading the information contained in the datasets, execution of the tasks that make up a pipeline and, finally, generating a report.

3.2.1 Load Data From The Dataset

Initially, the system loads information from the configured datasets into an abstraction called *Resource*. In a single simulation, it is possible to load more than one dataset at the same time. As shown in Figure 3.3, each *Resource* encapsulates all the information within a single dataset through the simulation lifetime. Each dataset is abstracted into a *Resource*.

Figure 3.3: Mapping of information from datasets to Resources.



Source: The Authors

The data in the dataset is separated into groups of information and each one of these groups is inserted into the structure showed in the Python File 3.1, called *Resource Entry*.

File 3.1: Resource Entry structure in Python

```
resource_entry = {
    "id": None,
    "question": None,
    "question_domain": None,
    "answer_type": None,
    "answers": [{"id": None,
                 "answer": None,
                 "documents": [{"id": None,
                                "name": None,
                                "document": None}],
                 "passages": [{"id": None,
                               "name": None,
                               "passage": None}],
```

```
                    "sentences": [{"id": None,
                                   "name": None,
                                   "sentence": None}]}
                ],
    "entities": [{"entity": None,
                  "start": None,
                  "end": None,
                  "type": None,
                  "subtype": None}],
    "tokens": [],
    "pre_evaluation_group": None,
    "evaluation_group": None
}
```

A group of information is understood to be all data related to a single question present in the dataset; usually this data is located in a single row of the set. A *Resource* (Figure 3.2) consists of a list of *Resource Entries*. Table 3.1 presents an explanation of each of the most used fields of the resource entry structure 3.1.

Table 3.1: *ResourceEntry* fields definition.

Field name	**Field description**
id	Question identifier.
question	Question text.
question_domain	Answer domain (weather, health, sports, etc.).
answer_type	Type of information expected in the response (date, description, explanation, location, name, measure, object, organization, etc.).
answers	Data related to the expected answers.
answer.id	Answer identifier.
answer.document	Relevant documents containing the answer.
answer.passage	Possible passages containing the answer.
answer.sentences	Possible sentences containing the answer.
pre_evaluation_group	Some datasets split the inputs between training (train), prediction (dev) and test (test). One of these values is entered in this field.

Currently, the tool supports the following datasets: QAChave (SANTOS; ROCHA, 2004), available in Portuguese language, WikiPassageQA (COHEN; YANG; CROFT, 2018) and Antique (HASHEMI H.; CROFT, 2020), both available in English language, and UIUC (LI; ROTH, 2002), available in Portuguese, English and Spanish languages.

To use any of these datasets in a simulation, it is required to configure file *datasets.yaml* accordingly. The configuration File 3.2 is an example of configuration that makes avail-

able for simulation the datasets SQUAD (RAJPURKAR et al., 2016) and WikiPassageQA (COHEN; YANG; CROFT, 2018). There are instructions on how to use and descriptions of each field in the documentation available in the repository [1].

File 3.2: Configuration of datasets that will be loaded.

```yaml
dataset:
  name: "SQUAD"
  reader_type: "SQUAD"
  path: "datasets/SQUAD/"
  dataset_setup:
    type: "fixed-split"
---
dataset:
  name: "WikiPassageQA"
  reader_type: "WikiPassageQA"
  path: "datasets/WikiPassageQA/"
  dataset_setup:
    type: "cross-validation"
    folds_splitter: "shuffle-split"
    folds: 5
    test_size: 0.4
    random_state: 0
```

Moreover, this excerpt of code signals the fold strategy applied during the usage of the resources. *WikiPassageQA* is configured to be shuffled in five different folds and with a proportion of 40% of the dataset to be separated as a test set. Meanwhile, *SQUAD* is configured to be separated as *fixed-split*, which means, in more technical terms, that the training and prediction sets are based on what is stored in field *pre_evaluation_group* of each resource entry structure (File 3.1).

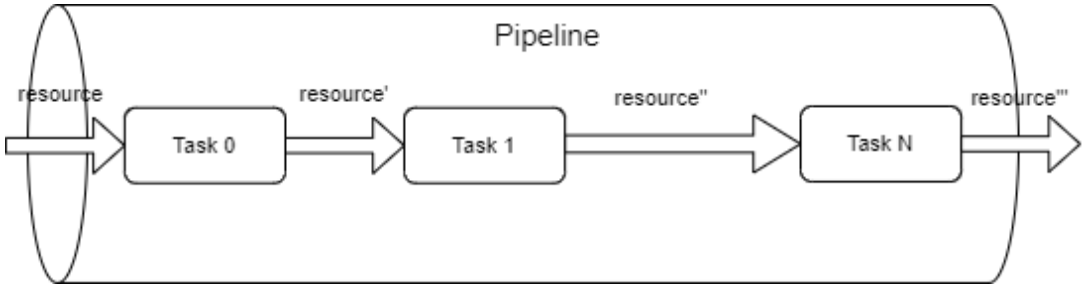
3.2.2 Pipeline Execution

A *Pipeline* is composed of a sequence of tasks that were previously implemented. Note on Figure 3.4 that the *Resource* undergoes a modification after each task. The modification changes or adds a value to each of the *Resource Entries* stored in the *Resource*. This behavior can be seen in more detail on Figure 3.5, where the value generated by task 0 is embedded in the *ResourceEntry* object. This feature plays an important role in the simulator itself, making it very flexible, as the output of one task can serve as input for later tasks. A *ResourceEntry* instance encapsulates all information referring to a group of information, including the values created or modified in each of the tasks that make up

[1] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/docs/dataset_fields.md

the *Pipeline*.

Figure 3.4: Schematic of tasks being executed in a pipeline.



Source: The Authors

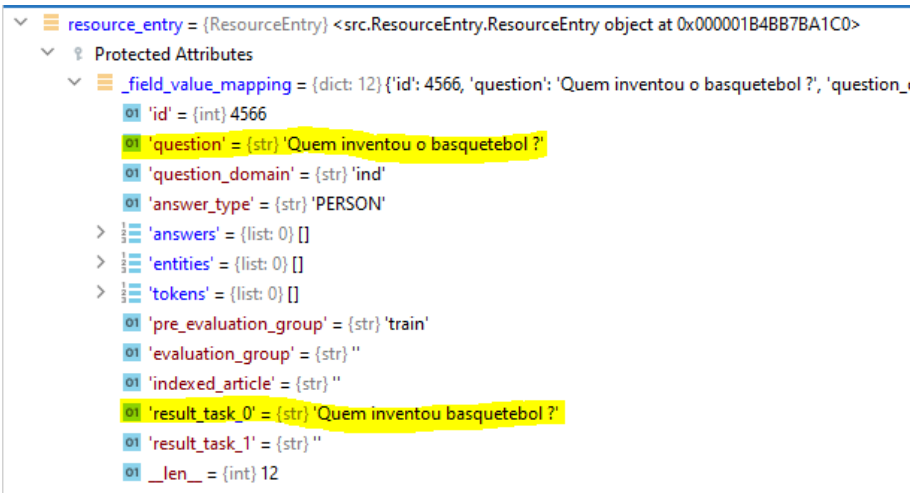
A *Task* is an abstraction that encapsulates a *Technique*. The user can implement a technique of any task type: Question Classification, Question Reformulation, Document Retrieval, Passage Retrieval, Candidate Answer Extraction, Candidate Answer Ranking, and Answer Generation.

The pipeline of tasks is configured in *pipeline.yaml*. Although configuration is easy, there are several fields and parameters that can be used and the user should read the documentation available in the repository [2] in order to fully understand and use all the available features of the system.

Configuration File A.4 creates a simulation with only one task while configuration file A.5 creates a simulation with two tasks in sequence. Although they are both using technique *LinearSVCQuestionClassification*, the input will be the result of the of technique *nltkTokenizerWithoutStopWords* in the second one. Note that technique *LinearSVCQuestionClassification* is supposed to use the value *question_text* while reading the information from dictionary *resource_entry* (File 3.1); however, configuration in File A.5 signals the technique to use the value from result_task_0 instead.

[2] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/docs/pipeline_fields.md

Figure 3.5: Resource entry holding the value generated by task 0 in a debug session.



Source: The Authors

3.2.3 Report Generation

At the end of the simulation, it is generated a report in CSV format with the metrics that were calculated for the techniques and resources that execute in pipeline.

It is possible to evaluate each technique with different metrics. In the example shown on Figure 3.1, Task 2 is evaluated by metrics *F1 score* and *precision* while Task 3 is evaluated by metrics *F1 score*, *precision* and *recall*. Also, Task 0 is configured to not be evaluated. This can be used if the applied technique only assembles some data for the next tasks and does not require the evaluation of its generated results.

File 3.3: Configuring the evaluation of a task.

```yaml
evaluation:
should_evaluate: true
type: "DocumentRanking"
set_usage:
  evaluate_train: false
  evaluate_dev: false
  evaluate_test: true
fields:
  answer: "result_task_0"
generated_result: "result_task_0"
```

The excerpt of code in File 3.3 demonstrates how to configure an evaluation for a given technique. In this example it is used an evaluation of type *"DocumentRanking"*, which receives a list of relevant documents and its scores and, currently, it is able to

calculate both metrics *recall* and *mean average precision*. The configuration highlights that the evaluation method should use the result of task 0 as input and, under *set_usage*, it is configured to perform the evaluation only for the test set.

3.3 Tool Extensibility

The tool's architecture encourages new users to contribute. Via Python code, it is possible to add new dataset readers and implement and share new tasks with other users. Details on how to implement a new task and a new dataset reader are described in the documentation. Once the task is merged to the main branch of the repository, all users also have access to this code, and they will also be able to configure the pipeline to run the new task.

This section walks the reader through the steps how a user can contribute by implementing another dataset reader or a new technique. A more complete documentation is available at the Github repository [3].

3.3.1 Implement another dataset reader

To read information from a dataset that is not supported by the application, it is necessary to implement another **Dataset Reader**. This component is responsible for reading all the information from the dataset and store it into a *Resource*.

Step by step on how to implement a dataset reader:

1. Give the dataset a name (this name will be used in the configuration file).
2. Create a new enumeration type in *ImplementedDatasetReaders*. This enumeration is linked to a name that allude to the dataset itself. It is suggested to insert a comment in the *ImplementedDatasetReaders* file with the name created on step 1, in front of the created type.
3. Create a class that extends *DatasetReader* and save it at *"./src/datasetreader/"*
4. Implement method *DatasetReader.load_entries()*. This method needs to return a *list()* of resource entries, where each resource entry represents a group of information in the dataset. The resource encapsulates the Python dictionary shown on File

[3] https://github.com/MauricioCarmelo/QuestionAnsweringSystem

3.1. If the dataset does not have an information in particular, it is possible to leave that value empty.

5. Adjust method *BuilderDatasetReader.build_dataset_reader()* to return the correct class according to the recently created *ImplementedDatasetReaders* enum type.

6. Adjust method *SettingsYAML.determine_reader_type()* to return the reader type accordingly.

Method *DatasetReader.load_entries()* is particularly important because it is resposible for reading all the information related to the dataset and stored it in different resource entries.Each resource entry is an instance of class *ResourceEntry*, that can be found on *QuestionAnsweringSystem/src/ResourceEntry.py*.

For more details on how to implement another dataset reader, please refer to this documentation [4].

3.3.2 Implement another task and/or technique

An instance of a *Task* is the place where the technique runs. Step by step on how to implement a task:

1. Give the task a name (this name will be used in the configuration file).

2. Create a class that extends *Task* and save it at *"./src/tasks/"*.

3. Implement the abstract method *Task.run()* that returns a list with the results generated by the task for every resource entry that the technique were applied.

4. Adjust method *Simulation.__build_task()* by adding a new condition in order to return the correct class according to the task name.

An instance of a technique is where the information of each *ResourceEntry* is processed, generating a result that will be stored on the *ResourceEntry* itself. Step by step on how to implement a technique:

1. Give your technique a name (this name will be used in the configuration file).

2. Create a class that extends *Technique* and save it at *"./src/tasks/"*.

[4] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/docs/implement_dataset_reader.md

3. Implement the abstract method *Technique.run()* that returns a list with the results generated by the technique for every resource entry that the technique were applied.

4. Adjust method *Task.build_technique()* by adding a new condition in order to return the correct class according to the technique name.

5. It is possible to override methods *Technique.setup()*, *Technique.train()* and *Task.validate()* to perform more complex operations, including the usage of Machine Learning (ML).

For more information on how to implement another task and/or technique, please refer to the documentation available in the tool repository [5].

[5] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/docs/implement_task_and_technique.md

4 TESTS AND EXPERIMENTS

This chapter presents two experiments that were conducted with the aim of testing the ease of extending the tool. The first experiment consists of testing a question classification approach varying the number of tasks in the pipeline. It was necessary to implement a new dataset reader and two techniques. The second experiment focuses on testing an information retrieval technique using *Elasticsearch*. It was implemented a new dataset reader and one technique.

During the development, others tests were carried out in relation to the functioning of the system and the steps necessary to create a new dataset reader and other techniques. Currently, there is no technique related to answer processing implemented. This is suggested as an approach for future work.

4.1 Question Classification test

The first experiment consists in testing a question classification technique alongside two important features of the tool: concatenating more than one task in the pipeline and filtering undesirable lines of the dataset. It is intended to show how the final result of the collected metrics can change with simple changes in the configuration files *pipeline.yaml* and *datasets.yaml*.

This test encompasses the execution of four simulations, as below:

A Raw dataset (without filtering); raw question texts (original question text).

B Raw dataset; removing stop words (definition on section 2.2.1) from questions.

C Filtering questions whose answer type is equal to *"OTHER"*; raw question texts.

D Filtering questions whose answer type is equal to *"OTHER"*; removing stop words from questions.

Note: The complete code and configuration files used in simulation D can be found in the Github repository, branch TCC_TEST_1_SIM_D [1].

[1] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/tree/TCC_TEST_1_SIM_D

4.1.1 Load the dataset

It was decided to use dataset UIUC (LI; ROTH, 2002) in this experiment. This collection provides 5957 questions and their answer type. Also, there are 2740 questions whose answer type was normalized to *"OTHER"*; these questions are particularly interesting to be filtered out from the dataset before classifying it.

To successfully load the information from UIUC dataset it was necessary to following the steps mentioned on section 3.3.1. Once the name that allures to the dataset was given (UIUC) and the enumeration type created on class *ImplementedDatasetReaders*, it was created class *DatasetReaderUIUC* [2] that extends class *DatasetReader*. This class is responsible for reading the information from the files made available by the dataset creators and putting all relavant data into resource entry [3] objects.

Then, after adjust on methods *BuilderDatasetReader.build_dataset_reader()* and *SettingsYAML.determine_reader_type()* to take into consideration the new *dataset reader*, a *Resource* containing the data from dataset UIUC can be used by the simulation simply by adding the configuration shown on Files A.1 and A.2.

To execute a simulation with all questions from the dataset, it was used the configuration shown on File A.1. This configuration indicates the usage of 5 folds (re-shuffling and splitting iterations) and that, in each iteration, 40% of the dataset should be included in the test split. In order to configure the filtering of all questions whose answer type is equal to "OTHER", it was simply necessary to add tag *filter* into the configuration of the dataset (File A.2) and specify the resource entry field that should be taken into account while filtering.

4.1.2 Prepare the techniques

For classifying the questions, it was created class *TechniqueLinearSVCQuestionClassification* that extends *Technique*, according to the instructions described in section 3.3.2. It was used the Linear Support Vector Classification (SVC) [4] function, available in the *sklearn* [5] package.

This technique implements the *train()* function to train a model with the questions

[2] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/src/datasetreader/DatasetReaderUIUC.py
[3] https://github.com/MauricioCarmelo/QuestionAnsweringSystem/blob/main/src/ResourceEntry.py
[4] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
[5] https://scikit-learn.org/stable/index.html

from the train set. To train a model within the technique's class, the user simply has to extend function *train* from class *Technique*. There is also function *setup()* that can be implemented in order to prepare the environment for training, as the following:

File 4.1: *setup* function implemented in the *Technique*'s child class

```
1 self.model = LinearSVC()
2 self.vectorizer = TfidfVectorizer()
```

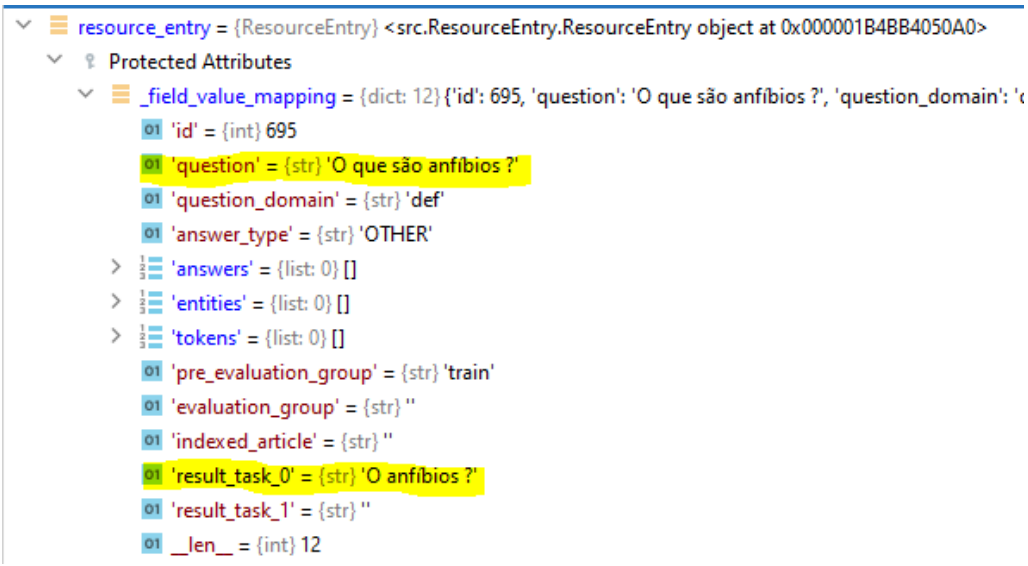
It was used variations in the input to train and test the technique with question texts as provided by the dataset (raw questions) and question texts without the stop words. In order to remove the stop words from the questions, it was necessary to add a new technique in the beginning of the pipeline (configuration shown in File A.5). For this purpose, it was implemented a technique that uses package *nltk* [6] to download the stop words, available in many languages, including Portuguese and English.

4.1.3 Collect metrics

The results from simulation A (result File B.1), simulation B (result File B.2), simulation C (result File B.1) and simulation D (result File B.4) listed in Appendix B show that filtering the questions increased the score of all metrics. This was expected because removing questions without a defined answer type creates more robust training and test sets. However, removing the stop words from questions had a negative impact in the overall results. This happened because removing stop words deliberately can oversimplify a question, causing it to lose context (as shown on Figure 4.1). Table 4.1 shows the average values and Figure 4.2 illustrates the collected metrics *f1_score*, *precision*, *accuracy* and *recall* for all four simulations.

[6] https://www.nltk.org/

Figure 4.1: Question too simple after removing stop words.

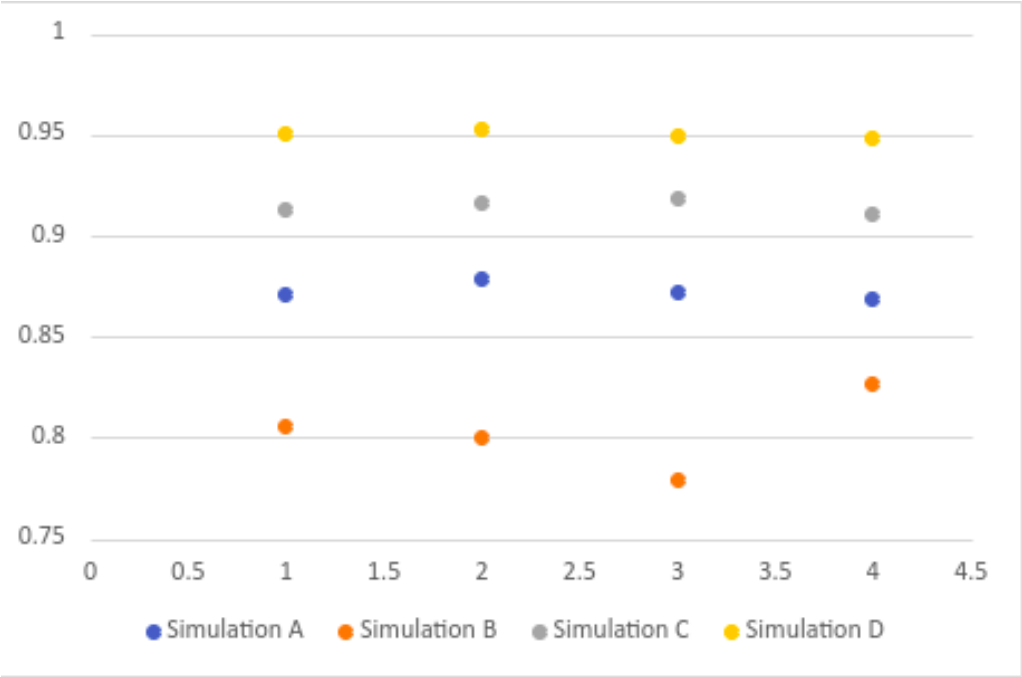


Source: The Authors

Table 4.1: Average values of the simulations.

	f1_score	**precision**	**accuracy**	**recall**
Simulation A	0.871102404	0.878193141	0.87167436	0.868308725
Simulation B	0.805339751	0.800393504	0.779101972	0.826190875
Simulation C	0.912408646	0.916526591	0.918259518	0.910536184
Simulation D	0.950476693	0.952957884	0.949339549	0.948571752

Figure 4.2: Result values of the simulations.



Source: The Authors

4.2 Information Retrieval test with *Elasticsearch*

The second experiment focuses on testing an IR technique mentioned in the end of section 2.2.1, where it is created an IR module that extensively uses *Elasticsearch* to retrieve relevant documents from a knowledge base.

Labs (2020) presents an information retrieval module that uses *Elasticsearch*[7] to implement a technique based on a component called Retriever. *Elasticsearch* can be used as a powerful IR tool to scrub large amount of text files and documents. The Retriever, as its name suggests, retrieves potentially relevant documents from a source of information and rank them according to a **relevance score**, calculated by *Elasticsearch match* function, which uses a technique call search time analysis to query the text.

The instructions given by Labs (2020) were adapted to fit into the system presented in this article. The next sections describe in details how this details how this was done. It is intended to show how easy it is to use the system alongside other tools.

[7]https://www.elastic.co/

4.2.1 Load the dataset

Dataset SQUAD (RAJPURKAR et al., 2016), provides a great number of questions and raw text that this experiments requires. The text that might contain the answers to the questions are available in the dataset files *dev-v2.0.json* and *train-v2.0.json*. Also, for each question, there is a field with the correct answer.

To load the information it was created class *DatasetReaderSQUAD* that extends *DatasetReader*. It was implemented both function *load_entries()* and *load_articles()* to read, respectively, the question information that is encapsulated into resource entries and the information related to the texts that might contain the answers to these questions. For correct execution of the dataset reader, it was necessary to follow the instructions mentioned in section 3.3.1.

4.2.2 Prepare the technique

This technique demands connection to Elasticseach tool to execute correctly. Method *setup()* was used to establish connection on port 9200 and to create the index [8], an abstraction provided by *Elasticsearch* used to store text in a structure that allows efficient search. Once the connection is established and the index created, method *train()* was used for indexing data.

It was implemented class *TechniqueRetrievalBasedInformationRetrieval* that extends *Technique*. The model was populated with 20239 resource articles (text passages) and tested with approximately 12000 questions. The output of this technique is a list with 10 (this value can be configured) relevant documents, ranked according to the likelihood of containing the correct answer.

The excerpt on File 4.2 shows how the output result for the question "In what country is Normandy located?", with list of document and its score:

File 4.2: Output result of technique *TechniqueRetrievalBasedInformationRetrieval*

```
result = [
    ('United_Nations_Population_Fund_17', 11.242065),
    ('Republic_of_the_Congo_22', 10.654305),
    ('Normans_14', 10.616264),
    ('Republic_of_the_Congo_24', 10.169264),
    ('Central_African_Republic_15', 10.161721),
    ('Raleigh,_North_Carolina_15', 9.984276),
```

[8] https://codingexplained.com/coding/elasticsearch/understanding-the-inverted-index-in-elasticsearch

```
8       ('Tucson,_Arizona_19', 9.836688),
9       ('Middle_Ages_35', 9.64839),
10      ('Middle_Ages_49', 9.511383),
11      ('Normans_30', 9.229682)
12  ]
```

4.2.3 Collect metrics

To evaluate an IR technique and collect metrics, it was implemented a new evaluation method, called *DocumentRanking*. This *Evaluator* passes through all the resource entries and checks if the correct answer can be found in each one of the documented that were ranked as relevant by the technique. Note that in order to do this it is required that the dataset provides the correct answer for comparison purposes.

This evaluator calculates the *mean average precision* and *recall*. Recall indicates the correct answer is present in any of the retrieved documents and mean average precision indicates the presence of the answer in the retrieved documents taking into account the ranking position of the list, for example, an output where the correct answer is in the top-most document has a higher mean average precision than an output where the correct answer is only in the last document ranked by the technique.

This experiment return the CSV file B.5. Table 4.2 describes these results.

Table 4.2: Results of the experiment.

Mean Average Precision	0.885
Recall	0.959

5 CONCLUSION

This work proposes the modeling of a base system for carrying out experiments in the area of QA. The main objective is to facilitate the performance of benchmark tests in the stages of Question Processing, Information Retrieval and Answer Processing. The main idea was to make the system as generic and extensible as possible.

The user has at his disposal a configuration file where they can determine the tasks and the order in which they will be executed in the pipeline. In addition, there are specific classes that can have their behavior inherited by others to implement new techniques, tasks, data set readers, and evaluation metrics. Therefore, if someone wants to use a data set or an unsupported technique, that person simply contributes to the project following steps specified in the documentation.

There are straightforward instructions on how the user can contribute in the tool repository, as well as a documentation on how to set up a pipeline of techniques.

With this work, we hope to make the preparation and execution of experiments in the field of QA as simple as possible, removing from the scientist the need to produce non-reusable code.

As an output of this work we had a demonstration paper [1] accepted in the 15th International Conference on the Computational Processing of Portuguese (PROPOR) 2022 [2]. The event took place remotely and was based in the city of Fortaleza, in Brazil.

5.1 Limitations of the tool

Like most software tools, development is a never-ending process due to features that can be included and bugs that need to be fixed and found. Below, a list of known bugs and require fix and future enhancements that can make the tool more efficient and robust. Also, it is presented a few ideas for future features that would make this tool better and more user friendly.

[1] https://sites.universidadedefortaleza.com/propor2022/?page_id=177
[2] https://sites.universidadedefortaleza.com/propor2022/

5.1.1 Known bugs and enhancements

- Enhance the way the techniques access the input field when the technique receives as input the output of a previous one. It is necessary to standardize this because currently a lot is in the hands of the programmer.
- Support more datasets; there are many other datasets available.
- Implement a technique related to Answer Processing.
- Currently the result report is being generated with repeated header. The column names should be only in the top of the file.
- Evaluate what other information may be relevant to the user and print it in the result report.

5.1.2 Ideas for future features

- Create a graphical interface so the user can manipulate the modules and create the pipeline in a more friendly way.
- Some datasets cannot be fully loaded during runtime. In this situation, it would be necessary to change the simulation process to parse the dataset in parts. This feature requires a major change in the system.

REFERENCES

BAJAJ, D. C. P. Ms marco: A human generated machine reading comprehension dataset. 2016.

CAO, Y. et al. Askhermes: An online question answering system for complex clinical questions. **Journal of Biomedical Informatics**, v. 44, p. 277–88, 2011.

COHEN, D.; YANG, L.; CROFT, W. B. Wikipassageqa: A benchmark collection for research on non-factoid answer passage retrieval. 2018.

CORTES, E. et al. A systematic review of question answering systems for non-factoid questions. **Journal of Intelligent Information Systems**, v. 58, p. 453–480, 2022.

CORTES, E. et al. An empirical comparison of question classification methods for question answering systems. **Proceedings of the 12th Language Resources and Evaluation Conference**, p. 5408–5416, 2020.

DIMITRAKIS, E.; SGONTZOS, K.; TZITZIKAS, Y. A survey on question answering systems over linked data and documents. **Journal of Intelligent Information Systems**, Publishing Press, v. 55, p. 233—259, 2020.

GUPTA D., K. S. E. A. B. P. Mmqa: A multi-domain multi-lingual question-answering framework for english and hindi. **European Language Resources Association (ELRA)**, 2018.

HASHEMI H., A. M. Z. H.; CROFT, W. Antique: A non-factoid question answering benchmark. **In European Conference on Information Retrieval**, p. 166–173, 2020.

HE KAI LIU, J. L. W. Dureader: a chinese machine reading comprehension dataset from real-world applications. 2018.

HUANG ZHIHENG, M. T.; QIN., Z. Question classification using head words and their hypernyms. **Proceedings of the 2008 Conference on empirical methods in natural language processing**, p. 927–936, 2008.

KOLOMIYETS, O.; MOENS, M.-F. A survey on question answering technology from an information retrieval perspective. **Information Sciences**, v. 181(24), p. 5412–5434, 2011.

LABS, C. F. F. **Evaluating QA: the Retriever the Full QA System**. 2020. Available from Internet: <https://qa.fastforwardlabs.com/elasticsearch/mean\%20average\%20precision/recall\%20for\%20irqa/qa\%20system\%20design/2020/06/30/Evaluating_the_Retriever_&_End_to_End_System.html>.

LI, X.; ROTH, D. Learning question classifiers. In: **Proceedings of the 19th International Conference on Computational Linguistics - Volume 1**. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002. (COLING '02), p. 1–7. Available from Internet: <https://doi.org/10.3115/1072228.1072378>.

MARX E., S. T. E. D. N. A. L. J. Wikipassageqa: An open question answering framework. 2018.

RAJPURKAR, P. et al. Squad: 100,000+ questions for machine comprehension of text. **Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing**, p. 2383–2392, 2016.

SANTOS, D.; ROCHA, P. The key to the first clef with portuguese: Topics, questions and answers in chave. In: **CLEF**. [S.l.: s.n.], 2004.

SOARES, C.; PARREIRAS. A literature review on question answering techniques, paradigms and systems. **Journal of King Saud University - Computer and Information Sciences**, v. 32, p. 635–646, 2020.

YANG, L. et al. Beyond factoid qa: Effective methods for non-factoid answer sentence retrieval. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag**, v. 9626, p. 115—128, 2016.

ZHANGL D., L. W. S. Question classification using support vector machines. **Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval**, p. 26–32, 2003.

APPENDIX A — CONFIGURATION FILES

File A.1: Configuration of dataset UIUC.

```yaml
dataset:
  name: "UIUC"
  reader_type: "UIUC"
  path: "datasets/UIUC/UIUC_pt"
  dataset_setup:
    type: "cross-validation"
    folds_splitter: "shuffle-split"
    folds: 5
    test_size: 0.4
    random_state: 0
```

File A.2: Configuration of dataset UIUC with filter.

```yaml
dataset:
  name: "UIUC"
  reader_type: "UIUC"
  path: "datasets/UIUC/UIUC_pt"
  dataset_setup:
    type: "cross-validation"
    folds_splitter: "shuffle-split"
    folds: 5
    test_size: 0.4
    random_state: 0
  filter:
    answer_type: 'OTHER'
```

File A.3: Configuration of dataset SQUAD.

```yaml
dataset:
  name: "SQUAD"
  reader_type: "SQUAD"
  path: "datasets/SQUAD/"
  dataset_setup:
    type: "fixed-split"
```

File A.4: Configuring a pipeline with only one task.

```yaml
task:
  id: 0
  ignore: false
  name: "answer_type_classification"
  technique: "LinearSVCQuestionClassification"
  predicts:
    predict_train: false
    predict_dev: false
    predict_test: true
  used_datasets:
    - used_dataset:
        name: "UIUC"
        input_fields:
  evaluation:
    should_evaluate: true
    type: "ValueComparison"
    set_usage:
      evaluate_train: false
      evaluate_dev: false
      evaluate_test: true
    fields:
      answer_type: "result_task_0"
    metrics:
      f1_score:
        average: "macro"
      precision:
        average: "macro"
      accuracy:
        -
      recall:
        average: "macro"
  generated_result: "result_task_0"
```

File A.5: Configuration of a pipeline with 2 tasks.

```yaml
task:
  id: 0
  ignore: false
  name: "generate_query"
  technique: "nltkTokenizerWithoutStopWords"
  predicts:
    predict_train: true
    predict_dev: false
    predict_test: true
  used_datasets:
    - used_dataset:
        name: "UIUC"
        input_fields:
  evaluation:
    should_evaluate: false
  generated_result: "result_task_0"
---
task:
  id: 1
  ignore: false
  name: "answer_type_classification"
  technique: "LinearSVCQuestionClassification"
  predicts:
    predict_train: false
    predict_dev: false
    predict_test: true
  used_datasets:
    - used_dataset:
        name: "UIUC"
        input_fields:
          question: "result_task_0"
  evaluation:
    should_evaluate: true
    type: "ValueComparison"
    set_usage:
      evaluate_train: false
      evaluate_dev: false
      evaluate_test: true
    fields:
      answer_type: "result_task_1"
    metrics:
      f1_score:
        average: "macro"
      precision:
        average: "macro"
      accuracy:
        -
      recall:
        average: "macro"
  generated_result: "result_task_1"
```

APPENDIX B — EXPERIMENTS' RESULTS

File B.1: Result of Experiment 1 Simulation A

```
task_id,dataset_name,fold,result_type,value
0,UIUC,0,f1_score,0.8780717744181242
0,UIUC,0,precision,0.875514418554672
0,UIUC,0,accuracy,0.8778850188837599
0,UIUC,0,recall,0.8852969524362029
0,UIUC,1,f1_score,0.8668528502361008
0,UIUC,1,precision,0.877628757616481
0,UIUC,1,accuracy,0.8699118757868234
0,UIUC,1,recall,0.85963822000068
0,UIUC,2,f1_score,0.8672357566197219
0,UIUC,2,precision,0.8753865759740579
0,UIUC,2,accuracy,0.8652958455728074
0,UIUC,2,recall,0.8651638571523348
0,UIUC,3,f1_score,0.8757981444745202
0,UIUC,3,precision,0.880433401428386
0,UIUC,3,accuracy,0.878304657994125
0,UIUC,3,recall,0.8751931743398572
0,UIUC,4,f1_score,0.8675534924529895
0,UIUC,4,precision,0.8820025508909113
0,UIUC,4,accuracy,0.8669744020142677
0,UIUC,4,recall,0.8562514222386997
```

File B.2: Result of Experiment 1 Simulation B

```
task_id,dataset_name,fold,result_type,value
0,UIUC,0,f1_score,0.8157831018708528
0,UIUC,0,precision,0.8008905925457865
0,UIUC,0,accuracy,0.789760805707092
0,UIUC,0,recall,0.8446787974820452
0,UIUC,1,f1_score,0.7998169177615756
0,UIUC,1,precision,0.7947934862565808
0,UIUC,1,accuracy,0.7771716323961393
0,UIUC,1,recall,0.8221157980686306
0,UIUC,2,f1_score,0.7980786517320381
0,UIUC,2,precision,0.7947888316062037
0,UIUC,2,accuracy,0.7700377675199328
0,UIUC,2,recall,0.8193893495136292
0,UIUC,3,f1_score,0.7991089890131446
0,UIUC,3,precision,0.7959265495100449
0,UIUC,3,accuracy,0.766261015526647
0,UIUC,3,recall,0.822395119883811
0,UIUC,4,f1_score,0.813911092475849
0,UIUC,4,precision,0.8155680619493116
0,UIUC,4,accuracy,0.7922786403692824
0,UIUC,4,recall,0.8223753123531425
```

File B.3: Result of Experiment 1 Simulation C

```
task_id,dataset_name,fold,result_type,value
0,UIUC,0,f1_score,0.8894496183259248
```

```
0,UIUC,0,precision,0.9062219994273688
0,UIUC,0,accuracy,0.8966588966588966
0,UIUC,0,recall,0.8773011434639553
0,UIUC,1,f1_score,0.9195885943828314
0,UIUC,1,precision,0.9131348054124815
0,UIUC,1,accuracy,0.9254079254079254
0,UIUC,1,recall,0.9275145038125991
0,UIUC,2,f1_score,0.9173495614459375
0,UIUC,2,precision,0.9187259991781268
0,UIUC,2,accuracy,0.9168609168609169
0,UIUC,2,recall,0.917799901950683
0,UIUC,3,f1_score,0.9101478268530271
0,UIUC,3,precision,0.9192208111826058
0,UIUC,3,accuracy,0.9207459207459208
0,UIUC,3,recall,0.9033396057506865
0,UIUC,4,f1_score,0.9255076277705744
0,UIUC,4,precision,0.9253293404589564
0,UIUC,4,accuracy,0.9316239316239316
0,UIUC,4,recall,0.9267257640589098
```

File B.4: Result of Experiment 1 Simulation D

```
task_id,dataset_name,fold,result_type,value
1,UIUC,0,f1_score,0.942237416477633
1,UIUC,0,precision,0.9507238554425526
1,UIUC,0,accuracy,0.9440559440559441
1,UIUC,0,recall,0.934579819550681
1,UIUC,1,f1_score,0.9517274808794434
1,UIUC,1,precision,0.9489976215652123
1,UIUC,1,accuracy,0.9518259518259519
1,UIUC,1,recall,0.954729635190267
1,UIUC,2,f1_score,0.9557658792589085
1,UIUC,2,precision,0.9551860355130598
1,UIUC,2,accuracy,0.9533799533799534
1,UIUC,2,recall,0.9564923701459507
1,UIUC,3,f1_score,0.9459738441046402
1,UIUC,3,precision,0.9526744541376271
1,UIUC,3,accuracy,0.9440559440559441
1,UIUC,3,recall,0.9408790119289435
1,UIUC,4,f1_score,0.9566788451628216
1,UIUC,4,precision,0.9572074551745832
1,UIUC,4,accuracy,0.9533799533799534
1,UIUC,4,recall,0.9561779221083334
```

File B.5: Result of Experiment 2

```
task_id,dataset_name,fold,result_type,value
0,SQUAD,0,recall,0.9599090373115472
0,SQUAD,0,mean_average_precision,0.8849688735286485
```