UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN COMPUTER ENGINEERING

A Comparative Study and Analysis of Conversational Search Algorithms to Improve their Reproducibility

Supervisor
Prof. Ferro Nicola

Submitted by
Carraretto Gianmarco

Co-supervisor
Faggioli Guglielmo

4 April 2022

Abstract

Conversational Search is a field of Information Retrieval that is steadily gaining popularity in recent years. A conversational retrieval system aims to engage with the users in conversations using natural language. In this work, we studied, implemented and compared a total of eleven state-of-the-art algorithms and strategies for conversational search. Additionally, we developed a conversational retrieval framework focused on modularity, extensibility and reproducibility, that we used to test said algorithms. The top-performant method we tested, *Context Query*, obtained an `nDCG@3` of 0.43, beating more complex methods, like the ones based on coreference resolution or Large Language models (i.e., BERT), by at least 10%. Concerning the reproducibility aspect, we've been able to reach comparable results on several methods for which we had a suitable comparing value.

Sommario

La Ricerca Conversazionale è un settore del Reperimento delle Informazioni che sta guadagnando sempre più popolarità negli ultimi anni. Un sistema di reperimento conversazionale punta a interagire con gli utenti mediante conversazioni tenute in linguaggio naturale. In questo lavoro, noi abbiamo studiato, implementato e comparato undici strategie e algoritmi allo stato dell'arte per la ricerca conversazionale. Abbiamo, inoltre, sviluppato un framework per la ricerca conversazionale incentrato sulla modularità, estensibilità e riproducibilità, che abbiamo usato per testare i suddetti algoritmi. Il metodo più performante da noi testato, *Context Query*, ha ottenuto un `nDCG@3` di 0.43, superando metodi più complessi, come quelli basati sulla risoluzione delle coreferenze o su grandi modelli linguistici (ovvero, BERT), di almeno il 10%. Per quanto concerne l'aspetto della riproducibilità, siamo stati in grado di raggiungere risultati comparabili in diversi metodi per i quali avevamo un appropiato valore di confronto.

Contents

Chapter 1

Introduction

Information Retrieval (IR) is a very old yet extremely actual subject. From traditional library catalogues to modern search engines, the task of finding useful data from previously stored information is essential to keep the knowledge accessible. With the technological advancement of recent years, we're seeing an increase in the popularity of personal assistants with interactive voice-based (or chit-chat based) interfaces: *Google Assistant*, *Apple Siri*, *Amazon Alexa* and many more. All these systems have their foundation inside the domain of **Conversational Search**.

Conversational Search Traditional IR interfaces are limited. They rely on the ability of the user to construct suitable questions that the system will be able to process and employ a *one-shot* user-machine interaction that is unsuited for non-keyboard based interfaces. A conversational retrieval system aims to provide the opposite interaction: let the users express their information needs in *natural language*, and engage with them through **conversations**. Conversational Search shifts the responsibility of query understanding **from the user to the machine**, allowing for easier and more natural ways to search through the knowledge. Unfortunately, this is still a challenging task that leads to much research around the world.

The importance of context The most important challenge posed by Conversational IR is **natural language processing**. Queries expressed in natural language are often imprecise, ambiguous and frequently contains *reference to previous subjects.* Consequently, a conversational system must be able to **retain the context** of the ongoing conversation and exploit it to resolve ambiguity, *co-reference* and, in general, all the implicit information contained in the query.

Comparative analysis In this work, we studied several state-of-the-art algorithms for conversational search. We discussed their implementability, focusing on the unclear or implementation-depending parts and providing our implementation of said methods.

We developed a conversational search framework to make reproducible tests of the implemented methods. The framework is modular and easy to extend, allowing the integration of additional algorithms or the reuse of its components in other projects.

Employing the conversational collection from *"TREC CAsT 2019"* [DXC20], we ran extensive tests of the implemented algorithms. We present the obtained results, along with an accurate statistical analysis and performance charts, to make a fair comparison between the different methods. We included popular techniques, like *pseudo relevance feedback* and *coreference resolution*, and tested their effectiveness on a conversational context.

Contributions With this work, we provide the following contributions:

- Implemented a comparative framework for the main conversational search techniques;
- Evaluated the effect of different parameters on each of these techniques;
- Compared all the implemented methods with each other;
- Discussed and evaluated the reproducibility of some of the top performant submissions of TREC CAsT[DXC20];

Structure of this work In Chapter 2, we present the background on Information retrieval and Conversational Searching and describe the statistical tools used for the results' analysis. In Chapter 3, we provide a formal description of every implemented algorithms. In Chapter 4, we describe the structure of our framework and the implementation of the methods. In Chapter 5, we present the results of our tests and the statistical analysis. Finally, on Chapter 6, we summarized the work and suggest future improvements.

Chapter 2

Background

Information Retrieval (IR) is the process of finding information that satisfy the *information need* of the user from a collection of resources.

Traditional IR systems employ a *one-shot* interaction process with the user who is tasked to find the most-effective query that will lead to the wanted results. While these systems are widely implemented, from search engines to library catalogues, they can be complex to use effectively and limit the possible interactions with a platform.

Conversational IR tries to solve these problems by engaging with the user in a conversation using natural language. The next section mention some works that might help in outlining what is conversational search and how it's different from traditional IR.

2.1 Related Works

2.1.1 Traditional IR

Archiving and retrieving written information is a very old subject. Cataloguing methods and bibliographic systems were developed from ancient times to help find stored information. In the 20^{th} century, thanks to the technological advancement, some people started to think of creating automatic systems for storing and finding information.

In an essay published in 1945 titled *"As We May Think"*, Vannevar Bush described a hypothetical electromechanical device, named *Memex*, able to store and retrieve documents (microfilms) together with annotation and links [Bus45], introducing the idea of automatic access to large amount of stored knowledge [SG01]. Later in 1950, the American's computer scientist Calvin Mooers, coined the term *Information Retrieval* (IR) denoting the recovery, from a given collection of documents, of a set of documents that includes all documents with a specified content (possibly in conjunction with not relevant ones) [Fai58]. From there, a lot of researched developed. One of the most influential (according to [Fai58]) happened in 1957, when H.P. Luhn proposed a statistical approach for encoding and searching documents based on a "thesaurus-type dictionary and index" [Luh57].

Early-developed retrieval systems performed a boolean search combining words using `AND`, `OR` and `NOT` operators. These systems operate without notion of relevance or document ranking [Fai58]. To overcome this limitation, the Vector Space Model [SWY75] was introduced. In this model, a document is represented by a vector of terms where, if a term is inside a document, the vector for that document will have a non-zero value in the corresponding dimension. Comparing the vectors for two documents (or a document and a query), it's possible to measure a similarity score between them, reflecting the similarity between the corresponding terms and term weights [SWY75; Fai58]. In 1960, M.E. Maron proposed the idea of a probabilistic retrieval model and the notion of *relevance* [MK60]. This family of models ranks documents in decreasing order of probability of relevance (*Probability Ranking Principle* [Rob77]). Another common IR model is the Inference Model, introduced by Howard Turtle in 1990 [TC90], that make use of Bayesian inference networks.

2.1.2 Conversational Search

The root of conversational search trace back as early as 1980, when some researches started to investigate strategies for interactive IR systems, focusing on understanding the user's information need [Bel80; CT87]. With the diffusion of internet, more works was done towards models for information retrieval more suitable for user interaction; like the "berrypicking" model presented by Bates in 1989 [Bat89]. Belkin et al., in 1995, proposed a model of information retrieval system based on dialogue structure and specific information-seeking dialogues [Bel+95]. Nordlie, in 1999, studied the progression on user queries in online searches applying a communication model, based on theories of conversations between stranger [Nor99].

More recently, the work of Christakopoulou et al. studied how to make a recommendation system behave more like a human [CRH16]. Vtyurina et al. compared the effectiveness of current conversational search assistants with real human conversation and discussed the limitation of the former [Vty+17].

A comprehensive theoretical study on conversational approaches for information retrieval was conducted by Radlinski et al. [RC17]. In that works, a conversational search system is described as:

> a system for retrieving information that permits a mixed-initiative[1] back and forth between a user and agent, where the agent's actions are chosen in response to a model of current user needs within the current conversation, using both short and long-term knowledge of the user.

Additionally, they provide five properties that a conversation system must have:

User Revealment The system assists the user discovering their information needs;

System Revealment The system reveal to the user its capabilities, i.e, what it can or cannot do;

Mixed Initiative Both the system and the user can take initiative;

Memory The user can refer to past statements (which are considered true until contradicted);

Set Retrieval The system can present sets of complementary items as solution;

Modern conversational search includes a large amount of related task, like the generation of clarifying questions [Tav20], the analysis of spoken conversations [Tri+19], utterance rewriting [Mel+21] and many others.

Conversational search can be applied to various fields. Here's some of the most popular:

Recommendation System Recommendation system can use conversation as a more natural way to learn user preferences [Jan+22];

Chatbot Chatbots are non-goal-oriented conversational agents that engage on users through chit-chat (i.e., small talk), traditionally with a text-based interface. They are a traditional application of conversational search, dating back in the 1960s [WY19a; Zam+22];

SERP Traditional Search Engine use a query-based paradigm that can make difficult for the user to express their information needs. A conversational-based interface can make this task considerably easier [Ren+21];

[1]Mixed-initiative refers to a flexible interaction strategy where all agents (human or computer) can contribute with "what it is best suited at the most appropriate time" [AGH99]

Speech-based search Smart assistants, embedded in mobile devices or home smart devices (like Google Home or Amazon Alexa), that are gaining in popularity in recent years have speech-only interfaces that aim to work in natural language. For that reason they are one of the most popular application of conversational search and take huge benefits from its improvements [Tri19; Tri+20; Cla+19];

A conversational search system is often a *sequence-aware* system as it take in input a list of user interaction in chronological order [QCJ18]. Sequence-aware systems can be divided in two groups: *sequential* and *session-based* [JMB20].

Sequential systems Sequential systems incorporate all the past interactions, i.e., past interactions with the users are stored for future usage;

Session-based systems Session-based systems focus only on the ongoing session and do not keep any long-term information;

Another common distinction between conversational systems, is whether the system is *goal-oriented* or non-goal-oriented (*chit-chat*).

Goal-oriented A goal-oriented system is designed to help people in a specific task (e.g., making a reservation, buy a ticket) [WY19b]. In goal-oriented systems there is, often, a single correct (while context-dependent) answer [GAS19];

Chit-chat A chit-chat system (chatbot) aim, instead, to engage in a conversation with the user for entertainment/companionship reasons [WY19b];

2.1.3 Existing Methods for Conversational Search

While it's not a rule, a lot of conversational search's systems employ a pipeline composed of three stage:

$$\textit{rewriting} \rightarrow \textit{retrieval} \rightarrow \textit{reranking}$$

The majority of the work that distinguish a traditional IR pipeline from a conversational one, is in the *rewriting* and *reranking* stages.

Commonly used techniques

Considering that one of the main goal of the Conversational Search field is to interact with the user using natural language, there should be no surprise that **coreference resolution** is one of the most popular rewriting method.

Coreference resolution describe a set of techniques that aims at resolving repeated reference of an object inside a text. It's a core component of natural language processing (NLP) and it's often used in combination with other techniques [Suk+20]. This is a challenging task that is often implemented through Deep Learning Networks (like with the popular library AllenNLP [Gar+18]).

Another commonly used technique (not only in conversational search, but in IR in general) to improve the results is a *pseudo relevance feedback* model called **RM3**. This method consist in: do a first retrieval, take the top-M terms in the top-N documents and use them to expand the original query (using a weight w) then re-do the retrieval with the newly created query [Jal+04]. The idea behind this method is that terms that appear frequently in the best documents should be relevant to the query itself, so adding them to the query for a new search may help to move relevant documents to the top of the results.

A popular model used for reranking is **BERT**. Bidirectional Encoder Representations from Transformers (BERT) is a language representation model used in machine learning for NLP tasks [Dev+18]. It is designed to pre-train deep bidirectional neural networks from unlabelled text on both left and right contexts. BERT pre-trained model can be fine-tuned with a single additional layer. The resulting model can be used for a variety of tasks (e.g., question answering, language inference) without substantial modifications.

Example of conversational methods

In this section are reported some example of methods used for conversational search by other works (selected by the ones submitted for CAsT 2019 [DXC20]).

Kumar et al. [KC19] propose a system composed by three components. The first one decides if contextual information needs to be incorporated in the current query. This decision is based on the KL-divergence between the original query's retrieved documents and whether the query is made by pronouns. The second component identifies the contextual information for the current query using an SVN classifier with BERT attention weights. The third component does the actual retrieval using Indri.

Kaiser et al. [KRW20] proposed an unsupervised method for conversational passage ranking that formulate the passage score for a query as a combination of similarity and coherence. More specifically, the passages that contain semantically similar words to the question's ones, and where those words are more close by, are preferred. They built a word proximity network to achieve that result.

Ríssola et al. [Rís+19] developed a neural model (based on BERT) for identifying other turns relevant to the current one and reformulating the user's information need considering the conversational context. They also employ coreference resolution and reranking techniques (also based on BERT) to obtain better results.

L.A. Clarke [Cla19] proposed a pipeline with coreference resolution, custom stopword set and a rewriting method that glue the first query in every conversation to the current one[2]. Employ a standard BM25 search with RM3 and a rerank method based on BERT.

2.2 Tools for Statistical Analysis

This section outline some statistical tools used for methods comparison in this work. The tools used were Analysis of Variance, Multiple Comparison and Partial Omega Squared, all three computed using MATLAB[3].

2.2.1 Analysis of Variance

Analysis of Variance (ANOVA) is a statistical tool that can be used to analyse differences among means and reveal the influence of the input variables on the result. ANOVA can be presented as a linear model in which the probability distribution of the responses follow these assumptions:

- Independence of observations;
- Normality of residuals;
- Equality of variances;

While the correctness of these assumptions may not be always respected in this specific use-case, the analysis of variance remains a very powerful tool to distinguish models and parameters that lead to meaningful difference in the results.

This work use the MATLAB function `anovan` to generate ANOVA tables. Said function implements an n-way analysis of variance, i.e., ANOVA between n different variables that, for this work, will be models or models' parametrizations.

The returned ANOVA table contains six columns:

[2]This rewriting method is used by other works, including this one, and will be later referred as *First Query*

[3]`https://mathworks.com/products/matlab.html`

Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
X1	3.781	1	3.781	0.82	0.4174
X2	199.001	1	199.001	42.95	0.0028
X3	0.061	1	0.061	0.01	0.914
Error	18.535	4	4.634		
Total	221.379	7			

Table 2.1: Example of ANOVA from MATLAB documentation (`anovan`)

Source This column contains the name of the variables, plus Error and Total;

Sum Squares (Sum Sq.) This column contains the sum of squared deviation for every variable, plus the error due a lack-of-fit. It measures the variability of the distribution associated with the variable;

Degrees of Freedom (d.f.) It measures the number of values that are free to vary without violating any constraint;

Mean Squares (Mean Sq.) The mean squared is an estimation of the variance of the variable. It's obtained by scaling the sum of squares by the degrees of freedom;

F-test (F) This column contains the F-statistic, i.e., the ratio of the mean squares;

P-value (Prob>F) It's the probability that the F-statistic can take a value larger than the computed test-statistic one (derived from the cumulative distribution function of the F-distribution);

A p-value < 0.05 generally denote a meaningful variable.

In addition to the table, `anovan` return a collection of statistics that can be used for the multiple comparison.

2.2.2 Multiple Comparison

The multiple comparison is a test performed on the statistics returned by ANOVA, using the `multcompare` MATLAB function. This function plot an interactive graph that shows the estimates and comparison intervals of an ANOVA variable, allowing to visually compare different methods (or choices of parameters).

All tested values are displayed on the y-axis, their mean is identified by an empty circle with a line extending from it that represent the values' interval.

The selected group is coloured in blue, while groups that are significantly different from it are red. Others, non-significant, groups are grey. In particular, not overlapping intervals are an indication of significant difference.

2.2.3 Partial Omega Squared

Partial omega squared (ω^2) is a measure of the degree of association for a population (effect size), i.e., it's an estimate of the strength of the relationship between two variable in a population. The *partial* in the name refer to the fact that we are considering the omega square calculated for a single factor.

As a rule of thumbs, values of $\omega^2 > 0.14$ denote a *large-size* effect, $0.06 \leq \omega^2 \leq 0.14$ denote a *medium-size* effect and $0.01 \leq \omega^2 < 0.06$ denotes a *small-size* effect. Negative values of ω^2 are meaningless, i.e., the factor does not provide contributions.

Chapter 3

Implemented Algorithms

This chapter is dedicated to explaining the various algorithms implemented in this work. We focus on two categories: rewriting and reranking. Rewriting algorithms provides a way to rewrite queries before issuing them to the retrieval system and, specifically in the conversational domain, focus on providing the missing context for the query. On the other end, reranking approaches are applied after a first retrieval phase (often called first stage retrieval) to revise the expected relevance, thus the rank, of the retrieved documents.

3.1 Structure and Notation

This section describes the general structure of rewriting and reranking algorithms and provides a basic notation to define them. However, if the paper that originally described an algorithm presents its notation, we'll use that instead.

A conversation (also called a session) is defined by a sequence of utterances:

$$S =< u_1, u_2, ..., u_i, ..., u_{n-1}, u_n > \tag{3.1}$$

where u_i is the user's utterance of the i^{th} turn (i.e., the i^{th} query expressed by the user inside the session). All utterances of all conversations form a list of query:

$$Q =< S_1, ..., S_j > = < q_1, ..., q_N > \tag{3.2}$$

Rewriting Algorithm A rewriting algorithm takes as input the current utterance (and possibly all the previous ones in the conversation) and return as output the rewrite of that utterance.

Formally, a rewrite algorithm can be defined as a function:

$$rewrite : S \mapsto S_r \tag{3.3}$$

where S_r is defined as:

$$S_r =< u_{r_1}, ..., u_{r_n} > \tag{3.4}$$

where u_{r_i} is the rewrite of u_i.

Results Ranked List The search will provide an ordered list of documents from the documents' collection:

$$D =< D_1, ..., D_N > = < d_{1,1}, ..., d_{1,k}, ..., d_{N,1}, ..., d_{N,k} > \tag{3.5}$$

where $D_i =< d_{i,1}, ..., d_{i,k} >$ is the result list for the query q_i. Every document $d_{i,j}$ have an associated score $s_{i,j}$ so that:

$$s_{i,j} \;\geq\; s_{i,j+1} \;\; \forall i = 1, ..., N \; \forall j = 1, ..., k-1 \tag{3.6}$$

Reranking Algorithm A reranking algorithm takes as input the collection of ranked documents and returns a new collection, with the same documents in a different order.

Formally, a reranking algorithms can be defined as a function:

$$rerank : D \mapsto D_{rr} \tag{3.7}$$

where D_{rr_i} is defined as:

$$D_{rr} =< d_{i,rr_1}, ..., d_{i,rr_k} > \tag{3.8}$$

where rr_j is the new rank for the document that originally had rank j.

Concatenation of Utterances The operation of utterances' concatenation (commonly used by rewriting algorithms) is represented with the addition symbol and always includes a whitespace character between different utterances.

Formally, given two utterances $u_i =< c_{i,1}, ..., c_{i,n} >$ and $u_j =< c_{j,1}, ..., c_{j,m} >$:

$$u_i + u_j =< c_{i,1}, ..., c_{i,n}, \sqcup, c_{j,1}, ..., c_{j,m} > \tag{3.9}$$

where $c_{x,k}$ is the k-th character of u_x.

Title:	Uranus and Neptune
Description:	Information about Uranus and Neptune.
Turn	**Conversation Utterance**
1	Describe Uranus.
2	What makes it so unusual?
3	Tell me about its orbit.
4	Why is it tilted?
5	How is its rotation different from other planets?
6	What is peculiar about its seasons?
7	Are there any other planets similar to it?
8	Describe the characteristics of Neptune.
9	Why is it important to our solar system?
10	How are these two planets similar to each other?
11	Can life exist on either of them?

Table 3.1: CAsT 2019 [DXC20] Training Topic #18

3.2 Rewriting Algorithms

3.2.1 Concat Query

Concat Query is a simple rewriting algorithm that concatenates all the previous utterances in the conversation with the current one. Formally:

$$u_i \mapsto u_1 + ... + u_{i-1} + u_i \tag{3.10}$$

While this makes sure to include all possible context, it may also include non-relevant (or too general) keywords, possibly repeated several times, thus leading to non-relevant results. However, it's worth mentioning that, sometimes, this method is used as a step inside more complex searching strategies precisely for the just mentioned reason of including all the available context.

For example, looking at the 3^{rd} utterance of Table 3.1:

Tell me about its orbit.

Rewriting it using *Concat Query* we obtain:

Describe Uranus. What makes it so unusual? Tell me about its orbit.

The rewritten utterance is certainly more meaningful than the first one, as we added the topic to it ("Uranus"). Yet, adding "unusual" to it's misleading, as it refers to the previous sub-topic, not the current one.

Algorithm 3.1 Concat Query

```
Input: $S$
Output: $S_r$
1: $u_{r_1} \leftarrow u_1$
2: for $i = 2, ..., |S|$ do
3:     $u_{r_i} \leftarrow u_{r_{i-1}} + u_i$
4: end for
5: $S_r \leftarrow < u_{r_1}, ..., u_{r_{|S|}} >$
```

3.2.2 First Query

First query [Mel+21] is a rewriting algorithm that rewrites utterances by concatenating the first utterance of the conversation with the current one. Formally:

$$u_i \mapsto u_1 + u_i \tag{3.11}$$

The advantage of this algorithm is that including the first utterance often means including the main topic of the conversation without adding too many irrelevant words. Still, this does not consider eventual context shifts and might lead to unwanted results after one occurred.

The algorithm's description in the mentioned paper does not explain how to rewrite the first utterance of a conversation. Thus, it's left to the implementation to choose between no rewrite (3.12) or utterance duplication (3.13).

$$u_1 \mapsto u_1 \tag{3.12}$$

$$u_1 \mapsto u_1 + u_1 \tag{3.13}$$

Looking at an example, the rewrite of the 3rd utterance of Table 3.1 will be:

Describe Uranus. Tell me about its orbit.

This is an improvement over the *Concat Query* rewrite, as we've only included the needed context. However, if we look at the 9th utterance:

Why is it important to our solar system?

The relative rewrite will be:

Describe Uranus. Why is it important to our solar system?

This seems a good rewrite until you notice there was a context shift on the 8th utterance and, thus, the planet that should be referenced by this query is Neptune, not Uranus. Looking at the table, we can observe that *First Query* will produce acceptable rewritings up to the 7th utterance and misleading rewritings for the last four ones.

Algorithm 3.2 First Query

```
Input: $S$
Output: $S_r$
1: $u_{r_1} \leftarrow u_1$ * or in alternative $u_1 + u_1$ *
2: for $i = 2, .., |S|$ do
3:     $u_{r_i} \leftarrow u_1 + u_i$
4: end for
5: $S_r \leftarrow < u_{r_1}, ..., u_{r_{|S|}} >$
```

3.2.3 Context Query

Context Query [Mel+21] is an extension of *First Query*. This rewriting technique concatenates both the first and the previous utterance with the current one. Formally:

$$u_i \mapsto u_1 + u_{i-1} + u_i \tag{3.14}$$

Even in this case, the original description does not mention how to rewrite the first and the second utterance of a conversation. So, the implementation should decide whether duplicate them (3.17; 3.18) or not (3.15; 3.16).

$$u_1 \mapsto u_1 \tag{3.15}$$
$$u_2 \mapsto u_1 + u_2 \tag{3.16}$$

$$u_1 \mapsto u_1 + u_1 + u_1 \tag{3.17}$$
$$u_2 \mapsto u_1 + u_1 + u_2 \tag{3.18}$$

Compared to *First Query*, this algorithm improves the response to context shifts by always including the previous utterance. However, there are no guarantees that the previous utterance contains the needed context.

For example, the 9th utterance of Table 3.1

Why is it important to our solar system?

will be rewritten as:

> *Describe Uranus. Describe the characteristics of Neptune. Why is it important to our solar system?*

We can observe that the rewrite includes the missing context ("Neptune") but also the old non-relevant one ("Uranus"). This will likely provide a better recall than *First Query*, but irrelevant documents might still appear in the first positions. Instead, if we took the 10th utterance

> *How are these two planets similar to each other?*

and its rewrite

> *Describe Uranus. Why is it important to our solar system? How are these two planets similar to each other?*

we can see that there's no meaningful context in it. We can expect the same to happen for every query after a context shift.

Algorithm 3.3 Context Query

```
Input: $S$
Output: $S_r$
1: $u_{r_1} \leftarrow u_1$ * or in alternative $u_1 + u_1 + u_1$ *
2: $u_{r_2} \leftarrow u_1 + u_2$ * or in alternative $u_1 + u_1 + u_2$ *
3: for $i = 3, .., |S|$ do
4:     $u_{r_i} \leftarrow u_1 + u_{1-1} + u_i$
5: end for
6: $S_r \leftarrow < u_{r_1}, ..., u_{r_{|S|}} >$
```

3.2.4 Coref 1

Coref 1 [Mel+21] is a coreference resolution algorithm based on machine learning that uses the coreference resolver implemented in *AllenNLP models*[1].

$$u_i \mapsto u_{i,resolved} \tag{3.19}$$

AllenNLP models is a collection of classes and pre-trained models, based on the AllenNLP[2] library, for executing a wide range of natural language processing (NLP) tasks.

[1]https://github.com/allenai/allennlp-models

[2]https://github.com/allenai/allennlp

In particular, we use *coref-spanbert* [LHZ18], a pre-trained model for coreference resolution that uses a fully differentiable approximation to high-order inference and a span-ranking architecture. The model employs an iterative approach that uses the antecedent distribution as an attention mechanism to update the existing span-representation, allowing the later coreference decisions to condition on earlier ones. Additionally, to reduce the computational cost, it uses a coarse-to-fine approach with a less accurate but more efficient bilinear factor that increases pruning without hurting the accuracy. Given the heaviness of the algorithm, the library provides a GPU-accelerated implementation of this model.

Algorithm 3.4 provides a high-level, formal description of this method, where *resolve* is the function associated with the coreference resolver. We concatenate all the conversational utterances up to the current one in a way that allows us to re-split them after the resolution (i.e., using a non-misleading separator string). Then we pass the constructed utterances' concatenation to the resolver and split the output, as we need to return only the current query's resolution.

Algorithm 3.4 Coref (1 & 2)

```
Input: $< u_1, ..., u_i >$
Output: $u_{i,resolved}$
1: $sep \leftarrow$ *string separator*
2: $full\_concat \leftarrow u_1 + sep + u_2 + sep + ... + u_{i-1} + sep + u_i$
3: $full\_resolved \leftarrow resolve(full\_concat)$
4: $< u_{1,resolved}, ..., u_{i,resolved} > \leftarrow split(full\_resolved)$
```

For example, the phrase

> *What is throat cancer? Is it treatable? Tell me about lung cancer. What are its symptoms?*

is resolved by the AllenNLP model in

> *What is throat cancer? Is throat cancer treatable? Tell me about lung cancer. What are lung cancer's symptoms?*

3.2.5 Coref 2

Coref 2 [Mel+21] apply the same algorithm of *Coref 1* (i.e., Algorithm 3.4) but uses *neuralcoref*[3] for the coreference resolution instead of AllenNLP.

Neuralcoref is an extension of SpaCy[4], a popular library for natural

[3]`https://github.com/huggingface/neuralcoref`

[4]`https://spacy.io/`

language processing, that resolve coreference clusters using a neural network.

The *neuralcoref* model works[5] by averaging the word-embedding of words inside and around each mention and obtaining a features' representation of each mention and its surroundings. Then, it passes these representations into two neural networks: the first scores each pair of a mention and a possible antecedent, the second scores mentions without antecedents. Comparing these scores, the model determines if a mention have an antecedent and which one could be.

Compared with *Coref 1*, *neuralcoref* is significantly faster, however, it's not able to resolve as many references as the former. If we take the phrase:

> *What is throat cancer? Is it treatable? Tell me about lung cancer. What are its symptoms?*

the resolution will be:

> *What is throat cancer? Is throat cancer treatable? Tell me about lung cancer. What are throat cancer symptoms?*

As you can see, *neuralcoref* resolve correctly only the first mention but fails to detect the context switch and thus incorrectly keep the old reference for the second one.

3.2.6 Historical Query Expansion

Historical Query Expansion (HQE) [JC19] is a rewriting algorithm that aims to find relevant keywords in previous utterances (by comparing their scores with some thresholds) and using them to fill for the missing context.

To achieve that result, this algorithm divides every query in tokens (corresponding to *words*) and find the maximum BM25 score between every token and the documents' collection. Then, compares the score's value to two thresholds to determine if the token is a relevant keyword to the current query (r_Q) or the current session (r_S). A third threshold (θ) is compared with the higher score for a particular query to determine if that query is ambiguous or not. Non-ambiguous queries are expanded with all the session keywords found up to that moment. Ambiguous ones receive a further expansion with all the query keywords from the last three queries.

The Algorithm 3.5 shows the pseudocode for the algorithm using the same notation of the paper's authors, where:

[5]`https://medium.com/huggingface/state-of-the-art-neural-coreference-resolution-for-chatbots-3302365dcf30`

Algorithm 3.5 HQE: Historical Query Expansion

```
Input: $S = \{Q_i\}_{i=1}^{N}$, $D$
Output: $S$
 1: $W_S \leftarrow ()$
 2: $W_Q \leftarrow ()$
 3: for $i = 1$ to $N$ do
 4:     for $k = 1$ to $n(i)$ do
 5:         $r_k^i = \max_{d_j \in D} F(d_j, (q_k^i))$
 6:         if $r_k^i > r_S$ then
 7:             $W_S$.insert($q_k^i$)
 8:         end if
 9:         if $r_k^i > r_Q$ then
10:             $W_Q$.insert($q_k^i$)
11:         end if
12:     end for
13:     if $i > 1$ then
14:         $R_i = \max_{d_j \in D} F(d_j, Q_i) = \max_{d_j \in D} F(d_j, (q_0^i, q_1^i, ..., q_{n(i)}^i))$
15:         $Q_i$.insert($q_n^k$)$\forall q_k^n \in W_S$
16:         if $R_i < \theta$ then
17:             $Q_i$.insert($q_k^n$) $\forall q_k^n \in W_Q \wedge n \geq i - 3$
18:         end if
19:     end if
20: end for
```

S is the session, i.e., the list of conversational utterances;

D is the documents' collection;

W_S and W_Q are the sets of session's (respectively query's) keywords;

F is the function that calculate the BM25 score between a token and a document;

r_S and r_Q are the session's and query's relevant thresholds;

θ is the threshold for considering a query non-ambiguous;

Q_i is the current query (represented as a list of tokens);

Despite providing the pseudocode for the algorithm, the authors don't provide suggestions on reasonable values for the thresholds or strategies to find them, leaving the decision to the implementation.

3.2.7 DBpedia & ConceptNet Query Expansion

DBpedia [Sam19] is a rewriting algorithm that uses external knowledge to enrich the queries with contextual words that might lead to a higher recall on the results list.

The algorithm works by concatenating the previous conversational utterances to the current one, then searching for entities using DBpedia Spotlight[6]. Spotlight is a public API to find entities linked to DBpedia[7], a project that extracts and publish structured content from Wikipedia. If at least one is found, they are used to querying Wikipedia's API and retrieve snippets of relevant text. Then, all the unique terms from the snippets are scored, and the current utterance is expanded with the top-10 ones. When no entity is found, n-grams of nouns and adjectives from the query are used to retrieve connected nodes (i.e., related terms) from ConceptNet[8] and use them for expansion.

Algorithm 3.6 DBpedia & ConceptNet Query Expansion

```
Input: $S =< u_1, ..., u_n >$
Output: $S$
  for $i = 1$ to $n$ do
      $q_c = u_1 + ... + u_{i-1} + u_i$
      $E \leftarrow spotlight(q_c)$
      if $|E| > 0$ then
          $O =< o_1, ..., o_m >= wikipedia(E)$
          $U =< t_1, ..., t_k >= unique\_terms(E)$
          $score_{t_i} = \sum_{o_1}^{o_n} (tf(o_j, t_i)/|o_j|) \cdot r(o_j) \cdot log(1/df(t_i))$
          $U_s = \{t_{s_i} \in U\} \mid |U_s| \leq 10 \wedge score_{t_{s_i}} \geq score_{t_{s_{i+1}}} \forall i = 1, ..., |U_s| - 1$
          $u_i = concat(\{u_i\} \cup U_s)$
      else
          $NG = \{< t_1, ..., t_t >| t_i \in nouns \cup adjectives\}$
          $connected\_nodes = concept\_net(NG)$
          $u_i = concat(\{u_i\} \cup terms(connected\_nodes))$
      end if
  end for
```

Algorithm 3.6 formally describe this algorithm, where:

spotlight is the function that retrieve liked entities from DBpedia Spotlight;

[6]https://www.dbpedia-spotlight.org/
[7]https://www.dbpedia.org/
[8]https://conceptnet.io/

wikipedia is the function that retrieves snippets of text from Wikipedia;

concat is the function that create the concatenation between terms in a set (separated by a space);

concept_net is the function that retrieves connected nodes from ConceptNet;

$\boldsymbol{tf(o_j, t_i)}$ is the *term frequency* of the i-th term in the j-th snippet;

$\boldsymbol{r(o_j)}$ is the *reciprocal rank* of o_j in the results;

$\boldsymbol{df(t_i)}$ is the document frequency of a term;

Some details are not specified by the authors, like if the document frequency should be calculated on the documents' collection or on the retrieved snippets and how to choose n-grams from the text.

3.3 Reranking Algorithms

3.3.1 Seen Filter

Seen Filter is a reranking algorithm adapted from *Top-Down* [Mar19] that penalize documents already returned for a previous utterance of the same conversation with the rationale that, if a document was relevant for a previous less-specific utterance, then it's unlikely to be relevant for the subsequent ones.

The original version works by keeping in memory the top-20 documents of each utterance's results and removing them from all the subsequent utterances' results list in the same conversation.

The algorithm implemented in this work generalize that idea by allowing to tune two parameters:

m the multiplicative factor to reduce the score of the already seen documents. Should be comprised between 0 and 1. If 0 then the original algorithm's behaviour is preserved[9];

k the maximum rank to consider when keeping documents in memory, i.e., only the top-k documents of each utterance's results list will be kept and penalized in subsequent results. If 20 the original algorithm's behaviour is preserved;

[9]Technically, the document isn't removed from the list, however, setting its score to 0 put it at the end of the list, effectively removing it from the calculated measures.

Formally:

$$s_{i,j} \mapsto \begin{cases} m \cdot s_{i,j} & if\ d_{i,j} \in \bigcup\limits_{l=1}^{j-1} D_l[1..k] \\ s_{i,j} & otherwise \end{cases} \tag{3.20}$$

where $s_{i,j}$ is the score of $d_{i,j}$, i.e., the j-th documents in the results list of the i-th utterance in the conversation.

The Algorithm 3.7 shows the pseudocode of this method, where $conversation(i)$ return the conversation identifier of the results list D_i and $sort(D_i)$ sort the list of documents $D_i =< d_{i,1}, ..., d_{i,n} >$ by their score $< s_{i,1}, ..., s_{i,n} >$ by decreasing values.

Algorithm 3.7 Seen Filter

```
Input: $D, k, m$
Output: $D$
 1: for $D_i \in D,\ i = 1, ..., |D|$ do
 2:     if $i = 1 \vee conversation(i) \neq conversation(i-1)$ then
 3:         $E \leftarrow \emptyset$
 4:     end if
 5:     for $d_{i,j} \in D_i,\ j = 1, ..., |D_i|$ do
 6:         if $d_{i,j} \in E$ then
 7:             $s_{i,j} \leftarrow m \cdot s_{i,j}$
 8:         end if
 9:         $E \leftarrow E \cup D_i[1..k]$
10:         $D_i \leftarrow sort(D_i)$
11:     end for
12: end for
```

3.3.2 Bottom Up

Bottom Up [Mar19] is a reranking algorithm that penalize documents assigned to later utterances of a conversation, in previous ones. It's based on the idea that further utterances will ask about more specific knowledge, so assigning documents from the last utterance (in a conversation) to the first and filtering out the already assigned ones should leave better results.

As with *Seen Filter*, the implemented version accept the parameters m (multiplier) and k (max rank). Formally:

$$s_{i,j} \mapsto \begin{cases} m \cdot s_{i,j} & if\ d_{i,j} \in \bigcup\limits_{l=j+1}^{|D|} D_l[1..k] \\ s_{i,j} & otherwise \end{cases} \tag{3.21}$$

The Algorithm 3.8 shows the pseudocode for the *Bottom Up* algorithm. It's nearly the same as *Seen Filter*, but it loops through the results list in reverse order so that the first encountered utterance in a conversation would be the last one.

Algorithm 3.8 Bottom Up

```
Input: D, k, m
Output: D
for $D_i \in D,\ i = |D|, ..., 1$ do
    if $i = |D_i| \lor conversation(i) \neq conversation(i+1)$ then
        $E \leftarrow \emptyset$
    end if
    for $d_{i,j} \in D_i,\ j = 1, ..., |D_i|$ do
        if $d_{i,j} \in E$ then
            $s_{i,j} \leftarrow m \cdot s_{i,j}$
        end if
        $E \leftarrow E \cup D_i[1..k]$
        $D_i \leftarrow sort(D_i)$
    end for
end for
```

3.3.3 Historical Answer Expansion

Historical Answer Expansion (HAE) [JC19] is a reranking technique that is meant to be used on top of *HQE* (Section 3.2.6).

HAE exploits a BERT classifier to estimate the log-likelihood between queries and documents. Then, for every query in a conversation:

1. Consider the current query's result documents with unmodified likelihood;
2. Consider the previous query's result documents with likelihood decreased by a factor λ;
3. Sort the documents by likelihood;
4. Assign the top-k documents to the current query;

The Algorithm 3.9 shows the pseudocode of HAE using a notation close to the original paper[10], where:

[10]The original pseudocode can be difficult to interpret on some passages, so we made some changes to make it easier to understand.

Algorithm 3.9 HAE: Historical Answer Expansion

```
Input: $A = \{(Q_i, D_i)\}_{i=1}^{N}, \lambda$
Output: $A$
 1: for $i = 1$ to $N$ do
 2:     for $j = 1$ to $k$ do
 3:         $L_{i,j} \leftarrow log\_likelihood(Q_i, d_{i,j})$
 4:         $P[i][j] \leftarrow (d_{i,j}, L_{i,j})$
 5:     end for
 6: end for
 7: $P_{temp} \leftarrow < \emptyset >$
 8: for $i = 1$ to $N$ do
 9:     $P_{temp} \leftarrow P_{temp} + P[i]$
10:     $P_{temp} \leftarrow sort(P_{temp})$
11:     $P_{temp} \leftarrow P_{temp}[1..k]$
12:     $D_i \leftarrow < d_{i,j} > \in P_{temp}$
13:     for $j = 1$ to $k$ do
14:         $P_{temp}[i] \leftarrow (d_{i,j}, \lambda \cdot L_{i,j}) \ \forall \ (d_{i,j}, L_{i,j}) \in P[i][j]$
15:     end for
16: end for
```

A is the collection of input pairs of same-conversation query Q_i and relative documents returned by the search D_i;

$\boldsymbol{\lambda}$ is the decay factor, i.e., the multiplicative factor to apply to the log-likelihood of previous query's documents before adding them to the current list;

$\boldsymbol{sort(P_{temp})}$ function that sort P_{temp} by decreasing values of log-likelihood;

Chapter 4

Implementation

This chapter describes the software developed as part of this work, which includes:

- The implementation of the previously mentioned methods;
- A conversational search framework, based on pyterrier[1];

In particular, we designed the framework to achieve:

- Modularity: classes can be combined in different ways or reused in other projects;
- Extensibility: new classes can be integrated without modifying existing ones;
- Flexibility: search pipelines can be configured using text files and the framework support multiple output format;

4.1 Structure

The framework is structured around three main interfaces: `Pipeline`, `Step` and `PipelineFactory`. During the initialization of these interfaces, a dictionary containing all the possible common objects (e.g., query structure, cache folder, . . .) will be passed to the constructor. For this reason, every implementing class must accept an arbitrary number of arguments and discard the ones that don't use.

Every execution of the software, with its configuration, is called "run". A run might contain multiple pipelines to simplify comparisons and writes all the related outputs in a folder.

[1]`https://pyterrier.readthedocs.io/`

The next sections explain in details the mentioned interfaces.

4.1.1 Step

The `Step` interface (see Figure 4.1) represents a single step of search (e.g., a rewrite/rerank/retrieve step). A step has the following attributes:

name representative name for the step and its configuration;

type the type of the step, must be an instance of `StepType`. Three types of definition exist:

- `FULLY_PARALLEL` defines a step that can be parallelized on an arbitrary subset of queries;
- `CONVERSATIONALLY_PARALLEL` defines a step that can be parallelized between different conversations but must loop through every utterance of a conversation sequentially;
- `SEQUENTIAL` define a step that must be executed sequentially on the whole set. This type is currently used for steps that use the GPU as they need to manage the eventual parallelization on their own;

cleanup() optional method for cleaning up resources;

Callable the `Step` must receive a table of queries or results and return a new table with the appropriate modifications;

While the `Step` interface is enough to make the framework run, we provide two sub-interfaces to distinguish the steps' function:

Rewriter represents a query rewriter, with a `rewrite(queries)` method that receives a table of queries and returns the same table with the appropriate rewriting;

Reranker represents a document reranker, with a `rerank(results)` method that receives a table of queries' results and returns the same table with the documents for each query in a (possibly) changed order.

Additionally, the framework provides the wrappers:

Model for the retrieving step;

RM3 for the RM3 pseudo feedback model;

For every algorithm implemented, there is a class of the appropriate interface. These classes can be used and combined outside the specific pipelines presented in this work.

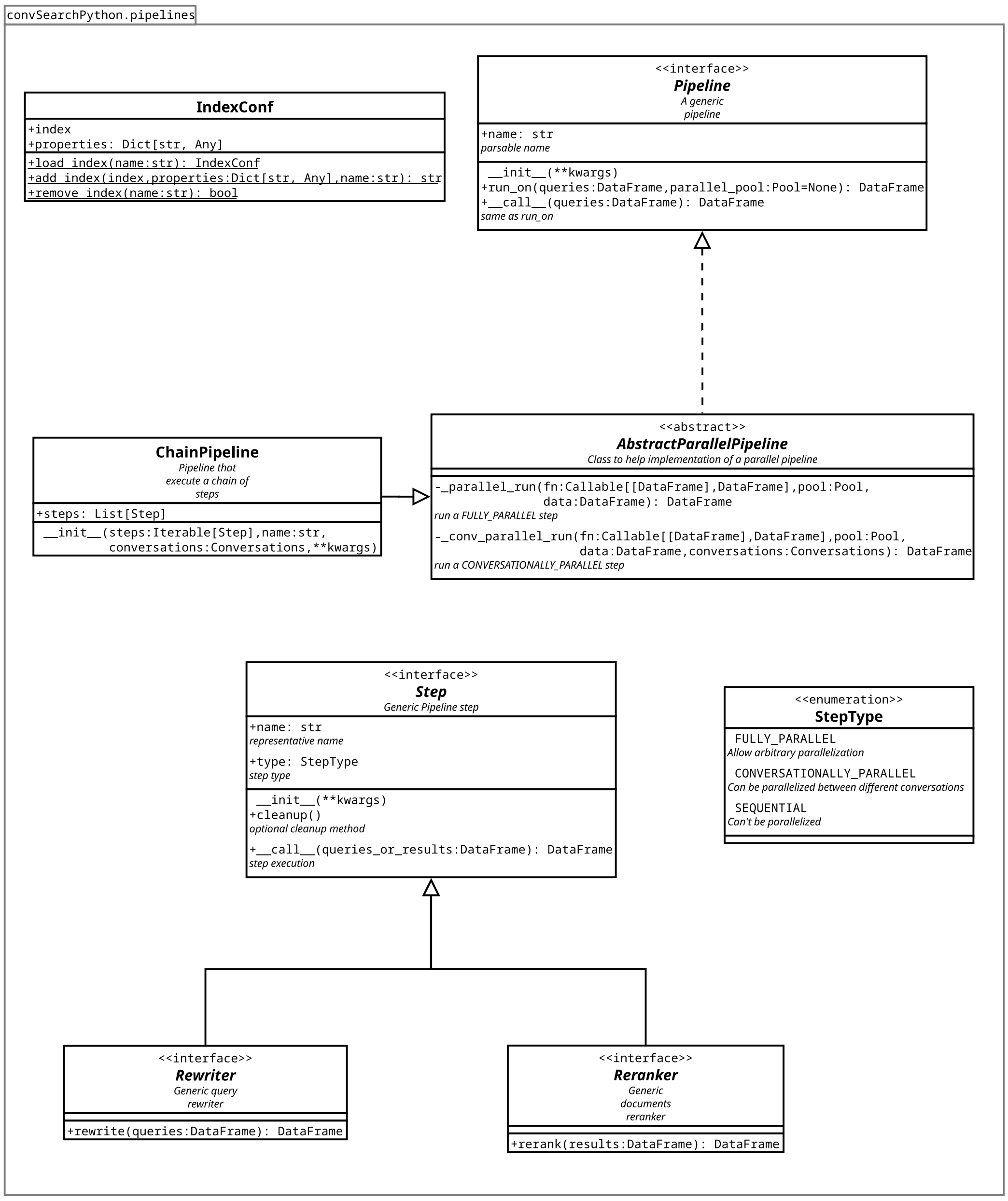

convSearchPython.pipelines
<<interface>>
Pipeline
A generic pipeline
+name: str
parsable name
__init__(**kwargs)
+run_on(queries:DataFrame,parallel_pool:Pool=None): DataFrame
+__call__(queries:DataFrame): DataFrame
same as run_on
IndexConf
+index
+properties: Dict[str, Any]
+load_index(name:str): IndexConf
+add_index(index,properties:Dict[str, Any],name:str): str
+remove_index(name:str): bool
ChainPipeline
Pipeline that execute a chain of steps
+steps: List[Step]
__init__(steps:Iterable[Step],name:str, conversations:Conversations,**kwargs)
<<abstract>>
AbstractParallelPipeline
Class to help implementation of a parallel pipeline
-_parallel_run(fn:Callable[[DataFrame],DataFrame],pool:Pool, data:DataFrame): DataFrame
run a FULLY_PARALLEL step
-_conv_parallel_run(fn:Callable[[DataFrame],DataFrame],pool:Pool, data:DataFrame,conversations:Conversations): DataFrame
run a CONVERSATIONALLY_PARALLEL step
<<interface>>
Step
Generic Pipeline step
+name: str
representative name
+type: StepType
step type
__init__(**kwargs)
+cleanup()
optional cleanup method
+__call__(queries_or_results:DataFrame): DataFrame
step execution
<<enumeration>>
StepType
FULLY_PARALLEL
Allow arbitrary parallelization
CONVERSATIONALLY_PARALLEL
Can be parallelized between different conversations
SEQUENTIAL
Can't be parallelized
<<interface>>
Rewriter
Generic query rewriter
+rewrite(queries:DataFrame): DataFrame
<<interface>>
Reranker
Generic documents reranker
+rerank(results:DataFrame): DataFrame

Figure 4.1: Framework Structure

4.1.2 Pipeline

The `Pipeline` interface (see Figure 4.1) represents a search pipeline composed of multiple steps (e.g., model → rewriter → reranker). A pipeline has the following attributes:

name easy-to-parse name of the pipeline. Must be a unique name that defines the pipeline and its configuration so that two pipelines with the same name will do the same steps and produce the same results;

run_on(queries, pool) method for executing the pipeline on a collection of queries, optionally with a parallel pool;

There are two ways to construct a pipeline:

- Writing a custom class that implements the interface. This option gives complete control over the execution flow but might be excessively tedious for simple pipelines;
- Using a factory. This alternative is limited to what the chosen factory class can achieve but allow for a more straightforward way to change the steps through the configuration file;

The framework includes two abstract `Pipeline` classes that might help writing a new one:

AbstractParallelPipeline abstract class that provides two methods to execute a `Step` on a parallel pipeline (one for a fully parallel step, the other for a conversationally parallel one);

ChainPipeline extension of the previous pipeline that executes a list of steps received when instantiated;

Both classes accept objects that do not fully implement the `Step` interface with the caveats that:

- They must be `Callable` objects that work like a step (i.e., receive a table of queries or results and return a new one);
- If they don't provide a `name` field, the class name will be used instead;
- If they don't provide a `type` field, will be considered `FULLY_PARALLEL`;

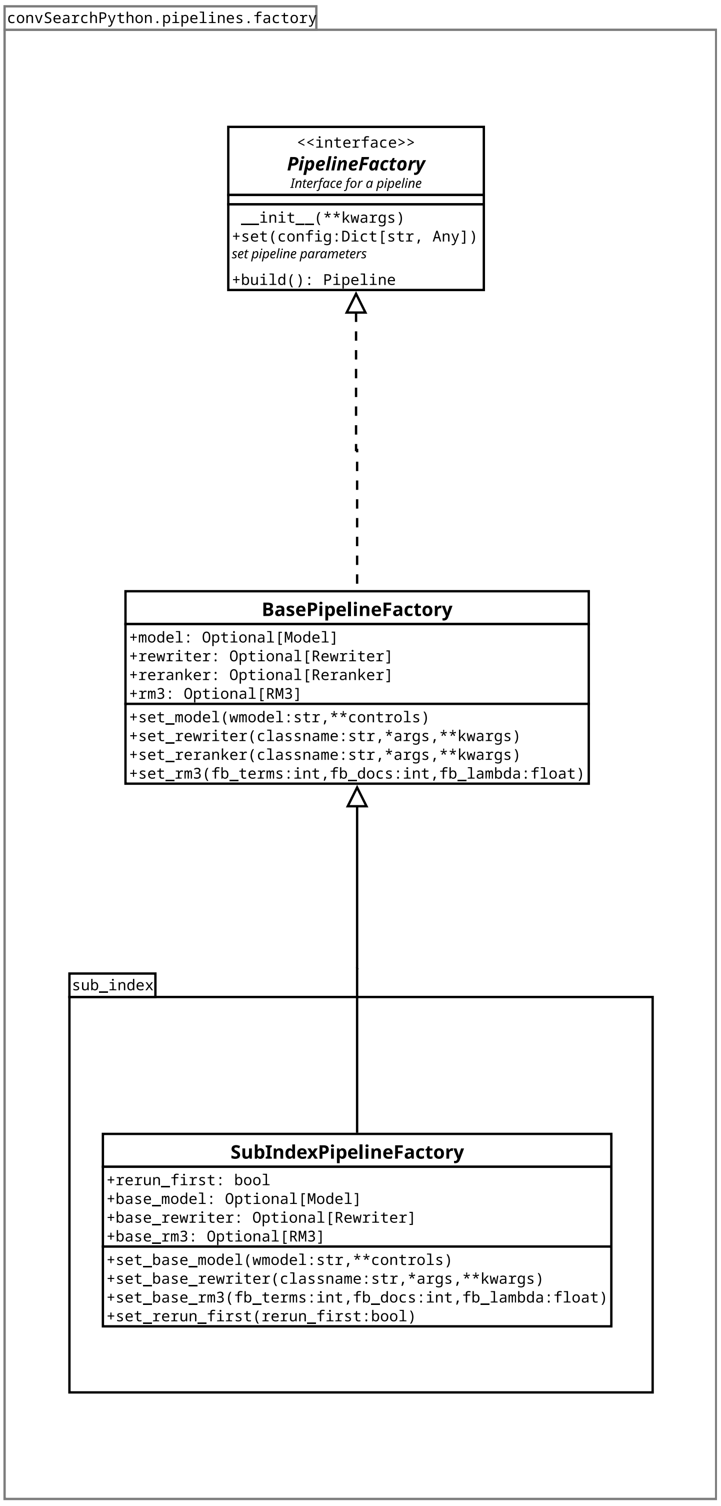

convSearchPython.pipelines.factory
<<interface>>
PipelineFactory
Interface for a pipeline
__init__(**kwargs)
+set(config:Dict[str, Any])
set pipeline parameters
+build(): Pipeline
BasePipelineFactory
+model: Optional[Model]
+rewriter: Optional[Rewriter]
+reranker: Optional[Reranker]
+rm3: Optional[RM3]
+set_model(wmodel:str,**controls)
+set_rewriter(classname:str,*args,**kwargs)
+set_reranker(classname:str,*args,**kwargs)
+set_rm3(fb_terms:int,fb_docs:int,fb_lambda:float)
sub_index
SubIndexPipelineFactory
+rerun_first: bool
+base_model: Optional[Model]
+base_rewriter: Optional[Rewriter]
+base_rm3: Optional[RM3]
+set_base_model(wmodel:str,**controls)
+set_base_rewriter(classname:str,*args,**kwargs)
+set_base_rm3(fb_terms:int,fb_docs:int,fb_lambda:float)
+set_rerun_first(rerun_first:bool)

Figure 4.2: Factory classes

4.1.3 PipelineFactory

The `PipelineFactory` interface (see Figure 4.2) allows creating `Pipeline` objects with the *Factory Method* pattern. It defines the following attributes:

set(config) method for parsing the configuration. `config` is a dictionary of options extracted from the run configuration file that the factory must parse to set its parameters. Every configurable aspect of the factory must be editable through this method;

build() method that returns a `Pipeline`'s instance in accordance to the factory configuration;

The framework provides two factory implementations:

BasePipelineFactory constructs a pipeline composed of:

$$[\text{Rewriter}] \rightarrow \text{Model} \rightarrow [\text{RM3}] \rightarrow [\text{Reranker}]$$

1. An optional `Rewriter`
2. A retrieval model
3. Optional RM3
4. An optional `Reranker`

SubIndexPipelineFactory constructs a pipeline that uses a sub-index strategy to retrieve conversational results.
For every conversation:

1. A first retrieval is done on the first utterance;
2. A temporary sub-index is generated with the returned documents;
3. Optionally, the first utterance's retrieval is executed again on the sub-index (`rerun_first` attribute);
4. All the subsequent utterances' retrievals are executed on the sub-index;

This factory accepts the same parameters of the previous one, plus a *base* rewriter, model, and RM3 that will be applied to the first utterance's retrieval only.

4.2 Methods

This section provides information on the implementation of the various methods. To understand the next algorithms, we need to define the basic data structure of queries and results.

Queries' Table Queries are represented by rows in a table with two columns:

- **qid**: contains the unique identifier for the query;
- **query**: contains the text associated with the query;

Results' Table Results are stored in an extension of the previous table with the following additional columns:

- **docno**: contains the identifier for the returned document (i.e., document number);
- **score**: contains the score of the returned document
- **rank**: contains the rank of the document, i.e., the position of the document by score value for the relative query;
- **text**: contains the text of the returned document;

Common functions We define these common functions that will be used in the followings pseudocodes:

map(qid) takes a query identifier and returns the conversation that contains said query[2];

join(q. . .) concatenates queries separating them with a space;

$\boldsymbol{join_{sep}(q\ldots)}$ concatenates queries separating them with a suitable separator;

$\boldsymbol{split_{sep}(q\ldots)}$ separate queries previously joined using $join_{sep}$;

$\boldsymbol{split_{table}}$ split a text by whitespace and construct a queries' table with a row for every word;

4.2.1 Concat Query Rewriter

Concat Query is implemented in the `ConcatQueryRewriter`. The class is conversationally parallel as we need all the queries in the same conversation. The concatenation is done by getting the query identifier and accessing the query structure to find the previous conversational utterances (Algorithm 4.1).

[2]This function is an exemplification of the actual procedure to access the queries' structure

Algorithm 4.1 ConcatQueryRewriter

```
1: procedure REWRITE_SINGLE($q$)
2:     $qid \leftarrow q[qid]$
3:     $conversation \leftarrow map(qid)$
4:     return $join(conversation_{\leq qid})$
5: end procedure
6: procedure REWRITE($queries$)
7:     for $q \in queries$ do
8:         $q[query] \leftarrow$ REWRITE_SINGLE($q$)
9:     end for
10:    return $queries$
11: end procedure
```

4.2.2 First Query Rewriter

The implementation class for *First Query* is called `FirstQueryRewriter`. It works similarly to `ConcatQueryRewriter`, this time concatenating only the first and the current utterance.

Given the ambiguity of the algorithm regarding the first query in a conversation, `FirstQueryRewriter` implements both the repeating and non-repeating strategies, with the non-repeating one used by default as no significant difference was observed in the resulting metrics. As you can see in Algorithm 4.2, an additional parameter (*repeat*) is used to select the desired variant. The REWRITE procedure is analogous to Algorithm 4.1.

Algorithm 4.2 FirstQueryRewriter

```
1: procedure REWRITE_SINGLE($q, repeat$)
2:     $qid \leftarrow q[qid]$
3:     $conversation \leftarrow map(qid)$
4:     if $qid = conversation[0][qid]$ then
5:         if $repeat$ then
6:             return $join(q, q)$
7:         else
8:             return $q[query]$
9:         end if
10:    end if
11:    return $join(conversation[0], q)$
12: end procedure
```

4.2.3 Context Query Rewriter

`ContextQueryRewriter` is the implementation class for *Context Query* algorithm.

Like with `FirstQueryRewriter`, it implements both the repeating a non-repeating variant to fill the ambiguity on the rewrite for the first and second utterance of a conversation and uses the non-repeating one by default. The Algorithm 4.3 shows the pseudocode for the REWRITE_SINGLE procedure.

Algorithm 4.3 ContextQueryRewriter

```
procedure REWRITE_SINGLE($q, repeat$)
    $qid \leftarrow q[qid]$
    $conversation \leftarrow map(qid)$
    if $qid = conversation[0][qid]$ then
        if $repeat$ then
            return $join(q, q, q)$
        else
            return $q[qid]$
        end if
    else if $qid = conversation[1][qid]$ then
        if $repeat$ then
            return $join(conversation[0], conversation[0], q)$
        else
            return $join(conversation[0], q)$
        end if
    end if
    return $join(conversation[0], conversation[1], q)$
end procedure
```

4.2.4 Coref 1 Rewriter

We provide a *Coref 1*'s cached and GPU-accelerated implementation named `AllennlpCoreferenceQueryRewriter`.

The GPU acceleration is automatically enabled if a *CUDA* environment is detected and currently support only one card. It's technically possible, though not trivial (with this framework's architecture), to implement a multi-GPU version of this rewriter. Instead, we focused on a cached approach that gives better speed benefits. If needed, it's permitted to disable both the acceleration and the cache, allowing to overcome eventual borderline cases.

The rewriter's type is decided at runtime. If the GPU acceleration is used, then it's considered `SEQUENTIAL`. Otherwise, it might try to access the same card with multiple processes resulting in a crash. Alternatively, if no acceleration is used, the returned type is `CONVERSATIONALLY_PARALLEL` to allow for some parallelization.

The Algorithm 4.4 shows the pseudocode for this rewriter, where the *resolve* function calls the *AllenNLP* coreference resolver.

Algorithm 4.4 AllennlpCoreferenceQueryRewriter

```
procedure COREF($q$)
    $qid \leftarrow q[qid]$
    $conversation \leftarrow map(qid)$
    $full\_concat \leftarrow join_{sep}(conversation_{\leq qid})$
    $full\_coref \leftarrow resolve(full\_concat)$
    return $split_{sep}(full\_coref)[last]$
end procedure
procedure REWRITE_SINGLE($q, cache$)
    $qid \leftarrow q[qid]$
    if $qid \in cache$ then
        return $cache[qid]$
    end if
    $rewrite \leftarrow$ COREF($q$)
    $cache[qid] \leftarrow rewrite$
    return $rewrite$
end procedure
procedure REWRITE($queries$)
    $cache \leftarrow load\_cache()$
    for $q \in queries$ do
        $q[query] \leftarrow$ REWRITE_SINGLE($q, cache$)
    end for
    $save\_cache(cache)$
    return $queries$
end procedure
```

4.2.5 Coref 2 Rewriter

The implementation for *Coref 2* is called `NeuralCorefRewriter`. It's a simpler implementation compared to the previous one as *neuralcoref* is pretty fast even on the CPU.

Two variants are provided that use a different *spacy* model: a default and a large one. Unfortunately, no improvement was observed with the large model. On Algorithm 4.5 you can see the pseudocode for the REWRITE_SINGLE function of this method that accepts an additional boolean parameter (*large*) that specify the variant to use.

Algorithm 4.5 NeuralCorefRewriter

```
1: procedure REWRITE_SINGLE($q, large$)
2:     $qid \leftarrow q[qid]$
3:     $conversation \leftarrow map(qid)$
4:     $full\_concat \leftarrow join_{sep}(conversation_{\leq qid}$
5:     $full\_coref \leftarrow resolve(full\_concat, large)$
6:     return $split_{sep}(full\_coref)[last]$
7: end procedure
```

4.2.6 Historical Query Rewriter

We provide two different implementation for *Historical Query Expansion*:

- `HistoricalQueryRewriter` is solely based on the algorithm description;
- `HQERewriter` is based on the castorini's HQE implementation[3]

To find suitable starting values for r_S, r_Q and θ we've proceeded as follows:

1. Ran a BM25 pipeline with single words treated as entire queries;
2. Manually analysed the results, comparing words' scores with:
   (a) How much important the word was in the original query;
   (b) How much important the word was in the entire conversation;
3. Ran a normal BM25 pipeline to compare the queries' scores to their ambiguity;

We found that:

- Ambiguous queries often have a score lower than 30 ($\theta = 30$);
- Session-relevant words tend to score at least 10 ($r_S = 10$);

[3]`https://github.com/castorini/chatty-goose`

- Query-relevant words usually score at least 15 ($r_Q = 15$);

Having $r_Q > r_S$ is justified by the fact that session-related words are frequently less specific than query-related ones. Moreover, our tests show that values of r_S bigger than r_Q produce results significantly worse than the opposite.

HistoricalQueryRewriter This version of *HQE* was the first implementation we made, looking only at [JC19], and follows the original algorithm as close as possible. You can see the pseudocode on Algorithm 4.6, with the following symbols:

tokens_table is a table with the current query's words in the query column;

$\boldsymbol{W_S}$ is the list of session-relevant keywords;

$\boldsymbol{W_Q}$ is the list of queries' relevant keywords, structure as a list of lists where every sub-list contains the keywords of a single query;

$\boldsymbol{first}$ is a boolean that signals if the current query is the first one in its conversations as we don't need to expand it;

Despite following the method's definition, this rewriter underperforms compared to the original paper's results.

HQERewriter Trying to pin down the reason for the performance below expectation, we decided to look at another *HQE* implementation and replicate it. `HQERewriter` is based on the code for the "Chatty Goose" framework developed by Castorini.

The most notable difference with our implementation resides in tokens filtering. `HQERewriter` uses *spacy* to analyse queries' text and only select nouns and adjectives as tokens.

Unfortunately, this version performs similarly to the previous one, meaning that the reason for the underperformance is likely in the core searching techniques of this framework.

4.2.7 DBpedia Rewriter

The *DBpedia* method is implemented in the `DbPediaRewriter` class. This method requires an additional parameter ($freq\_type$) to choose between the index-frequency and the snippets-frequency. The Algorithm 4.7 shows the pseudocode, where:

Algorithm 4.6 HistoricalQueryRewriter

```
1: procedure EXPANSION($text, first, W_S, W_Q$)
2:     $tokens\_table \leftarrow split(text)$
3:     $tokens\_table \leftarrow score(tokens\_table)$
4:     $Q_{rel} \leftarrow ()$
5:     for $row \in tokens\_table$ do
6:         if $row[score] > r_Q$ then
7:             $Q_{rel}.insert(row[query])$
8:         end if
9:         if $row[score] > r_S$ then
10:            $W_Q.insert(row[query])$
11:        end if
12:    end for
13:    $W_Q.insert(Q_{rel})$
14:    if $first$ then
15:        return $text$
16:    else
17:        $T \leftarrow tokens\_table[query] \cup W_S$
18:        if $score(text) < \theta$ then
19:            for $Q_{old} \in W_Q[last\ 4]$ do
20:                $T \leftarrow T \cup Q_{old}$
21:            end for
22:        end if
23:        return $join(T)$
24:    end if
25: end procedure
26: procedure REWRITE($queries$)
27:    $W_Q \leftarrow list()$
28:    $W_S \leftarrow list()$
29:    $current\_conv \leftarrow none$
30:    for $q \in queries$ do
31:        if $conversation\_of(q) \neq current\_conv$ then
32:            $W_Q.clear();\ W_S.clear()$
33:            $q[query] \leftarrow$ EXPANSION($q[query], true, W_S, W_Q$)
34:        else
35:            $q[query] \leftarrow$ EXPANSION($q[query], false, W_S, W_Q$)
36:        end if
37:    end for
38:    return $queries$
39: end procedure
```

***spotlight*(q_c)** is the function that query *DBpedia Spotlight* and return the list of found entities;

***wiki*(*entity*)** is the function that queries *Wikipedia* and returns snippets for the specified entity;

score_terms*(*snippets*, *freq_type is the function that scores the terms using either their frequency in the documents' index or their frequency in the returned snippets;

find_nouns_adjectives is a function that returns a list of nouns and adjectives (filtered with *spacy*) from the provided query;

concept_net is the function that queries *ConceptNet* and returns a list of terms related with the provided one;

To select the n-grams for *ConceptNet* (in function $find\_nouns\_adjectives$), we decided to choose adjacent nouns and adjectives from the queries' concatenated text, as it's likely for adjacent words to be related.

Algorithm 4.7 DbPediaRewriter

```
1:  procedure REWRITE_SINGLE($q, freq\_type$)
2:      $q_c \leftarrow join(map(q[qid]))$
3:      $entities \leftarrow spotlight(q_c)$
4:      $P \leftarrow list(q_c)$
5:      if $len(entities) > 0$ then
6:          for $entity \in entities$ do
7:              $snippets \leftarrow wiki(entity)$
8:              $scored\_terms \leftarrow score\_terms(snippets, score\_type)$
9:              $P \leftarrow P \cup scored\_terms[first\ 10]$
10:         end for
11:     else
12:         $ngrams \leftarrow find\_nouns\_adjectives(q_c)$
13:         for $n \in ngrams$ do
14:             $P \leftarrow P \cup \{concept\_net(n)\}$
15:         end for
16:     end if
17:     return $join(P)$
18: end procedure
```

4.2.8 Seen Filter Reranker

Seen Filter is implemented in `SeenFilterReranker`. The implementation is close to the formal definition and employs two additional types of data structures:

- `set` is an unordered collection of elements that doesn't allow duplicates (i.e., adding an already-existing element won't change the set);
- `dict` (dictionary) is a collection of key-value pairs where every key exist only one time (i.e., setting a value for an already-existing key will replace the old one);

The Algorithm 4.8 shows the pseudocode, where the function $conversation(qid)$ returns the identifier for the conversation that contains the query with the specified qid.

Algorithm 4.8 SeenFilterReranker

```
 1: procedure RERANK_SINGLE($r, k, m, seen$)
 2:     $score \leftarrow r[score]$
 3:     $qid \leftarrow r[qid]$
 4:     $conv\_id \leftarrow conversation(qid)$
 5:     if $conv\_id \notin seen$ then
 6:         $seen[conv\_id] \leftarrow set()$
 7:     end if
 8:     if $q[docno] \in seen[conv\_id]$ then
 9:         $score \leftarrow score \cdot m$
10:     end if
11:     if $q[rank] \leq k$ then
12:         $seen[conv\_id].add(q[docno])$
13:     end if
14:     return $score$
15: end procedure
16: procedure RERANK($results, k, m$)
17:     $seen \leftarrow dict()$
18:     for $r \in results$ do
19:         $r[score] \leftarrow$ RERANK_SINGLE($r, k, m, seen$)
20:     end for
21: end procedure
```

4.2.9 Bottom Up Reranker

We've implemented the *Bottom Up* algorithm in the `BottomUpReranker` class. As it's possible to see in Algorithm 4.9, it requires a 2-pass strategy:

1. In the first pass it loops through every conversation and every qid (in reversed order) to find all the (qid, docno) pairs that must be filtered;

2. In the second pass it changes the score of the previously identified rows;

This is done in two passages to manipulate the results' table more easily and keep the queries' order.

Algorithm 4.9 BottomUpReranker

```
procedure RERANK($results, k, m$)
    $R \leftarrow dict()$
    for $conversation \in results$ do
        $seen = set()$
        for $query\_results \in reversed(conversation)$ do
            $R[qid] \leftarrow set(query\_results[docno] \cap seen)$
            $top\_docs \leftarrow list(query\_results[docno])[1..k]$
            $seen \leftarrow seen \cup top\_docs$
        end for
    end for
    for $conversation \in results$ do
        for $r \in conversation$ do
            if $r[docno] \in R[qid]$ then
                $r[score] \leftarrow r[score] \cdot m$
            end if
        end for
    end for
    return $results$
end procedure
```

4.2.10 Historical Answer Reranker

Historical Answer Expansion is implemented in the `HAEReranker` class as `SEQUENTIAL` reranker. The algorithm is divided in two phases: classification and ranking.

Classification In the CLASSIFY function on Algorithm 4.10, we compute the log-likelihood value for every row. We use the `Hugging Face`[4] and `pytorch`[5] libraries to work with the BERT model. In particular, `Hugging Face` already has the same pre-trained model used in the original *HAE* paper in its collection. The classification process performs four steps:

1. The BERT tokenizer is used to encode the query-document pair in vectors compatibles with the model;
2. The BERT model is applied to the encoding to obtain the logits, i.e., the vector of raw predictions;
3. The *softmax* function is used to normalize the logits;
4. The negative logarithm is computed on the wanted prediction, i.e., the probability that the query and the documents belong to the same class. This is the *log-likelihood* for the current row;

Given the heaviness of the computation, it's recommended to use a GPU classifier. Our implementation includes both a CPU[6] and a GPU classifier, where the latter will be automatically chosen if a *CUDA* environment is detected. The GPU implementation have a limited support for multi-card environments[7].

Ranking The ranking phase, shown in the RERANK procedure in Algorithm 4.10, uses the likelihood computed with the BERT classifier to apply the *HAE* technique, staying as close as possible to the formal definition.

To lighten the computation, the reranker accept an additional parameter (k) that specify a cut-off for the number of documents-per-query that should be fed to the classifier.

4.3 Pipelines

In addition to the single rewriters and rerankers, our framework includes some pre-constructed pipelines that represent some noteworthy methods. We have implemented the following pipelines:

- **PlainBM25Pipeline**: simple pipeline with *BM25*, optional RM3 and no rewriters nor rerankers;

[4]`https://huggingface.co/`
[5]`https://pytorch.org/`
[6]single core implementation
[7]Multi-GPU use a threading model, so the CPU will become a bottleneck at some point

Algorithm 4.10 HAEReranker

```
procedure CLASSIFY($results$)
    for $row \in results$ do
        $encoding \leftarrow bert\_tokenizer(row[query], row[text])$
        $logits \leftarrow bert\_model(encoding)[logits]$
        $sft \leftarrow softmax(logits)$
        $row[log] \leftarrow -ln(sft[0][1])$
    end for
    return $results$
end procedure
procedure RERANK($results, \lambda, k$)
    $results \leftarrow$ * max k rows per qid *
    $results \leftarrow$ CLASSIFY($results$)
    $last \leftarrow list()$
    for $conversayion \in results$ do
        for $qid, query\_results \in conversation$ do
            $curr \leftarrow query\_results[docno, log]$
            $temp \leftarrow curr \cup \{(docno,\ log \cdot \lambda) \in last \setminus curr\}$
            $last \leftarrow curr$
            $temp \leftarrow temp.sort()[top\ k]$
            for $row \in query\_results$ do
                $row[log] \leftarrow temp[docno = row[docno]][log]$
            end for
        end for
    end for
    $results[score] \leftarrow results[log]$
    $result.sort()$
    return $results$
end procedure
```

- **PlainPipeline**: simple pipeline with *Dirichlet language model*, optional RM3 and no rewriters nor rerankers;
- **ConcatQueryPipeline**: extension of PlainPipeline with the `ConcatQueryRewriter`;
- **FirstQueryPipeline**: extension of PlainPipeline with the `FirstQueryRewriter`;
- **ContextQueryPipeline**: extension of PlainPipeline with the `ContextQueryRewriter`;
- **Coreference1Pipeline**: extension of PlainPipeline with the `AllennlpCoreferenceQueryRewriter`;
- **Coreference2Pipeline**: extension of PlainPipeline with the `NeuralCorefRewriter`
- **DBPediaPipeline**: extension of PlainPipeline with the `DbPediaRewriter`;
- **BottomUpUamPipeline**: implementation of the *Bottom Up* method, using BM25 and the `BottomUpReranker`;
- **HistoricalQueryExpansionPipeline**: implementation of *HQE* with BM25 and `HQERewriter` (or, optionally, `HistoricalQueryRewriter`);
- **HistoricalQueryAndAnswerExpansionPipeline**: extension of the previous with `HAEReranker`;

4.4 Technical Details

The framework is developed in python 3.8 (as required by some dependencies) using `pyterrier` for the core retrieval mechanism. `Step` objects (including rewriters and rerankers) must conform to the *pyterrier data model*[8].

Non-conforming Step Provided `PipelineFactory` classes are written to allow reusing pyterrier transformers or generic callable objects that conform to the pyterrier data model. However, when doing so, some assumptions will be made to fill for missing properties:

[8]`https://pyterrier.readthedocs.io/en/latest/datamodel.html`

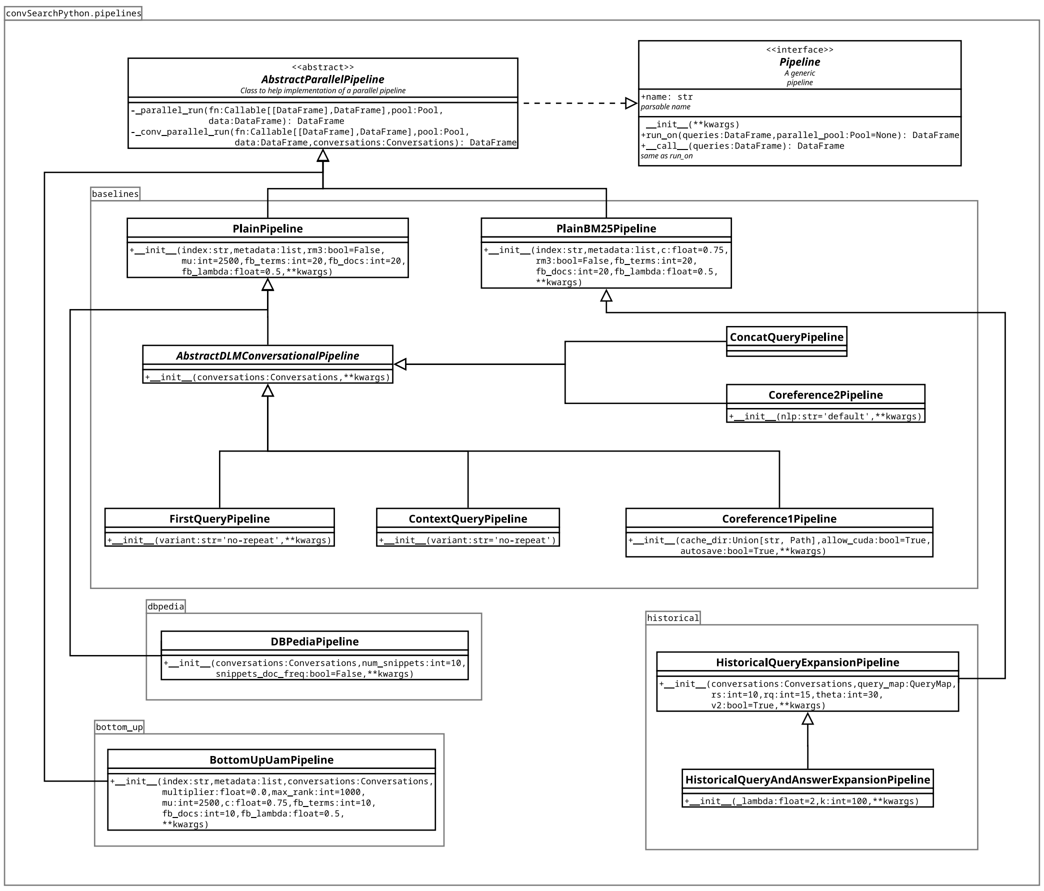

convSearchPython.pipelines
<<abstract>>
AbstractParallelPipeline
Class to help implementation of a parallel pipeline
-_parallel_run(fn:Callable[[DataFrame],DataFrame],pool:Pool, data:DataFrame): DataFrame
-_conv_parallel_run(fn:Callable[[DataFrame],DataFrame],pool:Pool, data:DataFrame,conversations:Conversations): DataFrame
<<interface>>
Pipeline
A generic pipeline
+name: str
parsable name
__init__(**kwargs)
+run_on(queries:DataFrame,parallel_pool:Pool=None): DataFrame
+__call__(queries:DataFrame): DataFrame
same as run_on
baselines
PlainPipeline
+__init__(index:str,metadata:list,rm3:bool=False, mu:int=2500,fb_terms:int=20,fb_docs:int=20, fb_lambda:float=0.5,**kwargs)
PlainBM25Pipeline
+__init__(index:str,metadata:list,c:float=0.75, rm3:bool=False,fb_terms:int=20, fb_docs:int=20,fb_lambda:float=0.5, **kwargs)
AbstractDLMConversationalPipeline
+__init__(conversations:Conversations,**kwargs)
ConcatQueryPipeline
Coreference2Pipeline
+__init__(nlp:str='default',**kwargs)
FirstQueryPipeline
+__init__(variant:str='no-repeat',**kwargs)
ContextQueryPipeline
+__init__(variant:str='no-repeat')
Coreference1Pipeline
+__init__(cache_dir:Union[str, Path],allow_cuda:bool=True, autosave:bool=True,**kwargs)
dbpedia
DBPediaPipeline
+__init__(conversations:Conversations,num_snippets:int=10, snippets_doc_freq:bool=False,**kwargs)
historical
HistoricalQueryExpansionPipeline
+__init__(conversations:Conversations,query_map:QueryMap, rs:int=10,rq:int=15,theta:int=30, v2:bool=True,**kwargs)
HistoricalQueryAndAnswerExpansionPipeline
+__init__(_lambda:float=2,k:int=100,**kwargs)
bottom_up
BottomUpUamPipeline
+__init__(index:str,metadata:list,conversations:Conversations, multiplier:float=0.0,max_rank:int=1000, mu:int=2500,c:float=0.75,fb_terms:int=10, fb_docs:int=10,fb_lambda:float=0.5, **kwargs)

Figure 4.3: Pipelines

- The class name will be used as step name, so the resulting pipeline name won't contain the configuration parameters. Consequently, issuing the same pipeline with different step's configuration won't work, as the subsequent pipelines will be discarded for being duplicates;
- The step will be considered `FULLY_PARALLEL`;

Even when providing the previous properties, it's not mandatory to extend the `Step` class as factories will try to dynamically load the property before falling back to the mentioned assumptions.

Spacy version *Neuralcoref* requires *spacy* 2, but the 3^{rd} version was released some times ago. Consequentially, classes that use it should account for both version to allow reusability. Provided classes that use *spacy*, except for `NeuralCorefRewriter`, account for that possibility.

Chapter 5

Experiments

5.1 Experimental setup

This work uses "TREC Conversational Assistance Track" (CAsT) [DXC20] as a source for topics and evaluations[1].

TREC CAsT is an initiative to facilitate Conversational Information Seeking research and create a large-scale reusable test collection for conversational search systems. The goal of the CAsT Task (in the 2019 edition) is to satisfy a user's information need (expresses as turns in a conversation) with responses limited to brief text passages (of $\approx$ 1-3 sentence length). Formally, given a sequence of conversational turns for a topic (T) with utterances (u), for each turn $T = \{u_1, ..., u_i, ..., u_n\}$, the task is to identify relevant passages P_i for each turn to satisfy the information needs in round i with the context of round $u_{<i} = \{u_1, ..., u_{i-1}\}$.

5.1.1 Topics

The topics were semi-manually constructed from a combination of previous TREC initiatives, MS MARCO Conversational Search Sessions and the authors' expertise.

They were selected to ensure:

- Requirement of multiple rounds of elaboration;
- Open-domain (no expert domain knowledge);
- Answerable within the passages' collection;

[1] `https://www.treccast.ai/`

Conversational utterances were manually created for each turn in a topic, ensuring that:

- Later turns only depend on previous utterances;
- Mimic realistic dialogues;
- Contains common conversation phenomena (e.g., coreference and omission);
- Simple *factoid* responses are insufficient;

5.1.2 Collection

The collection of passages used for retrieval includes the passages from *MS MARCO*[2] and the paragraph collection from *TREC CAR Y2*[3] [Die+18].

To handle the presence of near-duplicates, duplicate files are provided for both sources, so the resulting collection only contains unique passages.

Queries relevance

To compute the scores for the results, *TREC CAsT* provides a collection of *relevant judgements* composed of utterance-document pairs, manually labelled in the $[0, 4]$ range, where:

4. *Fully meets*: the passage is a perfect answer;
3. *Highly meets*: the passage is a satisfactory answer;
2. *Moderately meets*: the passage partially answer the query or contains unrelated information;
1. *Slightly meets*: the passage includes some information but doesn't answer the query;
0. *Fails to meet*: the passage is unrelated;

Given that the labels only cover a portion of the topics, from now on, we will restrict to that for analysis and results scoring (20 conversations, 173 queries covered with a total of 29350 labelled documents).

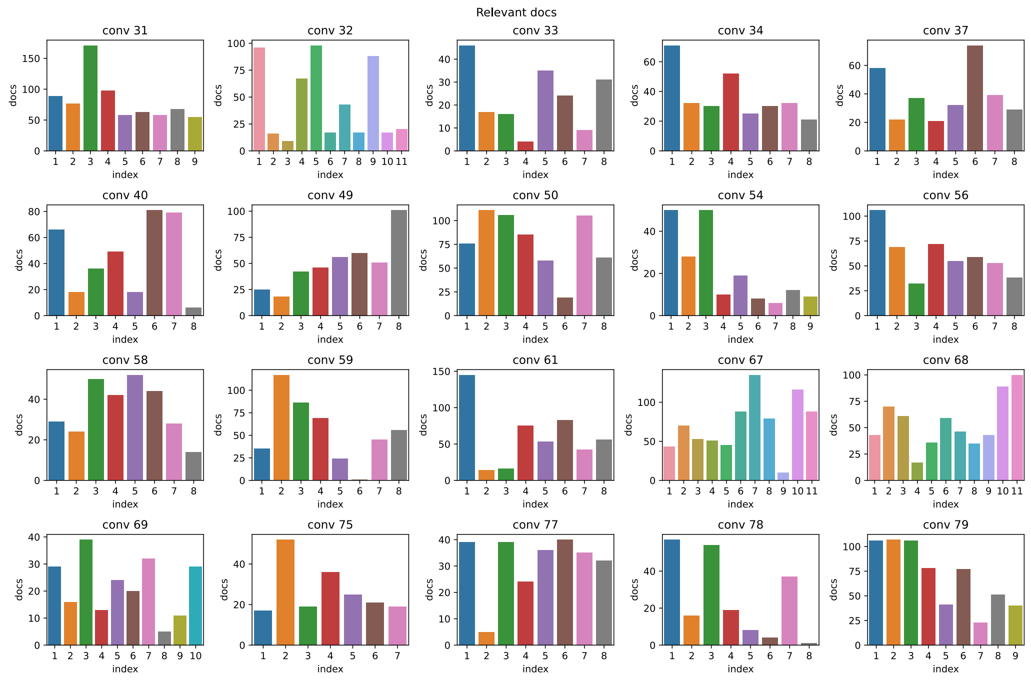


Figure 5.1: Number of relevant documents

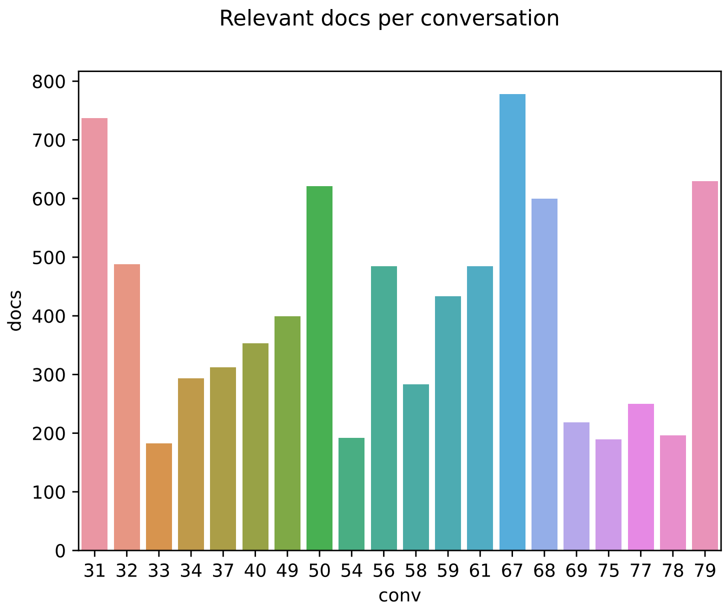


Figure 5.2: Mean of relevant documents' number per conversation

Relevance analysis

Figure 5.1 shows the number of relevant documents for every utterance in a conversation. We can observe that it's not equally distributed (in particular, for some queries, there are only a few relevant passages), and this will likely reflect on the results. In Figure 5.2 is shown the conversation mean of the same data.

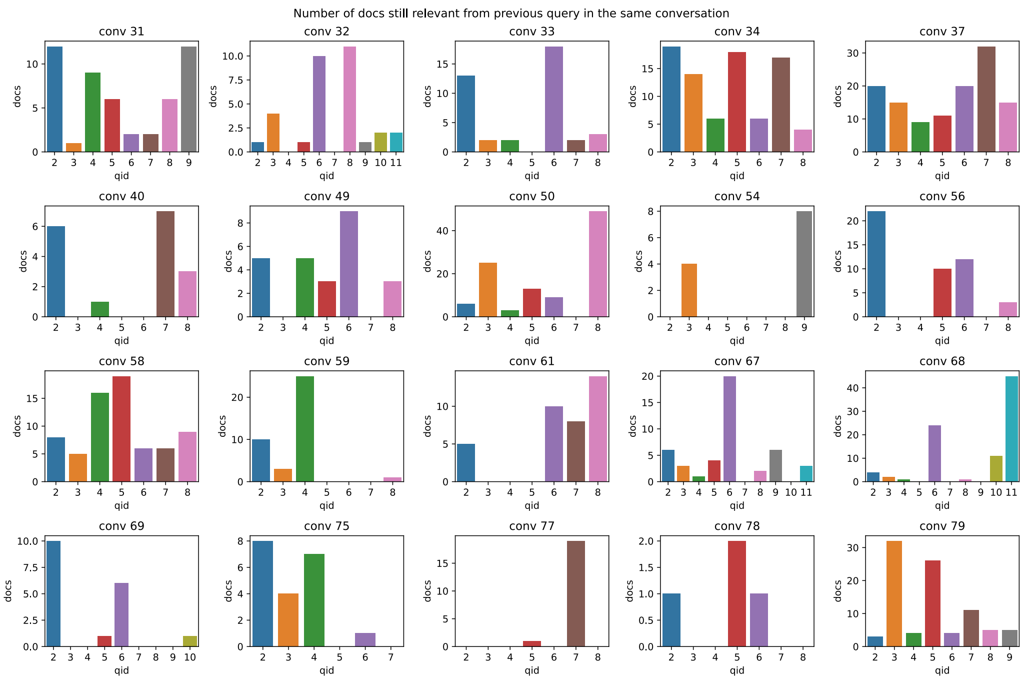


Figure 5.3: Number of still-relevant documents from the previous query

Figure 5.3 shows the number of documents, from the second utterance onwards, that are relevant for both the previous and the current query. This is interesting for methods like *Seen Filter* or *Bottom Up* where the documents from previous (or subsequent) utterances are filtered.

Figure 5.4 shows the heatmaps, conversation per conversation, of the **cosine similarity** between queries. This gives an idea of the dependence between the queries in a conversation. The cosine similarity was obtained by:

1. Constructing a list (of arbitrary order) with all the terms in the two queries (every term a single time);

[2]http://www.msmarco.org/
[3]https://trec-car.cs.unh.edu/

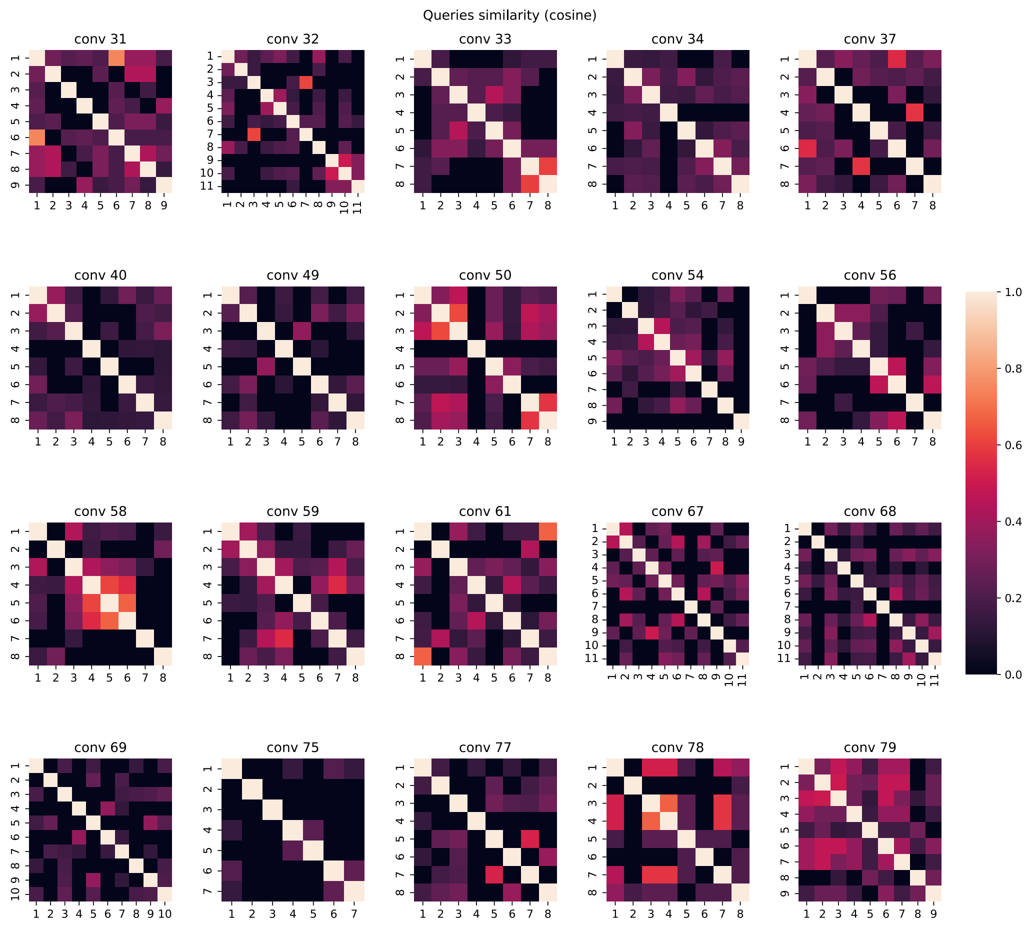


Figure 5.4: Cosine similarity between queries

2. Constructing two vectors, one for every query, that have, for every component, 1 if the query contains the term, 0 otherwise;

3. Computing the cosine similarity as

$$\frac{A \cdot B}{\parallel A \parallel \parallel B \parallel} \tag{5.1}$$

Figure 5.5 and Figure 5.6 show the overall recall when considering the documents returned from the first conversational utterance only. This provides some insight on the performance that we can expect from the *sub-index* technique, where the search for subsequence utterance is performed on the first utterance results' documents.

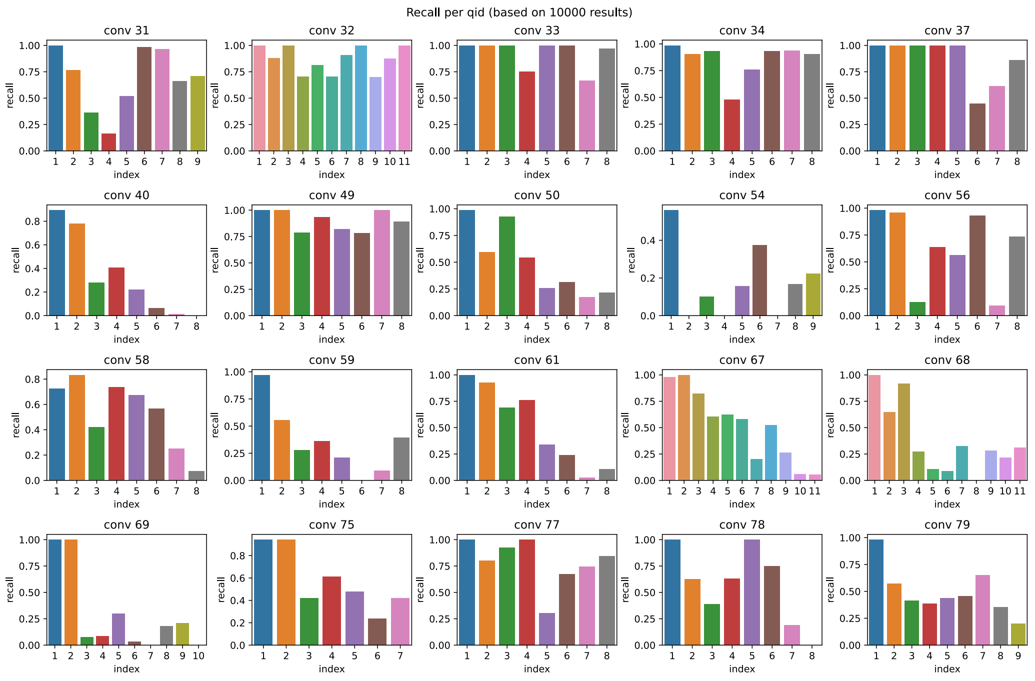


Figure 5.5: Recall per qid on the docs retrieved with the 1st utterance

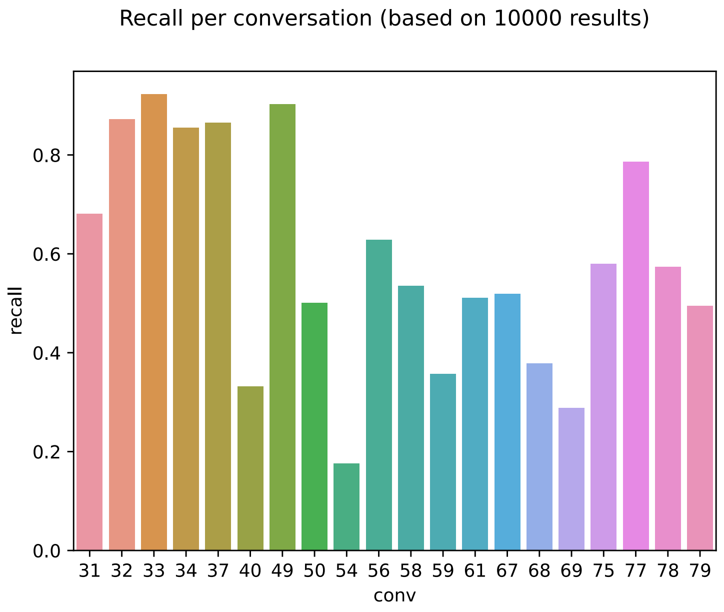


Figure 5.6: Recall per conversation on the docs retrieved with the 1st utterance

5.2 Results

This section shows some results on the discussed methods. Following *TREC CAsT* [DXC20] the main evaluation metric will be the *discounted cumulative gain* for the first three documents (`nDCG@3`), calculated as follows:

$$nDCG@3 = \sum_{i=1}^{3} \frac{rel_i}{log_2(i+1)} \tag{5.2}$$

where rel_i is the label for the i-th document.

We decided to consider also the *recall* for the first 100 documents (`R@100`) because methods with a good recall but a poor *nDCG@3* might still be used in combination with other reranking techniques.

$$R@100 = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \tag{5.3}$$

Sometimes, we provide additional metrics to compare our implementation with the original results, as not every author has picked the same evaluation metrics we have chosen.

For the retrieval stage, we used an index with the Krovetz stemmer and the Indri's[4] stopwords list, to be as compliant as possible to the indexing settings on [Mel+21]. All the results were calculated by doing the mean of the per-conversation mean of the per-query results.

5.2.1 Concat Query

Concat Query was tested using *Dirichlet language model* of parameter μ and *RM3 pseudo relevance feedback* (prf). Table 5.1 shows the best results obtained, sorted by `nDCG@3`. It's plain to notice that the first three results are identical: this is confirmed by Figure 5.7d where it's shown that these variations are equivalent.

μ	RM3	AP	R@100	nDCG@3
1000	t5-d20-l0.3	0.174461	0.322022	0.277390†
1000	t5-d20-l0.5	0.174461	0.322022	0.277390
1000	t5-d20-l0.7	0.174461	0.322022	0.277390
1000	t5-d30-l0.3	0.178709	0.322448	0.275817

Table 5.1: Concat Query results

[4] `https://www.lemurproject.org/indri/`

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	82.6448	19	4.34973	3739.76	$<$ 1e-5	0.9548
model	0.0036	5	0.00073	0.62	0.6817	—
prf	3.9509	27	0.14633	125.81	$<$ 1e-5	0.5007
Error	3.8475	3308	0.00116			
Total	90.4469	3359				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	57.5769	19	3.03036	1097.14	$<$ 1e-5	0.8611
model	0.2626	5	0.05253	19.02	$<$ 1e-5	0.0261
prf	1.5993	27	0.05923	21.45	$<$ 1e-5	0.1411
Error	9.1369	3308	0.00276			
Total	68.5758	3359				

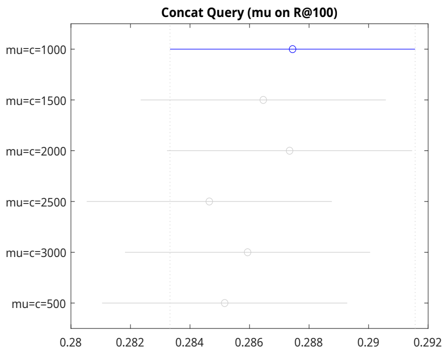
(b) nDCG@3

Table 5.2: ANOVA Tables for Concat Query

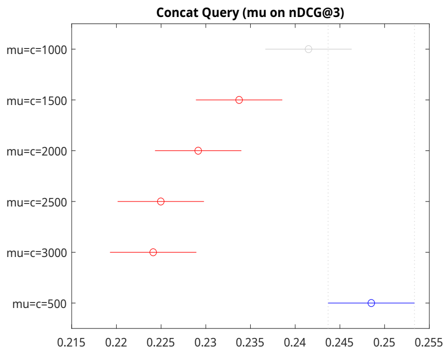
Table 5.2 and Figure 5.7 show the statistical analysis performed on these results. On `R@100`, we can observe that both *conversation* and *RM3* cause a strong variation on the results. However, looking at Figure 5.7c we can see that RM3 always worsen the results. The model variations (i.e., μ), instead, does not have a meaningful impact.

On `nDCG@3`, all the three parameters (conversation, model and prf) provides a significant contribution. In particular, both *conversation* and *RM3* have a strong effect, while *model* only has a small one.

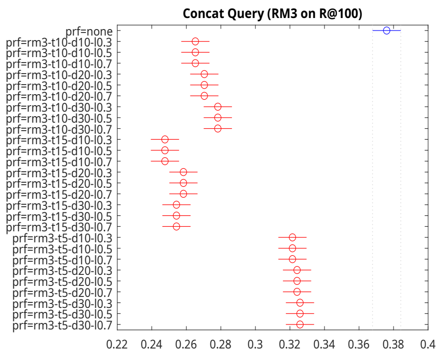
The strong impact of conversations in both metrics is confirmed by Figure 5.8 where we can see that there is a wide performance variation between them. In addition, we can notice that not all the conversations that scores better with a metrics, keep that position on the other.



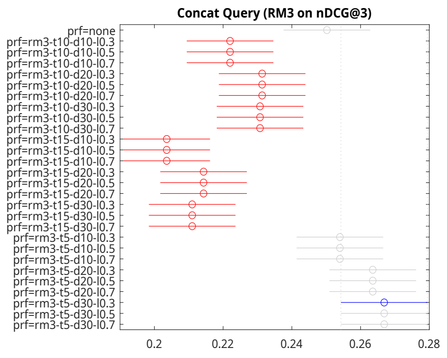
(a) Dirichlet μ on R@100



(b) Dirichlet μ on nDCG@3

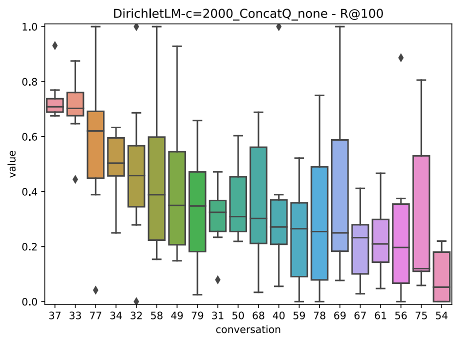


(c) RM3 on R@100

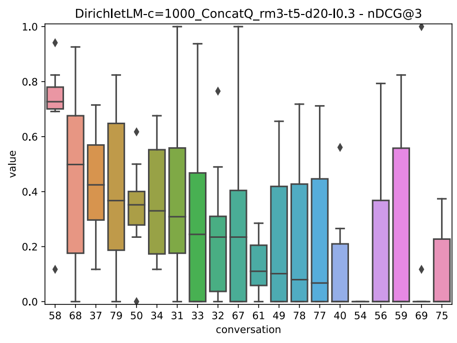


(d) RM3 on nDCG@3

Figure 5.7: Multcompare for Concat Query



(a) R@100



(b) nDCG@3

Figure 5.8: Concat Query performances

5.2.2 First Query

First Query was tested with *Dirichlet language model* of parameter μ, *RM3 pseudo relevance feedback* and using both the repeating and non-repeating (no-rep) variant.

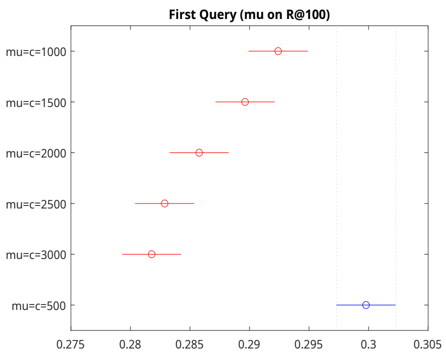
Table 5.3 presents the first results (ordered by `nDCG@3`) and the comparison with the original [Mel+21] values. As can be noted, the first values are the same: this agrees with Figure 5.9d where the multiple comparison shows the equivalence of these values. Making a comparison with the original values, our same-settings run scored significantly lower on recall and average precision and slightly lower on `nDCG@3`. However, our better run scored better on the discounted cumulative gain, slightly lower on the recall and similarly on the average precision.

Given the simplicity of the method, we think that there might be two possible explanation for the difference with the same-settings run. Either the original paper used different settings that were not reported, or the underlying retrieval framework is responsible for this difference.

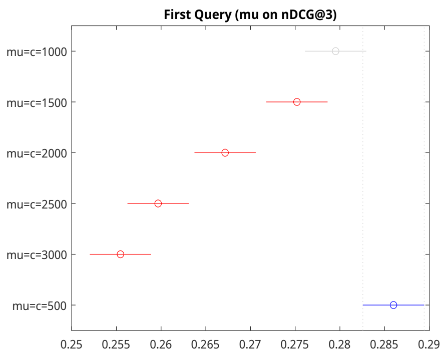
mu	variant	RM3	AP	R@100	R@200	nDCG@3
1000	repeat	t5-d20-l0.3	0.202914	0.330487	0.414397	0.327624†
1000	repeat	t5-d20-l0.5	0.202914	0.330487	0.414397	0.327624
1000	no-rep	t5-d20-l0.7	0.202914	0.330487	0.414397	0.327624
1000	no-rep	t5-d20-l0.3	0.202914	0.330487	0.414397	0.327624
1000	no-rep	t5-d20-l0.5	0.202914	0.330487	0.414397	0.327624
1000	no-rep	t5-d20-l0.7	0.202914	0.330487	0.414397	0.327624
2000	repeat	t5-d30-l0.3	0.195073	0.329968	0.426240	0.315478†
2500	no-rep	t20-d20-l0.5	0.146738	0.259525	0.333781	0.242949†
original [Mel+21]						
2500	—	t20-d20-l0.5	0.2091	—	0.4588	0.2663

Table 5.3: First Query results

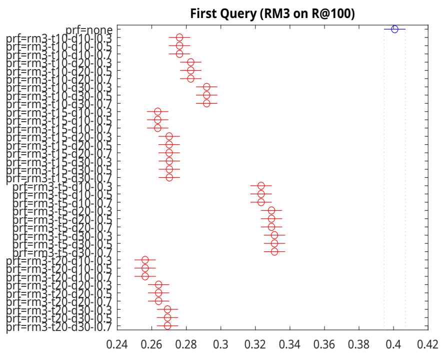
The statistical analysis of the results is displayed on Table 5.4 and Figure 5.9. In both metrics, the rewriting *variant* is irrelevant to the results, while *conversation*, *model* and *RM3* provides a significant contribution. The *model*'s contribution is equally small in both cases. Instead, *RM3* holds a large-but-negative (Figure 5.9c) effect on the recall and a medium one on the discounted cumulative gain. Like with *Concat Query*, the high ω^2 value for the conversations is the symptom of a wide performance variation (Figure 5.10).



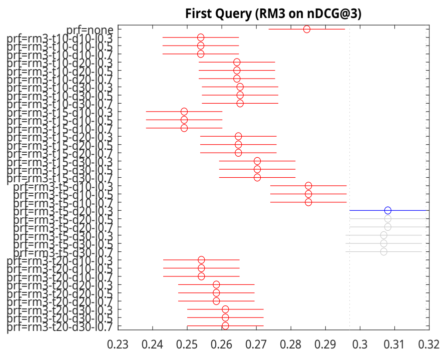
(a) Dirichlet μ on R@100



(b) Dirichlet μ on nDCG@3

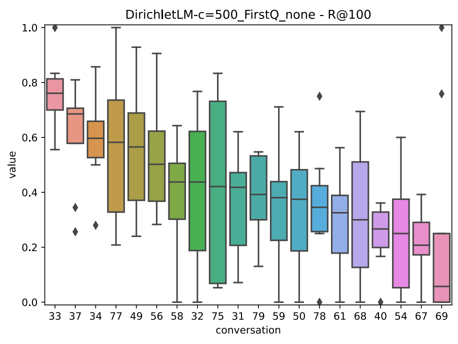


(c) RM3 on R@100

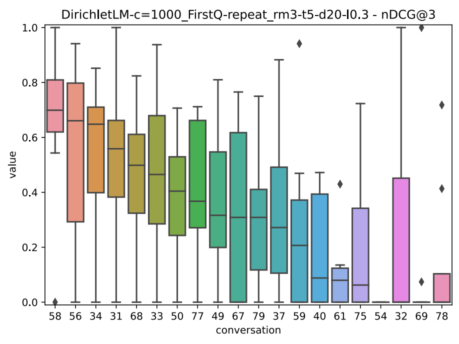


(d) RM3 on nDCG@3

Figure 5.9: Multcompare for First Query



(a) R@100



(b) nDCG@3

Figure 5.10: First Query performances

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	196.17	19	10.3247	8329.36	$< 1e{-4}$	0.9469
model	0.337	5	0.0674	54.4	$< 1e{-4}$	0.0292
variant	0	1	0	0	1	—
prf	8.938	36	0.2483	200.29	$< 1e{-4}$	0.4469
Error	10.93	8818	0.0012			
Total	216.375	8879				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	147.165	19	7.74554	1961.59	$< 1e{-5}$	0.8075
model	1.034	5	0.20674	52.36	$< 1e{-5}$	0.0281
variant	0	1	0	0	1	—
prf	3.125	36	0.0868	21.98	$< 1e{-5}$	0.07847
Error	34.819	8818	0.00395			
Total	186.142	8879				

(b) nDCG@3

Table 5.4: ANOVA Tables for First Query

5.2.3 Context Query

Context Query was tested with *Dirichlet language model* of parameter μ, *RM3 pseudo relevance feedback* and using both the repeating and non-repeating (no-rep) variant.

Table 5.5 shows the first results (by `nDCG@3`) and the comparison with the [Mel+21] result. We can immediately notice that the first six results are identical, as shown also in Figure 5.11d.

Comparing the results with the original (using the same settings), our implementation performs slightly better on the discounted cumulative gain, while it underperforms on recall and average precision. Our best result, instead, achieved better scores in all the three comparing metrics. Like for *First Query*, we think this difference is caused by either the usage of different settings in the original work or by the underlying retrieval framework.

The statistical analysis for this method's results is displayed in Table 5.6 and Figure 5.11. We can observe that the *variant* (repeat or no-repeat) is irrelevant on both metrics.

On `R@100`, the *model* provides a very small (but still significative) contribution and *RM3* a strong one, although Figure 5.11c shows that the RM3 contribution has only a negative effect.

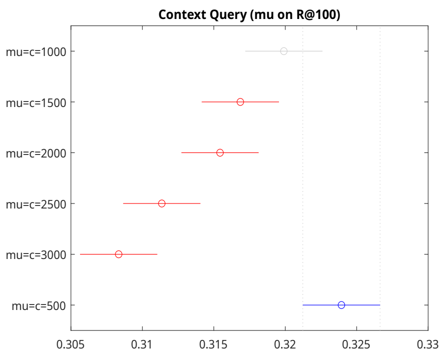
On `nDCG@3`, the significance of *RM3* is reduced, and we can see on Fig-

mu	variant	RM3	AP	R@100	R@200	nDCG@3
1000	no-rep	t5-d30-l0.3	0.215190	0.379442	0.471900	0.311693†
1000	no-rep	t5-d30-l0.5	0.215190	0.379442	0.471900	0.311693
1000	no-rep	t5-d30-l0.7	0.215190	0.379442	0.471900	0.311693
1000	no-rep	t5-d30-l0.3	0.215190	0.379442	0.471900	0.311693
1000	no-rep	t5-d30-l0.5	0.215190	0.379442	0.471900	0.311693
1000	no-rep	t5-d30-l0.7	0.215190	0.379442	0.471900	0.311693
1000	repeat	t5-d20-l0.3	0.209381	0.367899	0.457245	0.308339
2500	no-rep	t20-d20-l0.5	0.159414	0.283115	0.364840	0.258069†
original [Mel+21]						
2500	—	t20-d20-l0.5	0.1903	—	0.4263	0.2315

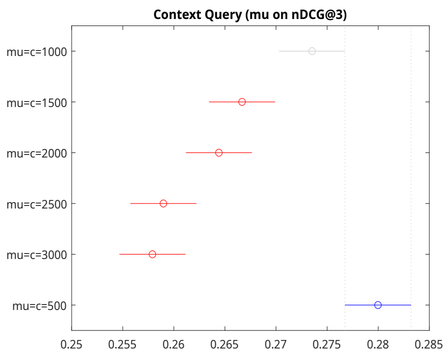
Table 5.5: Context Query results

ure 5.11d that there are no values of it that provide a meaningful improvement on the results.

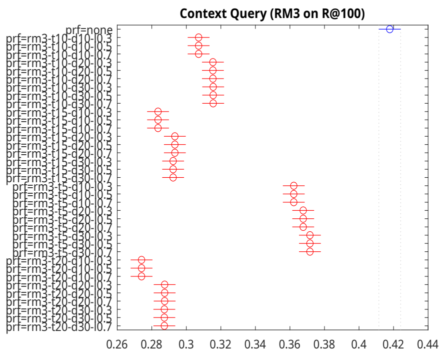
Even in this case, we see a wide performance variation between different conversations. Additionally, it's noticeable in Figure 5.12b that there is a huge variance in the performance of the same conversation. This is likely caused by the presence of *context shifts* that completely overturn the effectiveness of this method.



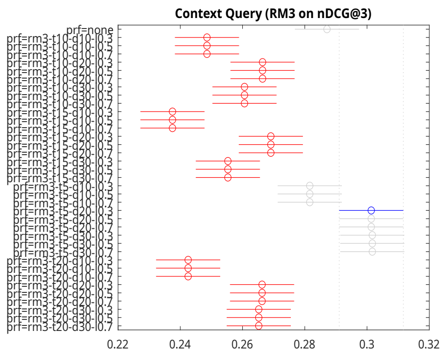
(a) Dirichlet μ on R@100



(b) Dirichlet μ on nDCG@3

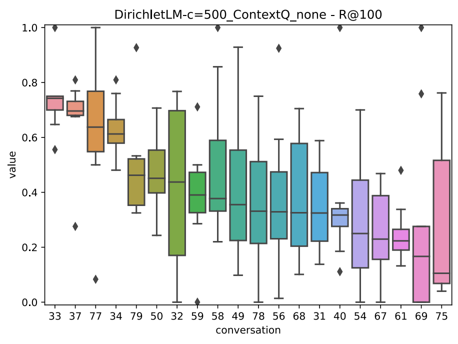


(c) RM3 on R@100

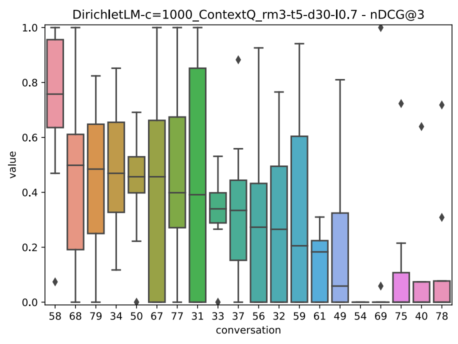


(d) RM3 on nDCG@3

Figure 5.11: Multcompare for Context Query



(a) R@100



(b) nDCG@3

Figure 5.12: Context Query performances

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	193.357	19	10.1767	7871.52	$<$ 1e-4	0.9439
model	0.236	5	0.0472	36.49	$<$ 1e-4	0.0196
variant	0	1	0	0	1	—
prf	12.202	36	0.339	262.18	$<$ 1e-4	0.5143
Error	11.4	8818	0.0013			
Total	217.195	8879				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	153.386	19	8.07292	2346.85	$<$ 1e-5	0.8339
model	0.54	5	0.10804	31.41	$<$ 1e-5	0.0168
variant	0	1	0	0	1	—
prf	3.405	36	0.09458	27.5	$<$ 1e-5	0.0970
Error	30.333	8818	0.00344			
Total	187.664	8879				

(b) nDCG@3

Table 5.6: ANOVA Tables for Context Query

5.2.4 Coref 1

Coref 1 was tested using *Dirichlet language model* of parameter μ and *RM3 pseudo relevance feedback*. Table 5.7 shows the first results and the comparison with the original [Mel+21] one. The first three results are equivalent, in accordance with Figure 5.13d.

Using the same settings of the original work, our implementation scored a little better on `nDCG@3`, but significative lower on the recall. Our best result, instead, achieved a better score in all the three considered metrics (`nDCG@3`, `R@200`, `AP`). Like with the previous methods, we think this difference might be caused by different non-reported settings or by the underlying retrieval framework.

The statistical analysis (Table 5.8 and Figure 5.13) shows that both the *model* and *RM3* provide a significant contribution to the results. On `R@100`, the *model* has a small effect on the results, while *RM3* provides a strong but negative contribution (as shown on Figure 5.13c). On `nDCG@3`, both the *model* and *RM3* still provide a small effect, and we can observe (Figure 5.13d) that *RM3* does not bring meaningful improvements to the results.

It's possible to observe, on Figure 5.14, that there is a huge performance variance on both metrics, probably related to the inability of this method to always correctly solve the coreference.

mu	RM3	AP	R@100	R@200	nDCG@3
1000	t5-d30-l0.3	0.193598	0.305603	0.391142	0.290831†
1000	t5-d30-l0.5	0.193598	0.305603	0.391142	0.290831
1000	t5-d30-l0.7	0.193598	0.305603	0.391142	0.290831
1000	t5-d10-l0.3	0.192467	0.300844	0.378579	0.288469†
2500	t20-d20-l0.5	0.144803	0.239380	0.304888	0.237636†
		original [Mel+21]			
2500	t20-d20-l0.5	0.1691	—	0.3772	0.2148

Table 5.7: Coref 1 results

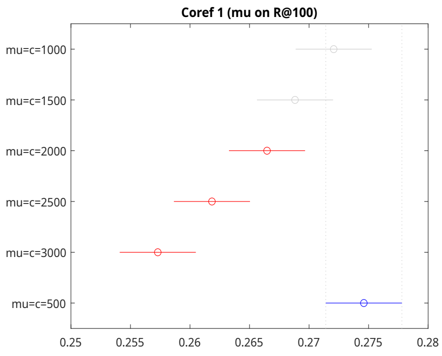
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	91.2871	19	4.80458	4308.84	$< 1e-5$	0.9485
model	0.1537	5	0.03074	27.57	$< 1e-5$	0.0291
prf	3.3352	36	0.09264	83.09	$< 1e-5$	0.399
Error	4.8828	4379	0.00112			
Total	99.6588	4439				

(a) R@100

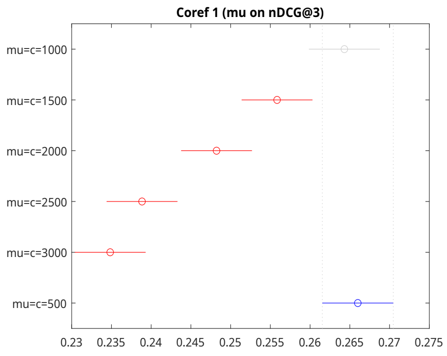
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	74.5224	19	3.92223	1113.73	$< 1e-5$	0.8264
model	0.6226	5	0.12453	35.36	$< 1e-5$	0.0373
prf	0.5887	36	0.01635	4.64	$< 1e-5$	0.0287
Error	15.4216	4379	0.00352			
Total	91.1553	4439				

(b) nDCG@3

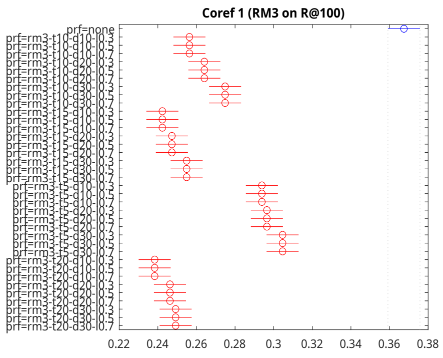
Table 5.8: ANOVA Tables for Coref 1



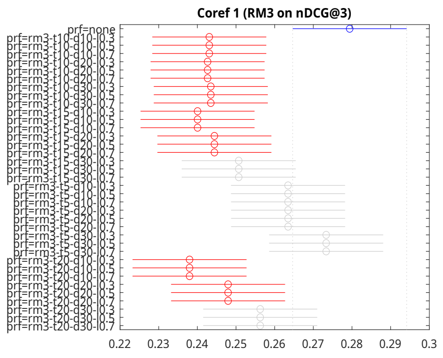
(a) Dirichlet μ on R@100



(b) Dirichlet μ on nDCG@3

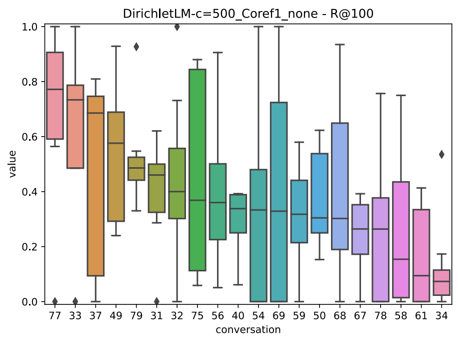


(c) RM3 on R@100

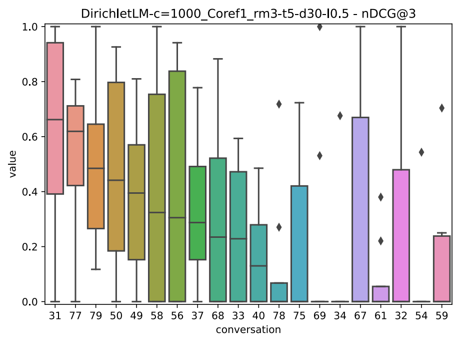


(d) RM3 on nDCG@3

Figure 5.13: Multcompare for Coref 1



(a) R@100



(b) nDCG@3

Figure 5.14: Coref 1 performances

5.2.5 Coref 2

Coref 2 was tested with *Dirichlet language model* and *RM3 pseudo relevance feedback*.

The results are shown on Table 5.9, along with the original [Mel+21] comparison. Our implementation underperform on all metrics when using the original settings, while it achieved a slightly better discounted cumulative gain in our best result. Like with the previous methods, we think this difference might be caused by different non-reported settings or by the underlying retrieval framework.

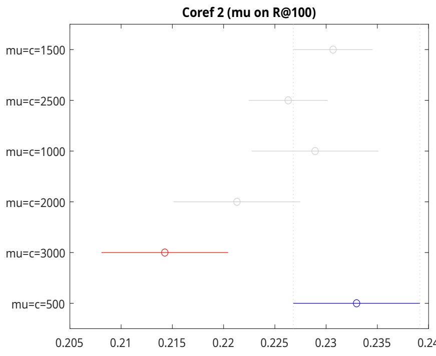
mu	RM3	AP	R@100	R@200	nDCG@3
500	t20-d10-l0.3	0.130272	0.223447	0.273474	0.232829†
500	t20-d10-l0.5	0.130272	0.223447	0.273474	0.232829
500	t20-d10-l0.7	0.130272	0.223447	0.273474	0.232829
500	t20-d20-l0.3	0.130804	0.222762	0.277730	0.228096
2500	t20-d20-l0.5	0.122344	0.209496	0.266959	0.198805†
original [Mel+21]					
2500	t20-d20-l0.5	0.1845	—	0.3818	0.2209

Table 5.9: Coref 2 results

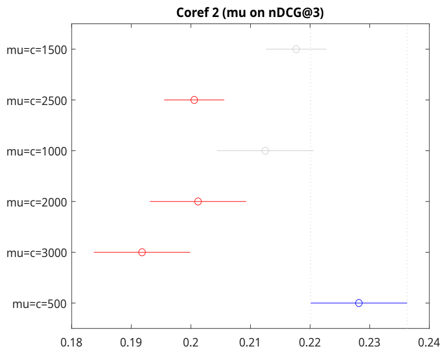
Table 5.10 and Figure 5.15 show the statistical analysis for *Coref 2*. On `R@100`, the *model* has a significant, thought very small effect on the results (in accordance with Figure 5.15a), while *RM3* provides a string but negative contribution.

On `nDCG@3`, *RM3* is not meaningful any more, Figure 5.15d even thought it still has a positive (but very low) ω^2. The *model*, on the other hand, keeps providing a small-effect contribution.

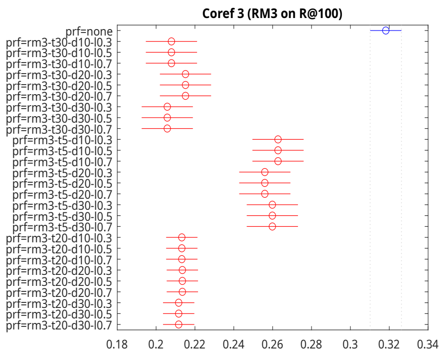
Figure 5.16 shows a performance variation between different conversations that's even bigger compared to the other coreference method (*Coref 1*). We think that's due to the lower effectiveness of *Coref 2* to correctly fill for the context than the previous method.



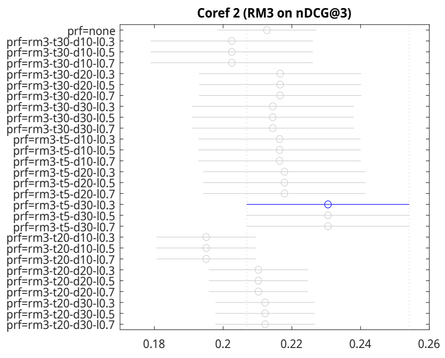
(a) Dirichlet μ on R@100



(b) Dirichlet μ on nDCG@3

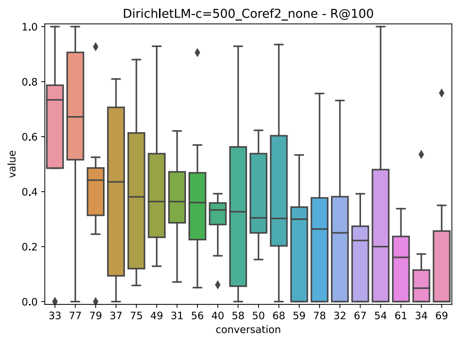


(c) RM3 on R@100

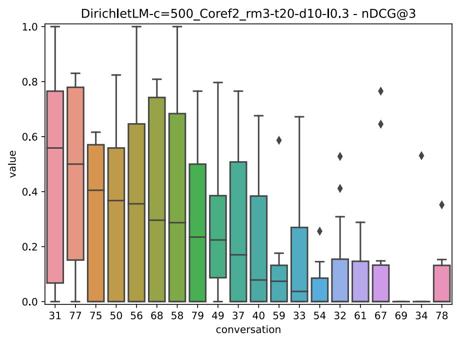


(d) RM3 on nDCG@3

Figure 5.15: Multcompare for Coref 2



(a) R@100



(b) nDCG@3

Figure 5.16: Coref 2 performances

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	35.6758	19	1.87768	1909.53	$<$ 1e-5	0.9497
model	0.0492	5	0.00984	10	$<$ 1e-5	0.0229
prf	1.6908	27	0.06262	63.69	$<$ 1e-5	0.4685
Error	1.8368	1868	0.00098			
Total	39.2581	1919				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	30.35	19	1.59737	506.45	$<$ 1e-5	0.8334
model	0.2374	5	0.04749	15.06	$<$ 1e-5	0.0353
prf	0.1411	27	0.00523	1.66	0.0183	0.0092
Error	5.8917	1868	0.00315			
Total	36.612	1919				

(b) nDCG@3

Table 5.10: ANOVA Tables for Coref 2

5.2.6 HQE

The results for the *HQE* method are presented on Table 5.11, along with the expected value of AP from the original paper [JC19]. We tested using the *BM25* model and different values for r_S, r_Q and θ, but RM3 is not included in this test as it does not improve the results. This is not surprising considering that HQE itself adds a huge amount of keywords to the queries so, adding even more ones will likely make the search less focused.

BM25	rs	rq	theta	AP	R@100	nDCG@3
0.5	15	12	30	0.154524	0.314738	0.221171
0.5	15	12	40	0.149922	0.305166	0.218332
0.3	15	12	30	0.153835	0.312551	0.217699
original paper				0.194000		
chatty-goose				0.210900		0.260600

Table 5.11: HQE results

As we already discussed in a previous chapter, our implementation underperform compared to the original tests and, more surprising, the `chatty-goose` implementation on which we based our `HQERewriter`. Given that the only relevant difference between `HQERewriter` and the `chatty-goose` *HQE* implementation is the retrieval software used underneath (`pyterrier`

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	50.188	19	2.64147	1057.06	$<$ 1e-5	0.8479
rs	0.193	5	0.03859	15.44	$<$ 1e-5	0.0197
rq	0.0155	5	0.0031	1.24	0.2872	—
theta	0.0006	4	0.00016	0.06	0.9923	—
Error	8.911	3566	0.0025			
Total	59.3081	3599				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	
conv	37.4017	19	1.96851	1297.39	$<$ 1e-5	0.8725
rs	0.2871	5	0.05743	37.85	$<$ 1e-5	0.0487
rq	0.0142	5	0.00284	1.87	0.096	—
theta	0.0012	4	0.0003	0.2	0.9403	—
Error	5.4107	3566	0.00152			
Total	43.1149	3599				

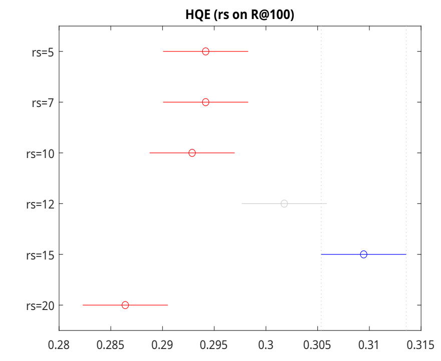
(b) nDCG@3

Table 5.12: ANOVA Tables for HQE

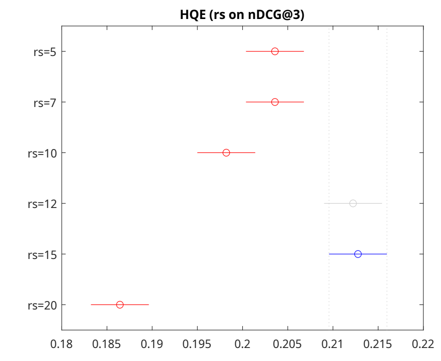
in our case, `pyserini` in the other) it's likely that the culprit for this poor performance is `pyterrier` itself.

Observing the statistical analysis (Table 5.12 and Figure 5.17) for *HQE*, we can observe that neither r_Q nor θ provide a meaningful contribution to the results. The only parameter with a (small) statistically relevant effect is r_S. Considering that r_S is responsible for the identification of conversational keywords, we can conclude that the ability to find the missing context is the most influential for the results.

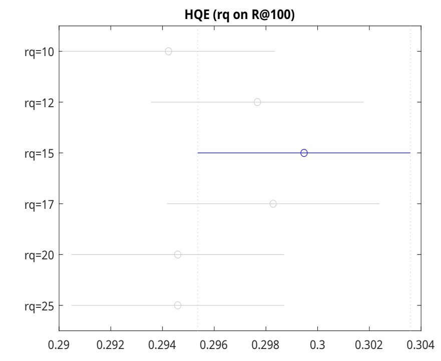
Even with *HQE*, we can see a wide variation of performance in different conversations, as well as a huge variance inside single conversations (Figure 5.18).



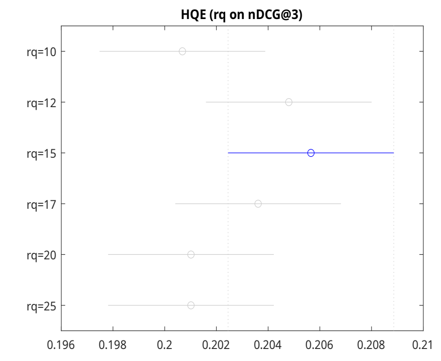
(a) r_S on R@100



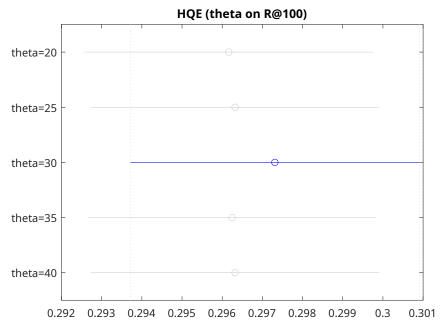
(b) r_S on nDCG@3



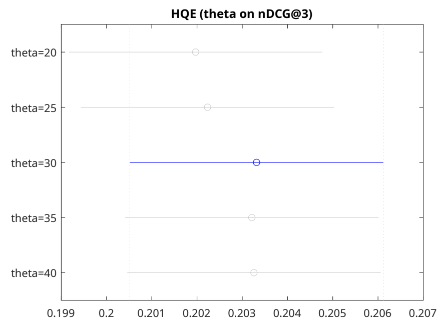
(c) r_Q on R@100



(d) r_Q on nDCG@3

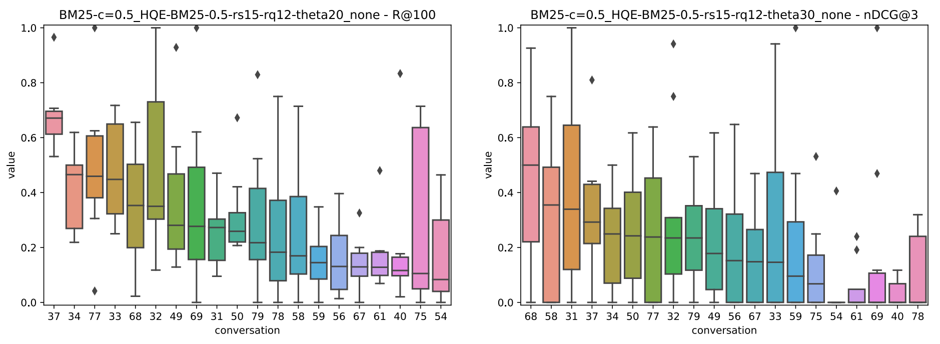


(e) θ on R@100



(f) θ on nDCG@3

Figure 5.17: Multcompare for HQE



(a) R@100 (b) nDCG@3

Figure 5.18: HQE performances

5.2.7 DBPedia

DBPedia was tested using *BM25* with different number of *snippets*, the two type of *frequency* (on index and snippets) and a standard *RM3* configuration. The nature of this rewriting method, which needs to access rate-limited API, make it difficult to complete large batches. Consequently, the tested combinations are less than what we have done for the previous techniques.

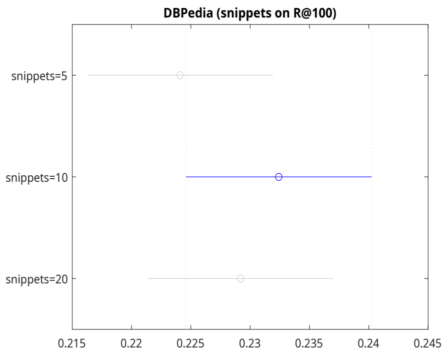
The results for this method are shown on Table 5.13, along with the original [Sam19] one. It's clear that our implementation severely underperforms. We are not sure why it's the case, but considering that the original paper is really undetailed and sometimes unclear on the performed passage, we think our implementation is likely not on-par with the original one.

snippets	freq on	RM3	AP	R@100	nDCG@3
10	snippets	none	0.119261	0.246111	0.169905
20	snippets	none	0.120076	0.249257	0.168225
10	index	t20-d20-l0.5	0.120791	0.239639	0.163024
original [Sam19]			0.173	—	0.234

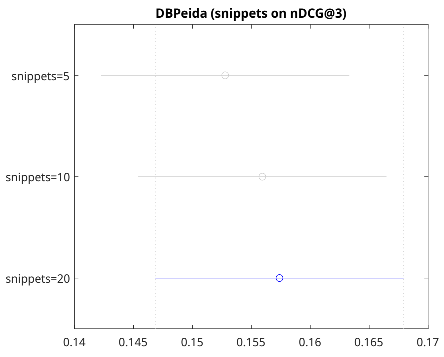
Table 5.13: DBPedia results

Table 5.14 and Figure 5.19 show the statistical analysis for *DBPedia*. As it can be observed, no parameter provides a significant effect on the results, except for a slight disadvantage of *RM3* on `R@100` (Figure 5.19e).

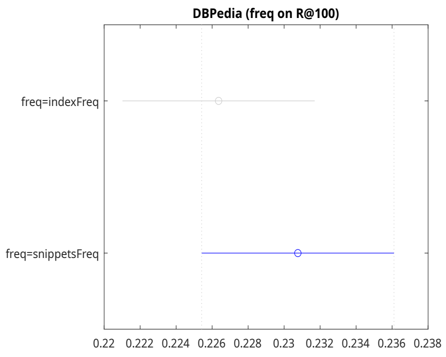
From Figure 5.20, we can see that this method performs poorly (with only a few exceptions) than the previous ones. As previously stated, this is likely something wrong with our implementation due to the unclarity of the original algorithm description.



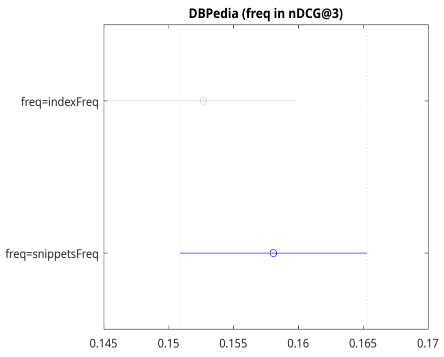
(a) #snippets on R@100



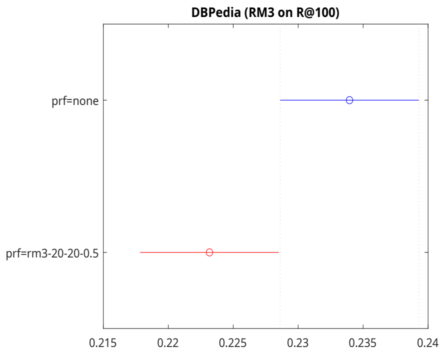
(b) #snippets on nDCG@3



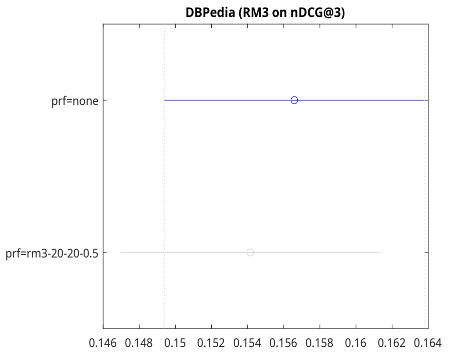
(c) Type of frequency on R@100



(d) Type of frequency on nDCG@3



(e) RM3 on R@100



(f) RM3 on nDCG@3

Figure 5.19: Multcompare for DBPedia

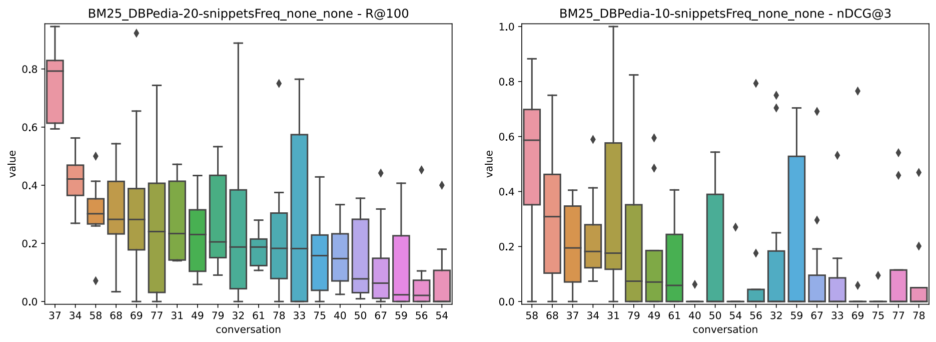
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	4.57977	19	0.24104	135.21	$<$ 1e-5	0.9140
snippets	0.00281	2	0.00141	0.79	0.4556	—
freq	0.00116	1	0.00116	0.65	0.4201	—
prf	0.00697	1	0.00697	3.91	0.0492	0.0120
Error	0.38508	216	0.00178			
Total	4.9758	239				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	1.98997	19	0.10474	32.39	$<$ 1e-5	0.7131
snippets	0.00089	2	0.00044	0.14	0.8718	—
freq	0.00175	1	0.00175	0.54	0.4633	—
prf	0.00036	1	0.00036	0.11	0.7399	—
Error	0.69843	216	0.00323			
Total	2.69139	239				

(b) nDCG@3

Table 5.14: ANOVA Tables for DBPedia



(a) R@100 (b) nDCG@3

Figure 5.20: DBPedia performances

5.2.8 Seen Filter

Seen Filter was tested using the best *Context Query* rewriting settings and different values for the *multiplier* (m) and the *maximum rank* (k).

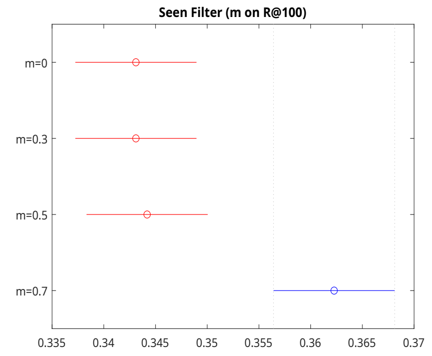
On Table 5.15 are shown the results and the comparison with the rewriter alone (no results' metrics are provided for this method in the original paper [Mar19]). We can observe that the `nDCG@3` value decreased. That means this reranker wasn't able to move the correct documents to the top. However, the recall is slightly higher, so this method is not completely wrong. Given that its performance is hugely dependent on the correctness of the previous queries results, we think the inconstant effectiveness that all examined methods are showing (*Context Query* included; Figure 5.12) prevent this technique to work correctly.

m	k	R@100	nDCG@3
0.7	10	0.390547	0.294418
0	10	0.378536	0.292008
0.3	10	0.378536	0.292008
0.5	10	0.379155	0.292008
Context Query		0.379442	0.311693

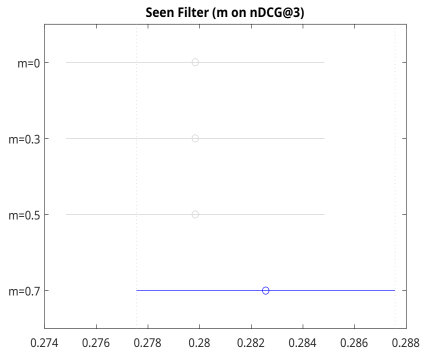
Table 5.15: Seen Filter results

The statistical analysis is displayed onTable 5.16 and Figure 5.21. We can observe that, on `R@100`, both m and k provides a significant contribution, but only k has a big effect on the results. In fact, from Figure 5.21a, is visible that three of the four tested values of m are statistically equivalent. On `nDCG@3` instead, m ceases to be significant, and k, while it's still meaningful, only provide a small contribution to the results. These observations on the significance of m and k on *nDCG@3* are likely connected to the poor performance that we already discussed and are probably originating from the same issues.

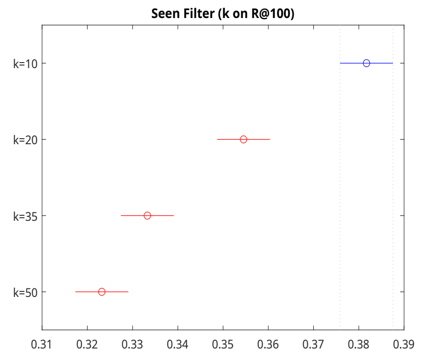
We can observe, from Figure 5.22, that the performance variation inside the same conversation is wider on `nDCG@3`, with some conversation (e.g., 31 or 58) where all possible values are met. Given these results, we ultimately think this method will be better suited as an intermediate step to increase the recall on the first $n << 1000$ documents to feed them to a more computationally heavy reranker.



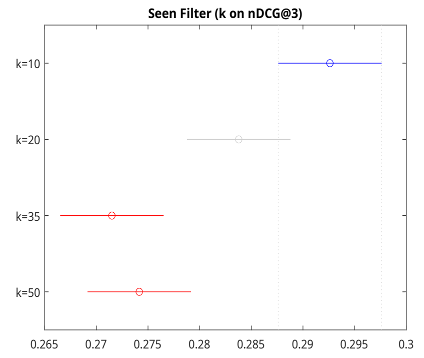
(a) m on R@100



(b) m on nDCG@3

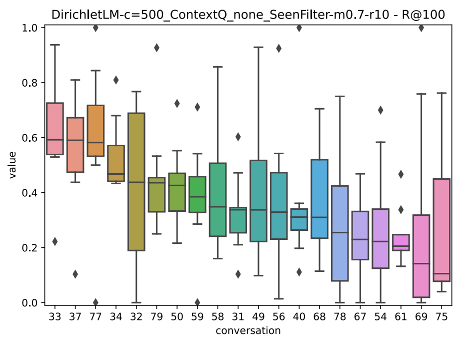


(c) k on R@100

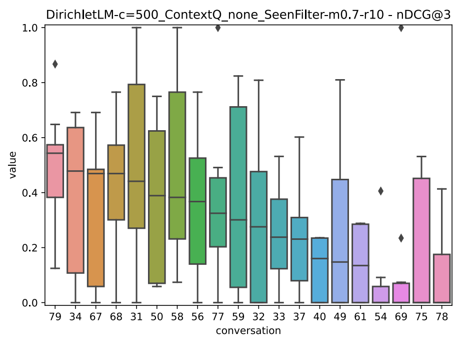


(d) k on nDCG@3

Figure 5.21: Multcompare for Seen Filter



(a) R@100



(b) nDCG@3

Figure 5.22: Seen Filter performances

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	2.34227	19	0.12328	148.15	$<$ 1e-5	0.8973
m	0.02128	3	0.00709	8.52	1.9e-05	0.0659
k	0.16081	3	0.0536	64.42	$<$ 1e-5	0.3729
Error	0.24464	294	0.00083			
Total	2.76899	319				

(a) R@100

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	5.47768	19	0.2883	474.24	$<$ 1e-5	0.9656
m	0.00045	3	0.00015	0.24	0.8651	—
k	0.02229	3	0.00743	12.22	$<$ 1e-5	0.0952
Error	0.17873	294	0.00061			
Total	5.67914	319				

(b) nDCG@3

Table 5.16: ANOVA Tables for Seen Filter

5.2.9 Bottom Up

Bottom Up was tested using the best *Coref 1* rewriting settings and different values for the *multiplier* (m) and the *maximum rank* (k). The choice of a coreference method is justified by the fact that the effectiveness of *Bottom Up* depends on the correctness of the documents returned by the latest utterances in a conversation. Thus, we decided to use *Coref 1* as it's the rewriter that has a higher probability to find the right context, even on the last queries.

Table 5.17 shows the results and the comparison with *Context Query* and *Coref 1* (the original paper [Mar19] uses a different set of collection/topics, so it's not comparable with our tests). We can see that it performs a little worse than both on `nDCG@3`, and worse than *Context Query*, but better than *Coref 1* on `R@100`. Like with *Seen Filter*, the scarce effectiveness is likely linked to the difficulty of filling the missing context by the rewriting methods. Just like *Seen Filter*, this technique might be used as an intermediate step in a more complex pipeline.

Table 5.18 and Figure 5.23 display the statistical analysis for *Bottom Up*. It's possible to observe that, the multiplier m does not provide significant contributions to the results, while the maximum rank k has a strong effect on them. This is reasonable, as changing k changes the documents that are allowed to appear in the other queries' results. Like the previous methods, even *Bottom Up* shows a wide variation between conversations and a huge variance inside them (Figure 5.24).

m	k	R@100	nDCG@3
0	10	0.360641	0.284027
0.3	10	0.360988	0.284027
0.5	10	0.360988	0.284027
0.7	10	0.364185	0.284027
Context Query		0.379442	0.311693
Coref 1		0.305603	0.290831

Table 5.17: Bottom Up results

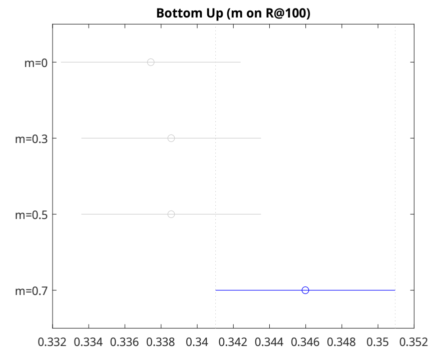
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	4.21762	19	0.22198	370.89	$< 1e\text{-}5$	0.9565
m	0.00372	3	0.00124	2.07	0.1042	—
k	0.07422	3	0.02474	41.33	$< 1e\text{-}5$	0.2744
Error	0.17596	294	0.0006			
Total	4.47151	319				

(a) R@100

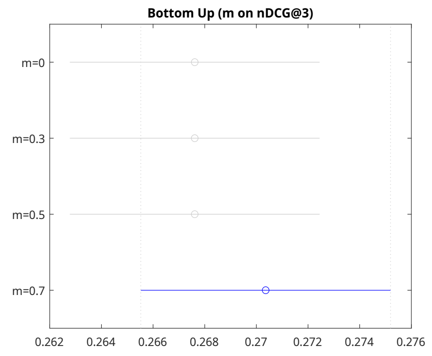
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	6.45545	19	0.33976	599.41	$< 1e\text{-}5$	0.9726
m	0.00045	3	0.00015	0.27	0.8497	—
k	0.03343	3	0.01114	19.66	$< 1e\text{-}5$	0.1489
Error	0.16665	294	0.00057			
Total	6.65598	319				

(b) nDCG@3

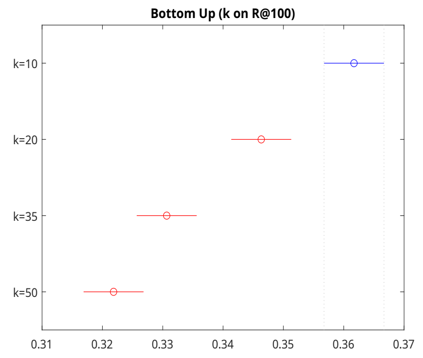
Table 5.18: ANOVA Tables for Seen Filter



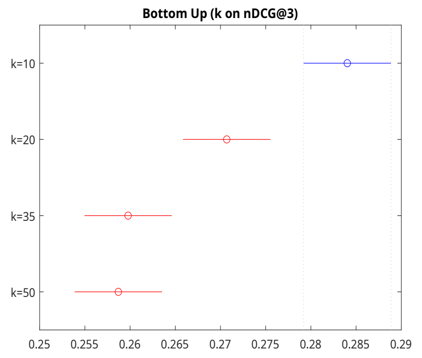
(a) m on R@100



(b) m on nDCG@3

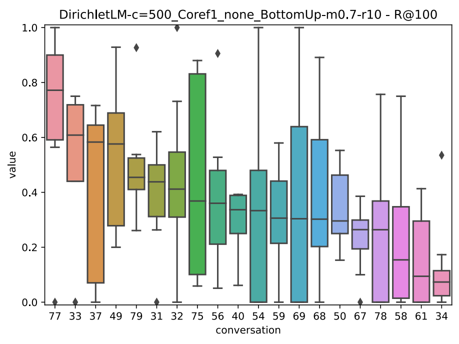


(c) k on R@100

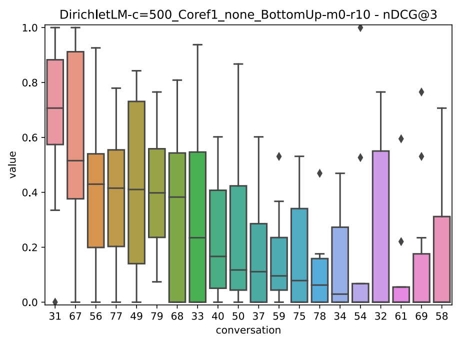


(d) k on nDCG@3

Figure 5.23: Multcompare for Bottom Up



(a) R@100



(b) nDCG@3

Figure 5.24: Bottom Up performances

5.2.10 HAE

HAE was tested on top of *HQE*, using different values for λ. Given the computational heaviness of the process, the number of values tested was less than what we have done for other methods, and we fixed the cut-off k to 100 to speed up the computation. Consequently, we're only considering `nDCG@3` for this method, as the recall at 100 won't change.

Table 5.19 shows the results for *HAE*, along with the *HQE* best one and the original results. We can see that this method significantly increase the discounted cumulative gain, in respect to the rewriter alone, but perform very poorly in comparison with the original results.

λ	k	AP	nDCG@3
1.5	100	0.158146	$0.247146^{\dagger}$
2	100	0.158030	0.246307
3	100	0.157218	0.244180
4	100	0.156882	0.244180
5	100	0.156873	$0.241368^{\dagger}$
HQE		0.139173	0.199007
TREC CAsT		0.267	0.436

Table 5.19: HAE results

Observing the statistical analysis on Table 5.20 and Figure 5.25, we can notice that the λ parameter is only slightly significant and that there's still a huge variation inside and outside different conversations (Figure 5.26).

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	1.57462	19	0.08287	2077.19	$< 1e\text{-}5$	0.9975
lambda	0.0004	4	0.0001	2.53	0.0473	0.0577
Error	0.00303	76	0.00004			
Total	1.57805	99				

Table 5.20: ANOVA Table for HAE (nDCG@3)

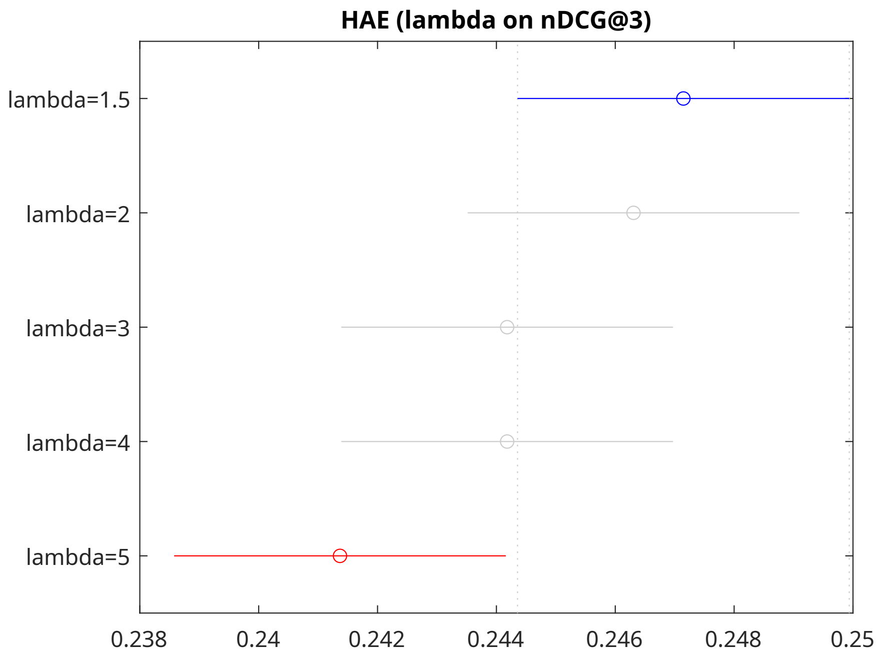


Figure 5.25: Multcompare HAE (nDCG@3)

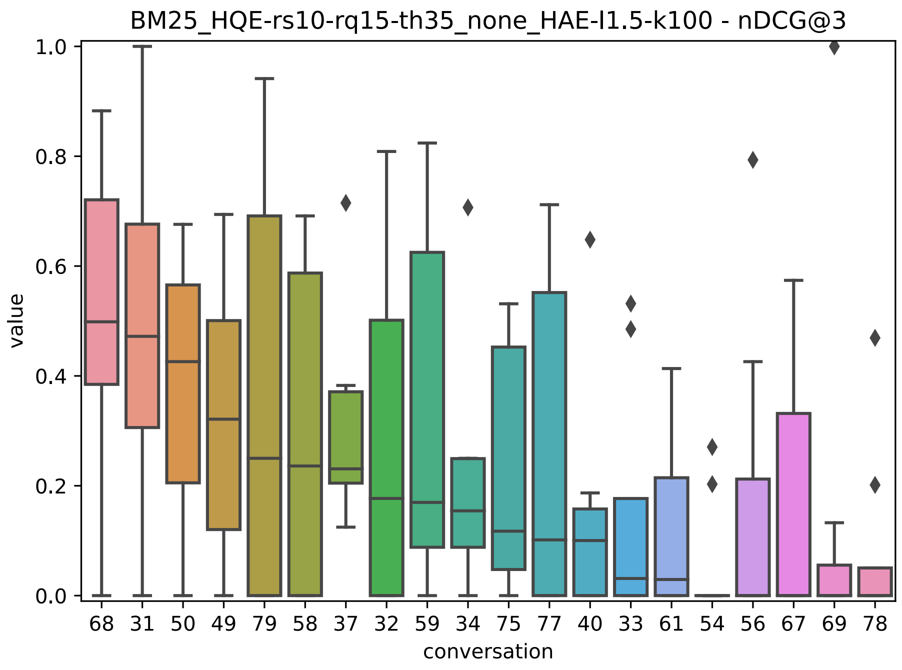


Figure 5.26: HAE performance (nDCG@3)

5.2.11 Sub Index

Sub Index is the search strategy implemented in `SubIndexPipelineFactory` (Section 4.1.3). It was tested using *BM25* and *Dirichlet* (DLM) as base-model, *Dirichlet* as sub-model, different sub-index sizes, different rewriters and *RM3* for both the base search and the subsequent ones.

Table 5.21 shows the results. Unfortunately, compared with *Coref 1* (the rewriter with which this technique achieved the best score), this technique slightly underperforms.

base model	index size	sub model	base RM3	R@100	nDCG@3
BM25	50000	DLM	t20-d20-l0.5	0.299397	0.255705
BM25	50000	DLM	t20-d20-l0.5	0.299397	0.255705
BM25	100000	DLM	none	0.308830	0.252840
BM25	100000	DLM	none	0.308830	0.252840
Coref 1				0.305603	0.290831

Table 5.21: Sub Index results

The statistical analysis (Table 5.22 and Figures 5.27 and 5.28) shows that every parameter (except for *rerunFirst*) provides a significant contribution. The biggest contribution (excluding the conversations) is given by *rm3Sub* (especially on `R@100`), meaning that performing RM3 on the topic-focused documents returned by the first-utterance search can refine the result significantly better. The *index size*, that we had thought would have been the most relevant, has a strong effect only on `R@100` (but we think this might be fixed using a good reranker). Figures 5.27e and 5.27f show that using a rewriter might still be a good choice even when using a sub-index, as it gives more stable results.

The overall performance (Figure 5.29) shows a very high variance of results inside the conversations. This might be explained if the documents needed to answer specific queries were left out of the sub-index, degrading the metrics.

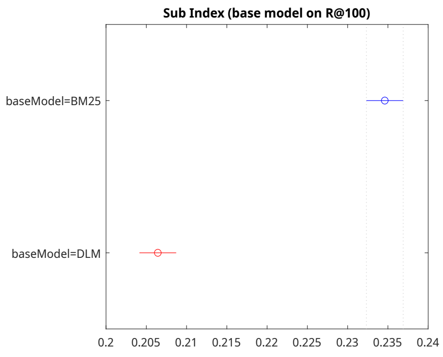
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	28.3203	19	1.49054	343.4	$< 1e-5$	0.6689
baseModel	0.6354	1	0.63536	146.38	$< 1e-5$	0.0432
indexSize	1.9468	4	0.48671	112.13	$< 1e-5$	0.1213
rerunFirst	0	1	0	0	1	—
rewriter	0.1277	2	0.06383	14.71	$< 1e-5$	0.0084
rm3Base	0.8283	1	0.82829	190.83	$< 1e-5$	0.0557
rm3Sub	2.2754	1	2.27536	524.21	$< 1e-5$	0.1398
Error	13.8464	3190	0.00434			
Total	47.9829	3219				

(a) R@100

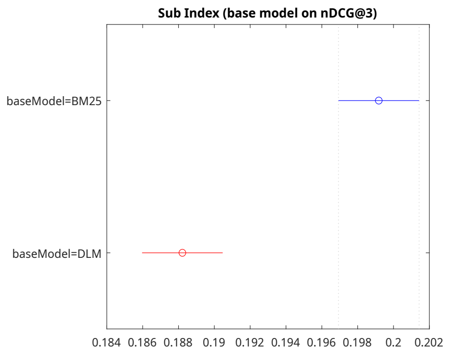
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	31.9899	19	1.68368	399.36	$< 1e-5$	0.7015
baseModel	0.0961	1	0.09612	22.8	$< 1e-5$	0.0067
indexSize	0.7218	4	0.18044	42.8	$< 1e-5$	0.0494
rerunFirst	0	1	0	0	1	—
rewriter	1.2873	2	0.64365	152.67	$< 1e-5$	0.0861
rm3Base	0.1735	1	0.17355	41.16	$< 1e-5$	0.0123
rm3Sub	1.4005	1	1.40049	332.19	$< 1e-5$	0.0933
Error	13.449	3190	0.00422			
Total	49.1288	3219				

(b) nDCG@3

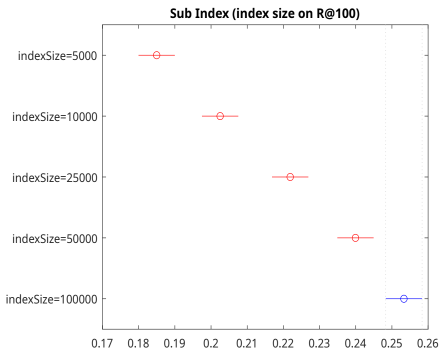
Table 5.22: ANOVA Tables for Sub Index



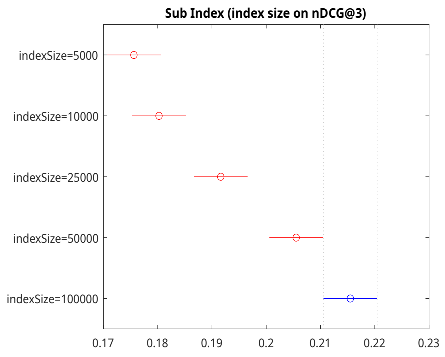
(a) base model (R@100)



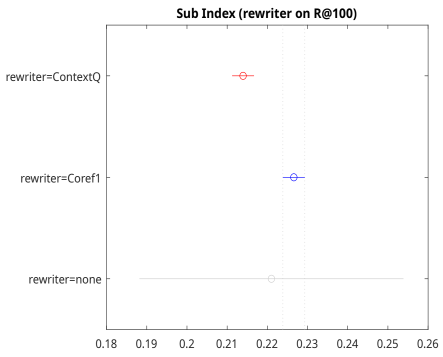
(b) base model (nDCG@3)



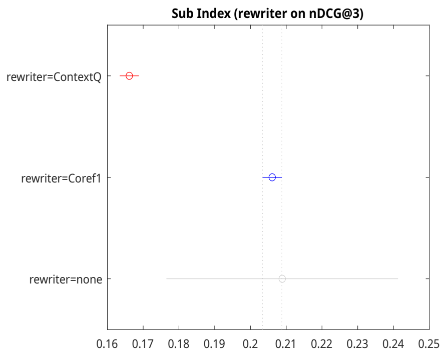
(c) index size (R@100)



(d) index size (nDCG@3)

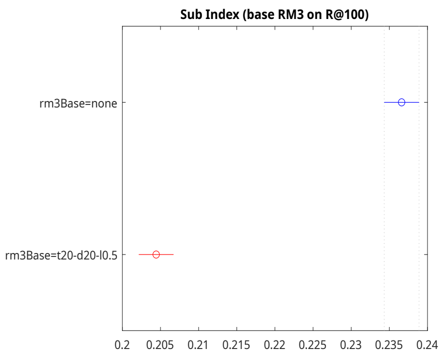


(e) rewriter (R@100)

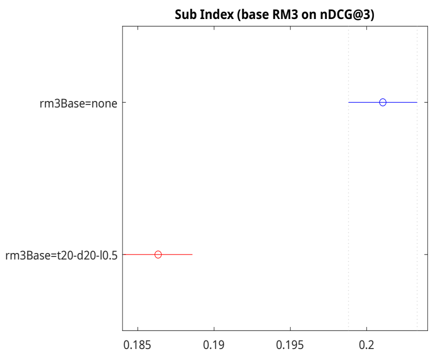


(f) rewriter (nDCG@3)

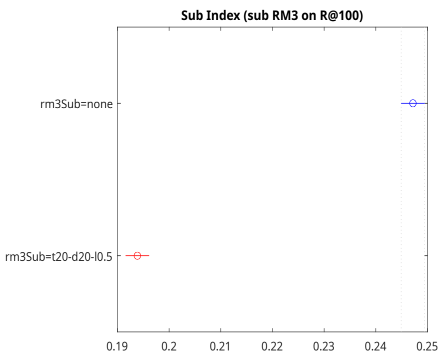
Figure 5.27: Multcompare for Sub Index (A)



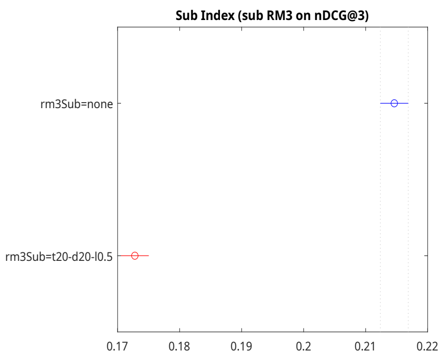
(a) base RM3 (R@100)



(b) base RM3 (nDCG@3)

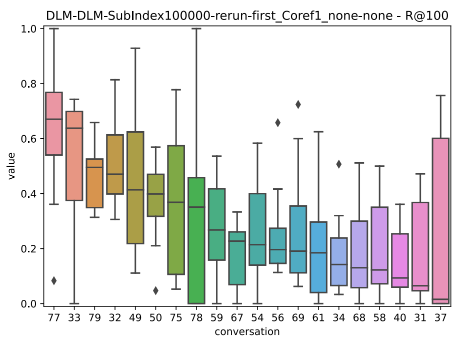


(c) sub RM3 (R@100)

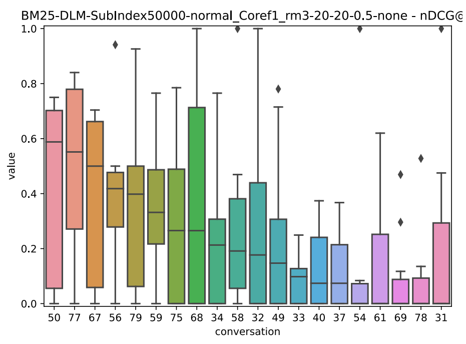


(d) sub RM3 (nDCG@3)

Figure 5.28: Multcompare for Sub Index



(a) R@100



(b) nDCG@3

Figure 5.29: Sub Index performances

5.2.12 Methods Comparison

In this section, we compare the best run for every method by *nDCG@3* and the best run for every method by *R@100*. Table 5.23 shows the result metrics for these runs. We can observe that the first two positions are disputed by *First Query* and *Context Query*. *Seen Filter* keeps the third position in both cases, meaning that it's still a good middle ground. It's interesting to notice that the coreference methods are surpassed by simpler ones based on concatenation, suggesting that coreference resolution techniques are still inadequate for the task.

method	nDCG@3
FirstQuery	0.327624
ContextQuery	0.311693†
SeenFilter	0.294418
Coref1	0.290831
BottomUp	0.284027
ConcatQuery	0.277390
SubIndex	0.255705†
HAE	0.247146
Coref2	0.232829
HQE	0.221171†
DBPedia	0.169905†

(a) nDCG@3

method	R@100
ContextQuery	0.425602
FirstQuery	0.408919†
SeenFilter	0.390547
ConcatQuery	0.380579
Coref1	0.379368
BottomUp	0.364185
HAE	0.354257
Coref2	0.332654†
HQE	0.315918†
SubIndex	0.312226
DBPedia	0.249257†

(b) R@100

Table 5.23: Methods comparison

The statistical analysis (Table 5.24 and Figure 5.30) shows that the majority of the tested methods are statistically similar. We think this might depend on the difficulty of the *conversational searching* task itself which make it challenging to find effective techniques.

Table 5.25 shows the `nDCG@3` difference (in percent) from the original methods' results and our implementations' ones. When applicable, it shows both the difference using the (supposed) same settings of the original work and the difference with our best results.

Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	ω^2
conv	2.50478	19	0.13183	15.8	$< 1e\text{-}5$	0.5611
method	0.5104	10	0.05104	6.12	$< 1e\text{-}5$	0.1888
Error	1.58486	190	0.00834			
Total	4.60004	219				

(a) R@100

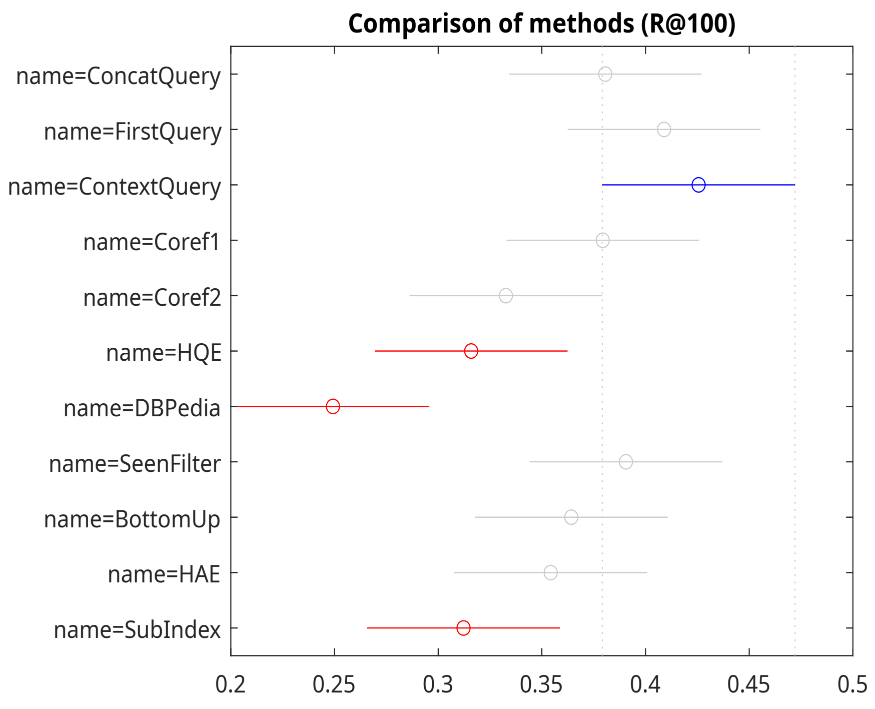
Source	Sum Sq.	d.f.	Mean Sq.	F	p-value	
conv	2.85233	19	0.15012	18.45	$< 1e\text{-}5$	0.6011
method	0.41108	10	0.04111	5.05	$< 1e\text{-}5$	0.1555
Error	1.54632	190	0.00814			
Total	4.80972	219				

(b) nDCG@3

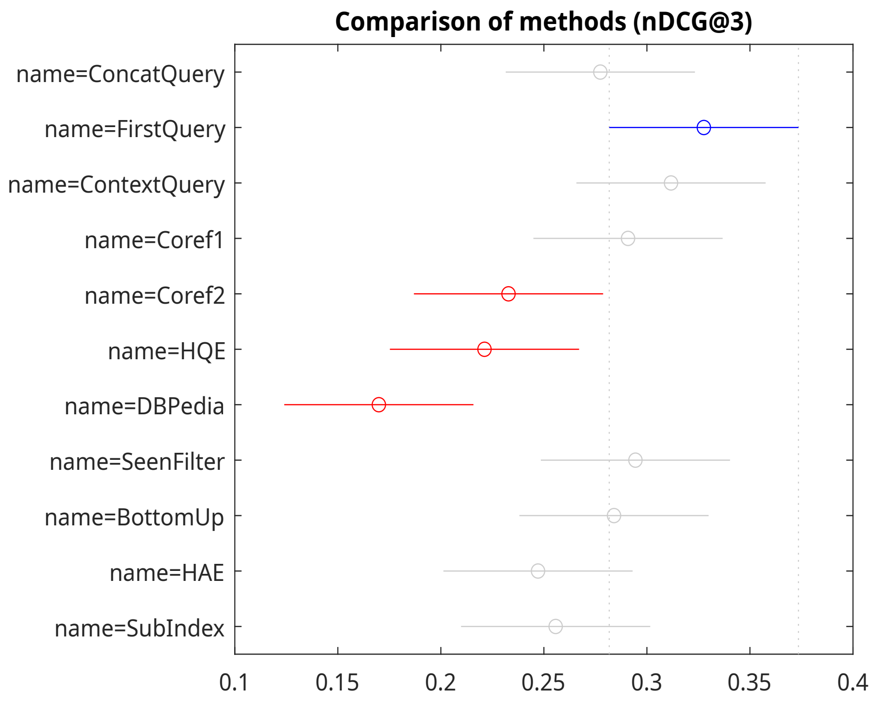
Table 5.24: Comparison of Methods (ANOVA Tables)

method	same sett.	best score
First Query	− 8.8%	+ 23.0%
Context Query	+ 11.5%	+ 34.6%
Coref 1	+ 10.6%	+ 35.4%
Coref 2	− 10.0%	+ 5.4%
HQE		− 15.1%
DBPedia		− 27.4%
HAE		− 43.3%

Table 5.25: nDCG@3 score difference



(a) R@100



(b) nDCG@3

Figure 5.30: Comparison of Methods (multcompare)

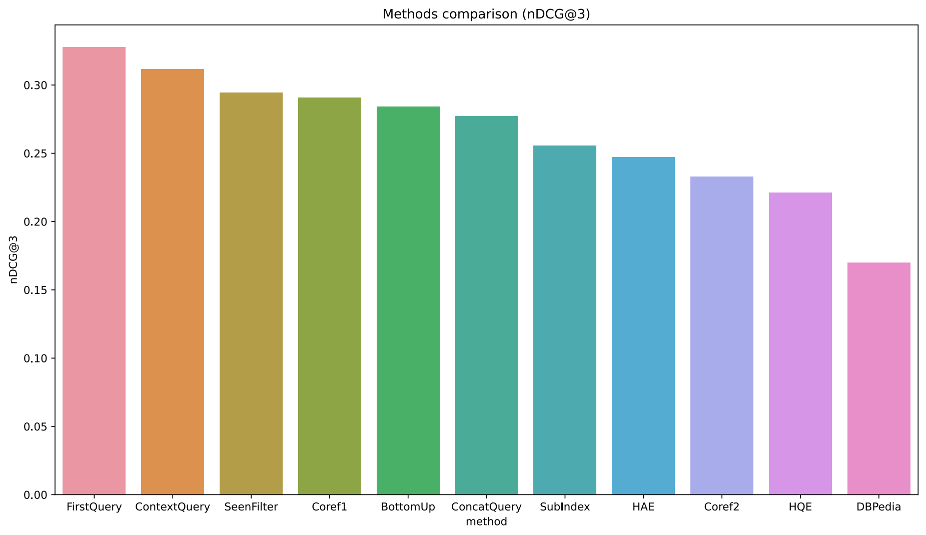


Figure 5.31: Methods comparison (nDCG@3)

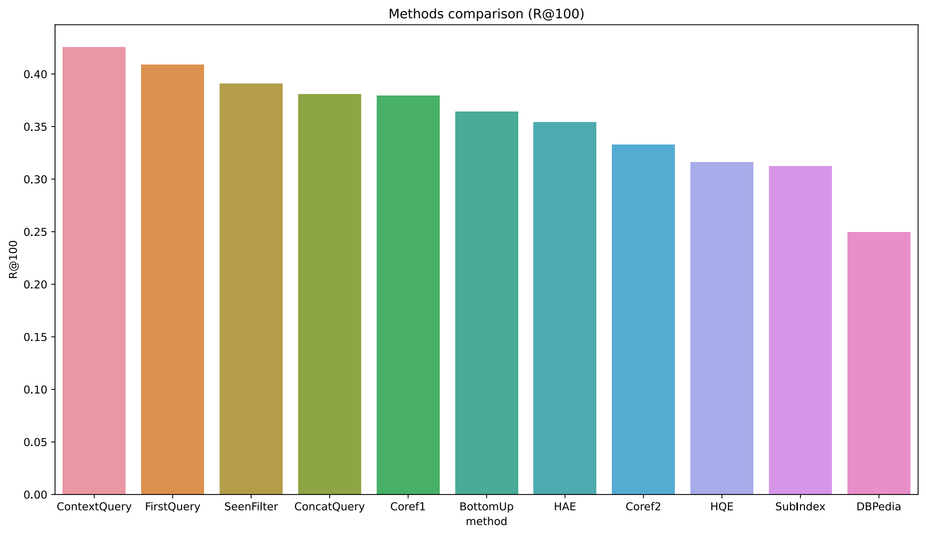


Figure 5.32: Methods comparison (R@100)

Chapter 6

Conclusion And Future Works

In this work, we explored the topic of *Conversational Search*, studying several state-of-the-art algorithms, focusing on their implementability. We presented them with a summarized but complete description, discussing the underlying ideas and providing examples. We discussed the presence of unclear or implementation-dependent passages and how we intended to overcome them. We implemented said methods, providing alternative options for dubious parts and motivating our choices.

We conducted extensive tests with different parametrizations to compare the techniques' performances against the original works and the other methods. We used a combination of performance metrics, charts and statistical analysis to provide a sensible comparison and discuss the influence of the various parameters on the overall results. We analysed the influence of commonly used techniques, like *RM3 Pseudo Relevance Feedback* or *Coreference Resolution*, on a conversational context, justifying their effectiveness with reproducible result data. We discussed the success and failure of our implementations, supporting them with data and motivating the results, drawing a connection with their implementability.

We developed a *conversational retrieval framework* based on `pyterrier`, focused on modularity, extensibility, and reproducibility, that includes said algorithms' implementations and allows others to remake our tests. We discussed our framework architecture, its relation with the underlying retrieval system, and the integration with the implemented algorithms.

Future Works

Future works might more deeply research the reason for the underperforming algorithms and try to replicate the results with another retrieval system. Furthermore, it would be interesting to test more complex pipelines employing different combinations of algorithms. Exploring more methodology, especially on the *reranking* side, or more generally, expanding the methods' library would be beneficial and would allow achieving more comprehensive comparison. The framework might be extended, providing additional ways to combine methods or alternative search strategies.

Bibliography

[Bus45] Vannevar Bush. "As We May Think". In: *Atlantic Monthly* 176 (July 1945), pp. 101–108.

[Luh57] H. P. Luhn. "A Statistical Approach to Mechanized Encoding and Searching of Literary Information". In: *IBM Journal of Research and Development* 1.4 (1957), pp. 309–317. DOI: `10.1147/rd.14.0309`.

[Fai58] R. A. Fairthorne. "Automatic Retrieval of Recorded Information". In: *The Computer Journal* 1.1 (Jan. 1958), pp. 36–41. ISSN: 0010-4620. DOI: `10.1093/comjnl/1.1.36`. eprint: `https://academic.oup.com/comjnl/article-pdf/1/1/36/1063042/010036.pdf`. URL: `https://doi.org/10.1093/comjnl/1.1.36`.

[MK60] M. E. Maron and J. L. Kuhns. "On Relevance, Probabilistic Indexing and Information Retrieval". In: *J. ACM* 7.3 (July 1960), pp. 216–244. ISSN: 0004-5411. DOI: `10.1145/321033.321035`. URL: `https://doi.org/10.1145/321033.321035`.

[SWY75] Gerard Salton, A. Wong, and Chung-Shu Yang. "A vector space model for automatic indexing". In: *Commun. ACM* 18 (1975), pp. 613–620.

[Rob77] Stephen Robertson. "The Probability Ranking Principle in IR". In: *Journal of Documentation* 33 (Dec. 1977), pp. 294–304. DOI: `10.1108/eb026647`.

[Bel80] Nicholas J Belkin. "Anomalous states of knowledge as a basis for information retrieval". In: *Canadian journal of information science* 5.1 (1980), pp. 133–143.

[CT87] W. B. Croft and R. H. Thompson. "I3R: A new approach to the design of document retrieval systems". In: *Journal of the American Society for Information Science* 38.6 (1987), pp. 389–404. DOI: `https://doi.org/10.1002/(SICI)1097-4571(198711)38:6<389::AID-ASI1>3.0.CO;2-4`.

[Bat89] Marcia J. Bates. “The design of browsing and berrypicking techniques for the online search interface”. In: *Online Review* 13(5) (1989), pp. 407–424.

[TC90] Howard R. Turtle and W. Bruce Croft. “Inference Networks for Document Retrieval”. In: *SIGIR*. 1990, pp. 1–24. URL: `https://doi.org/10.1145/96749.98006`.

[Bel+95] Nicholas J Belkin, Colleen Cool, Adelheit Stein, and Ulrich Thiel. “Cases, scripts, and information-seeking strategies: On the design of interactive information retrieval systems”. In: *Expert systems with applications* 9.3 (1995), pp. 379–395.

[AGH99] J.E. Allen, Curry Guinn, and E. Horvtz. “Mixed-initiative interaction”. In: *Intelligent Systems and their Applications, IEEE* 14 (Oct. 1999), pp. 14–23. DOI: `10.1109/5254.796083`.

[Nor99] Ragnar Nordlie. ““User Revealment”—a Comparison of Initial Queries and Ensuing Question Development in Online Searching and in Human Reference Interactions”. In: *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’99. Berkeley, California, USA: Association for Computing Machinery, 1999, pp. 11–18. ISBN: 1581130961. DOI: `10.1145/312624.312618`. URL: `https://doi.org/10.1145/312624.312618`.

[SG01] Amit Singhal and I. Google. “Modern Information Retrieval: A Brief Overview”. In: *IEEE Data Engineering Bulletin* 24 (Jan. 2001).

[Jal+04] Nasreen Jaleel, James Allan, W. Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Mark Smucker, and Courtney Wade. “UMass at TREC 2004: Novelty and hard”. In: Jan. 2004.

[CRH16] Konstantina Christakopoulou, Filip Radlinski, and Katja Hofmann. “Towards Conversational Recommender Systems”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 815–824. ISBN: 9781450342322. DOI: `10.1145/2939672.2939746`. URL: `https://doi.org/10.1145/2939672.2939746`.

[RC17] Filip Radlinski and Nick Craswell. “A Theoretical Framework for Conversational Search”. In: *Proceedings of the 2017 Conference on Conference Human Information Interaction and Retrieval*. CHIIR ’17. Oslo, Norway: Association for Computing Machinery, 2017, pp. 117–126. ISBN: 9781450346771. DOI: 10.1145/3020165.3020183. URL: https://doi.org/10.1145/3020165.3020183.

[Vty+17] Alexandra Vtyurina, Denis Savenkov, Eugene Agichtein, and Charles L. A. Clarke. “Exploring Conversational Search With Humans, Assistants, and Wizards”. In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pp. 2187–2193. ISBN: 9781450346566. DOI: 10.1145/3027063.3053175. URL: https://doi.org/10.1145/3027063.3053175.

[Dev+18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[Die+18] Laura Dietz, Ben Gamari, Jeff Dalton, and Nick Craswell. “Trec complex answer retrieval overview”. In: *Proceedings of Text REtrieval Conference (TREC)* (2018).

[Gar+18] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. *AllenNLP: A Deep Semantic Natural Language Processing Platform*. 2018. arXiv: 1803.07640 [cs.CL].

[LHZ18] Kenton Lee, Luheng He, and Luke Zettlemoyer. “Higher-Order Coreference Resolution with Coarse-to-Fine Inference”. In: *NAACL*. 2018.

[QCJ18] Massimo Quadrana, Paolo Cremonesi, and Dietmar Jannach. “Sequence-Aware Recommender Systems”. In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3190616. URL: https://doi.org/10.1145/3190616.

[Cla+19] Leigh Clark, Philip Doyle, Diego Garaialde, Emer Gilmartin, Stephan Schlögl, Jens Edlund, Matthew Aylett, João Cabral, Cosmin Munteanu, Justin Edwards, and Benjamin R Cowan. “The State of Speech in HCI: Trends, Themes and Challenges”. In: *Interacting with Computers* 31.4 (June 2019), pp. 349–371. ISSN: 1873-7951. DOI: 10.1093/iwc/iwz016. URL: http://dx.doi.org/10.1093/iwc/iwz016.

[Cla19] Charles L. A. Clarke. “WaterlooClarke at the TREC 2019 Conversational Assistant Track”. In: *Proceedings of the Twenty-Eighth Text REtrieval Conference, TREC 2019, Gaithersburg, Maryland,* *USA, November 13-15, 2019*. Ed. by Ellen M. Voorhees and Angela Ellis. Vol. 1250. NIST Special Publication. National Institute of Standards and Technology (NIST), 2019. URL: `https://trec.nist.gov/pubs/trec28/papers/WaterlooClarke.C.pdf`.

[GAS19] Ana Valeria Gonzalez, Isabelle Augenstein, and Anders Søgaard. "Retrieval-based goal-oriented dialogue generation". In: *arXiv preprint arXiv:1909.13717* (2019).

[JC19] Sheng-Chieh Lin Jheng-Hong Yang and Ming-Feng Tsai Chuan-Ju Wang Jimmy Lin. *Query and Answer Expansion from Conversation History*. 2019.

[KC19] Vaibhav Kumar and Jamie Callan. "A Step towards Context Identification for Conversational Search." In: *TREC*. 2019.

[Mar19] Mahsa S. Shahshahani Jaap Kamps Maarten Marx. *University of Amsterdam at the TREC 2019 Complex Answer Retrieval Track*. 2019.

[Rís+19] Esteban Andrés Ríssola, Manajit Chakraborty, Fabio A. Crestani, and Mohammad Aliannejadi. "Predicting Relevant Conversation Turns for Improved Retrieval in Multi-Turn Conversational Search". In: *TREC*. 2019.

[Sam19] Andrew Yates Samarth Mehrotra. *Incoporating Query Context into a BERT Re-ranker*. 2019.

[Tri19] Johanne Trippas. "Spoken conversational search: audio-only interactive information retrieval". In: *ACM SIGIR Forum* 53 (Dec. 2019), pp. 106–107. DOI: `10.1145/3458553.3458570`.

[Tri+19] Johanne R. Trippas, Damiano Spina, Paul Thomas, Mark Sanderson, Hideo Joho, and Lawrence Cavedon. *Towards a Model for Spoken Conversational Search*. 2019. arXiv: `1910.13166 [cs.IR]`.

[WY19a] Wei Wu and Rui Yan. "Deep Chit-Chat: Deep Learning for Chatbots". In: *Companion Proceedings of The 2019 World Wide Web Conference*. WWW '19. San Francisco, USA: Association for Computing Machinery, 2019, p. 1329. ISBN: 9781450366755. DOI: `10.1145/3308560.3320084`. URL: `https://doi.org/10.1145/3308560.3320084`.

[WY19b] Wei Wu and Rui Yan. “Deep Chit-Chat: Deep Learning for Chatbots”. In: *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR’19. Paris, France: Association for Computing Machinery, 2019, pp. 1413–1414. ISBN: 9781450361729. DOI: 10.1145/3331184.3331388. URL: https://doi.org/10.1145/3331184.3331388.

[DXC20] Jeffrey Dalton, Chenyan Xiong, and Jamie Callan. *TREC CAsT 2019: The Conversational Assistance Track Overview.* 2020. arXiv: 2003.13624 [cs.IR].

[JMB20] Dietmar Jannach, Bamshad Mobasher, and Shlomo Berkovsky. “Research directions in session-based and sequential recommendation”. In: *User Modeling and User-Adapted Interaction* 30.4 (2020), pp. 609–616. DOI: 10.1007/s11257-020-09274-4.

[KRW20] Magdalena Kaiser, Rishiraj Saha Roy, and Gerhard Weikum. *CROWN: Conversational Passage Ranking by Reasoning over Word Networks.* 2020. arXiv: 1911.02850 [cs.IR].

[Suk+20] Rhea Sukthanker, Soujanya Poria, Erik Cambria, and Ramkumar Thirunavukarasu. “Anaphora and coreference resolution: A review”. In: *Information Fusion* 59 (2020), pp. 139–162. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2020.01.010. URL: https://www.sciencedirect.com/science/article/pii/S1566253519303677.

[Tav20] Leila Tavakoli. “Generating Clarifying Questions in Conversational Search Systems”. In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management.* CIKM ’20. Virtual Event, Ireland: Association for Computing Machinery, 2020, pp. 3253–3256. ISBN: 9781450368599. DOI: 10.1145/3340531.3418513. URL: https://doi.org/10.1145/3340531.3418513.

[Tri+20] Johanne Trippas, Damiano Spina, Paul Thomas, Mark Sanderson, Hideo Joho, and Lawrence Cavedon. “Towards a model for spoken conversational search”. In: *Information Processing & Management* 57 (Mar. 2020), p. 102162. DOI: 10.1016/j.ipm.2019.102162.

[Mel+21] Ida Mele, Cristina Ioana Muntean, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, and Ophir Frieder. “Adaptive utterance rewriting for conversational search”. In: *Information Processing & Management* 58.6 (2021), p. 102682. ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2021.102682.

URL: https://www.sciencedirect.com/science/article/pii/S0306457321001679.

[Ren+21] Pengjie Ren, Zhumin Chen, Zhaochun Ren, Evangelos Kanoulas, Christof Monz, and Maarten de Rijke. *Conversations with Search Engines: SERP-based Conversational Response Generation*. 2021. arXiv: 2004.14162 [cs.IR].

[Jan+22] Dietmar Jannach, Ahtsham Manzoor, Wanling Cai, and Li Chen. "A Survey on Conversational Recommender Systems". In: *ACM Computing Surveys* 54.5 (June 2022), pp. 1–36. ISSN: 1557-7341. DOI: 10.1145/3453154. URL: http://dx.doi.org/10.1145/3453154.

[Zam+22] Hamed Zamani, Johanne R. Trippas, Jeff Dalton, and Filip Radlinski. *Conversational Information Seeking*. 2022. arXiv: 2201.08808 [cs.IR].