PyFinder

fast, intelligent search through Python's built-in documentation

by Dmitry Beresnev, Vsevolod Klyushev & Nikita Yaneev

01

Data Scraping

Data Scraping

`random.randrange(stop)`
`random.randrange(start, stop[, step])`

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the `range()` function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

> *Changed in version 3.2:* `randrange()` is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

> *Changed in version 3.12:* Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a `TypeError`.

Random.randrange description from Python documentation

Data Scraping

```
FUNCTION

random.randrange FROM random

PARAMETERS
start, stop, step

DESCRIPTION
Return a randomly selected element from range(start, stop, step).
This is roughly equivalent to choice(range(start, stop, step)) but
supports arbitrarily large ranges and is optimized for common cases.
The positional argument pattern matches the range() function.
Keyword arguments should not be used because they can be interpreted
in unexpected ways. For example randrange(start=100) is interpreted
as randrange(0, 100, 1).
Changed in version 3.2: randrange() is more sophisticated about producing equally distributed
values.  Formerly it used a style like int(random()*n) which could produce
slightly uneven distributions.
Changed in version 3.12: Automatic conversion of non-integer types is no longer supported.
Calls such as randrange(10.0) and randrange(Fraction(10, 1))
now raise a TypeError.
```

Resulted Random.randrange file

02

Overview

Application Modes

Search

Quickly find relevant documentation using one of two indexing approaches

Chat

Ask natural language questions and get intelligent sourced answers via RAG powered by LLMs

Technology stack

Backend

Python 3.12,
FastAPI

Libraries

NLTK,
scikit-learn,
PyTorch,
Transformers, g4f,
pybloom-live

Frontend

Next.js – React
framework for UI

Used Models

Hosted LLMs

qwen-2-72b, qwen-2.5-coder-32b, gpt-4o, wizardlm-2-7b, wizardlm-2-8x22b, dolphin-2.6, dolphin-2.9, glm-4, evil, command-r

Local LLMs

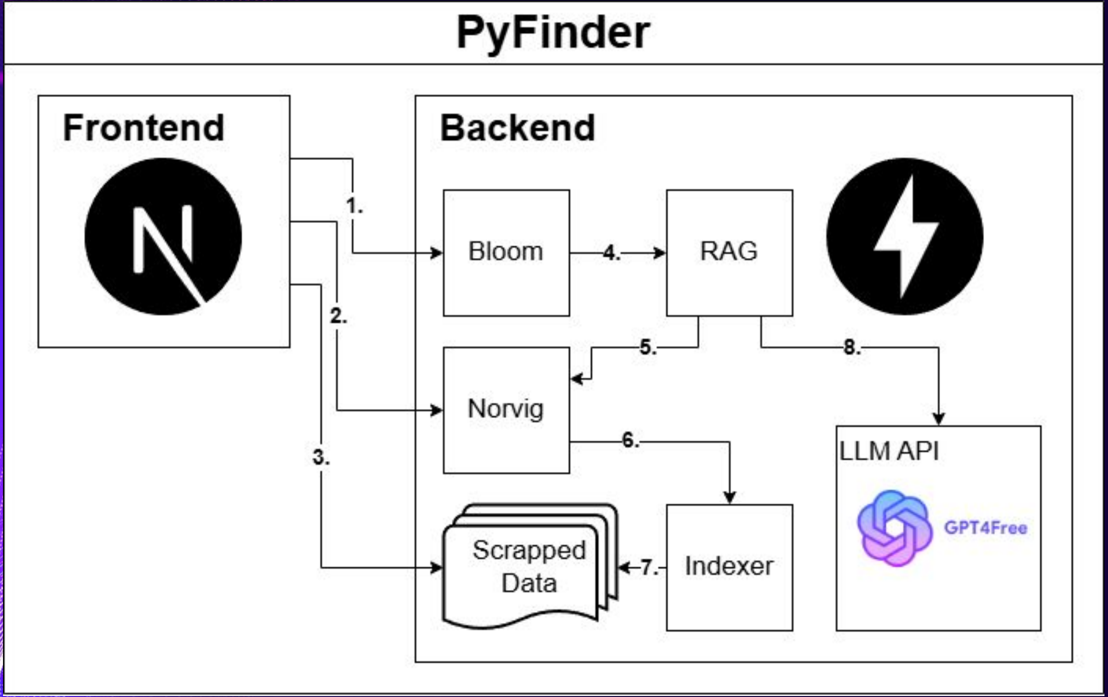
arnir0/Tiny-LLM, sshleifer/tiny-gpt2

Embeddings model

sentence-transformers/all-MiniLM-L6-v2

03

System Design



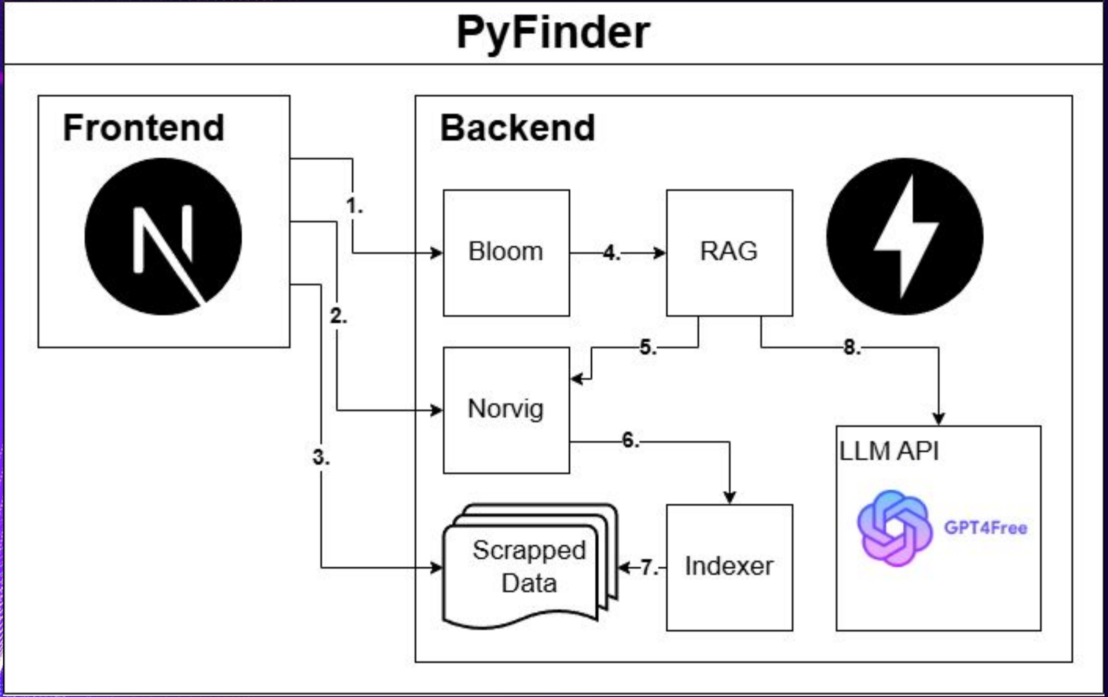
Workflows

1. Frontend ➡️ Bloom – Filters bad content

2. Frontend ➡️ Norvig – Spell corrector

3. Frontend ➡️ Scraped Data – Displays scraped docs

4. Bloom ➡️ RAG – Sends clean query to RAG



Workflows

5. RAG ➡️ Norvig – Filters bad content

6. Norvig ➡️ Indexer – Fetches relevant docs

7. Indexer ➡️ Scraped Data – Retrieves matched files

8. RAG ➡️ LLM API – Generates and returns LLM answer

04

Components

Bloom Filter

Bad Words list
Merged from Google Profanity List, LDNOOBW (English & Russian)

Efficient Storage
- Uses **ScalableBloomFilter** (5000-word capacity, 0.1% false positive)
- Stores words and phrases (up to 5 words)

Moderation Logic
- Scans individual and multi-word phrases
- Returns first match with offending term

Norvig Spell Corrector

1. Text Preprocessing

- Removes stopwords, tokenizes lowercase words and strips non-ASCII characters

2. Language Model

- Builds frequency model from cleaned docs
- Calculates word probabilities

3. Suggestions

- Edit types: deletion, transposition, replacement, insertion
- Filters to valid words, returns most likely candidate

4. Query Processing

- Transforms the original query, retains punctuation

Indexer: Inverted Index

1. Indexing

- Maps words to documents
- Stores: Word counts (TF), Doc lengths (normalization), Titles

2. Search

- Ranks using TF-IDF weighted by inverse Levenshtein distance

3. Fuzzy Matching

- Matches within edit distance
- Partial matches contribute based on similarity

Indexer: LLM Embeddings + Ball Tree

1. **Embedding Generation**
- Converts documents to dense vectors using mean pooling

2. **Indexing**
- Stores embeddings in a Ball Tree

3. **Search**
- Embeds query
- Finds top-k nearest documents
- Returns documents with corresponding distances (scores)

Strengths
- Handles semantic similarity
- Recognizes paraphrasing and related terms

RAG (Retrieval-Augmented Generation)

1. Prompt Engineering

- Restricts answers to context (fetched Python documents from indexer)
- Provides references to documents
- Forbids unsource info and code

2. Retrieval Process

1. Find top-k nearest documents using indexer
2. Build context using the found documents
3. Combine context with user query
4. Pass to LLM with source tracking

3. LLM Handling

- Async/sync client support
- Streaming + rate limiting
- Error and timeout handling

Architectures Comparison

	Technologies	Advantages
Inverted Index	Inverted Index + Levenshtein distance	Simple, exact match, lightweight
Embedding Search	LLM Embeddings + Ball Tree	Resilient to synonyms and phrasing
RAG	API + Prompt Engineering	Multi-source synthesis, deep answers, citation-based, filters irrelevant content

Challenges & Solutions

Word2Vec indexer was not accurate
Switched to LLM embeddings

Local LLM too slow or heavy
Switched to free hosted APIs

Poor spelling correction
Added Norvig-based spell corrector

05

Evaluation

Metrics

LLM

Assess the quality and relevance of LLM answers

Use cosine similarity of embeddings between answers, contexts, queries, and ground truths

Ranking

Evaluate the quality of document retrieval and ranking at cutoff K

LLM-specific metrics

Answer Relevancy – semantic similarity between the generated answer and the input query, reflecting how relevant the answer is to the question

Context Precision – maximum semantic similarity between the generated answer and each retrieved context document, indicating how well the answer aligns with the context

Context Recall – semantic similarity between ground truth answers and aggregated context documents, showing how well the context covers the correct answer

LLM-specific metrics (2)

Faithfulness – measures how faithfully the answer is grounded in the retrieved context, computed as similarity between the answer and combined context embeddings

BLEU – a precision-based metric evaluating n-gram overlap between generated answer and ground truth references, widely used in machine translation and text generation

ROUGE-1 – measures unigram overlap between generated answer and ground truth, indicating content similarity

Ranking metrics

F1@K – harmonic mean of precision and recall of relevant documents within top K

MAP@K – average precision of relevant documents ranked highly within top K

MAR@K – average recall at cutoff K, measuring proportion of relevant documents retrieved

MRR@K – average reciprocal rank of the first relevant document within top K, rewarding early relevant retrieval

nDCG@K – (Normalized Discounted Cumulative Gain), evaluates ranking quality by considering relevance and position of documents, favoring relevant documents appearing earlier

Evaluation workflow

We have decided to compare two models with API: qwen-2-72b and evil, in combinations with both proposed indexers: inverted_idx and llm_tree_idx

Without details, the evaluation workflow was as following:

1. Come up with the "evaluation queries" together with the ground truths
2. Generate and parse responses of models (both RAG and indexers) and save the results
3. Compute metrics based on the responses and save them
4. Plot the obtained results

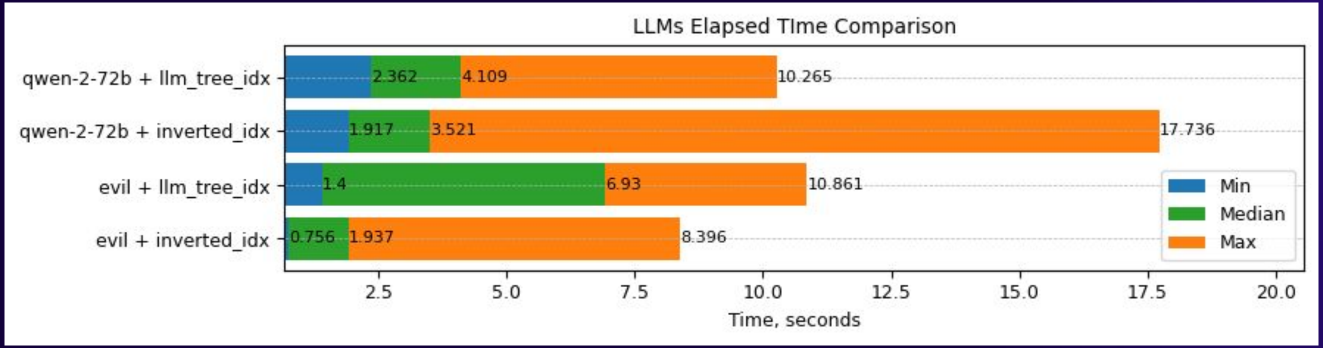
Evaluation queries example

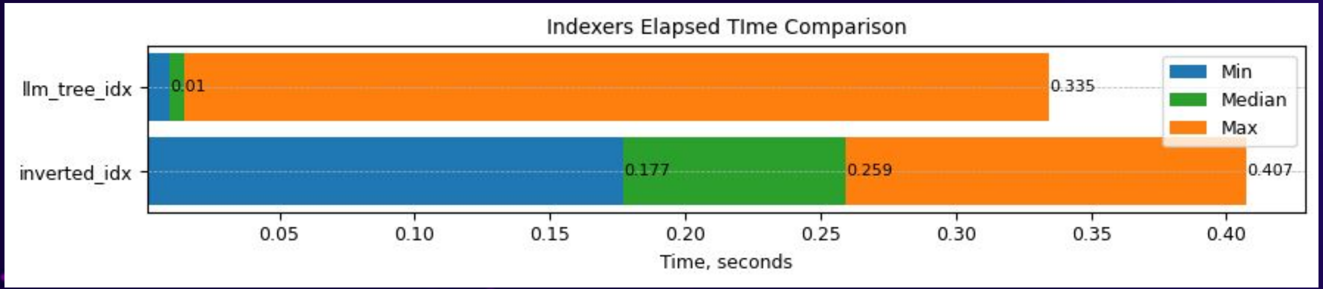
```
"datetime difference": {
    "query": "How do I calculate the difference between two dates?",
    "ground_truths": [
        "datetime.timedelta",
        "datetime.datetime"
    ]
},
"list permutations": {
    "query": "How do I generate permutations of a list?",
    "ground_truths": [
        "itertools.permutations"
    ]
},
"generate random number": {
    "query": "How can I generate a random number?",
    "ground_truths": [
        "random.random",
        "random.randint",
        "random.uniform",
        "random.randrange"
    ]
},
```

06

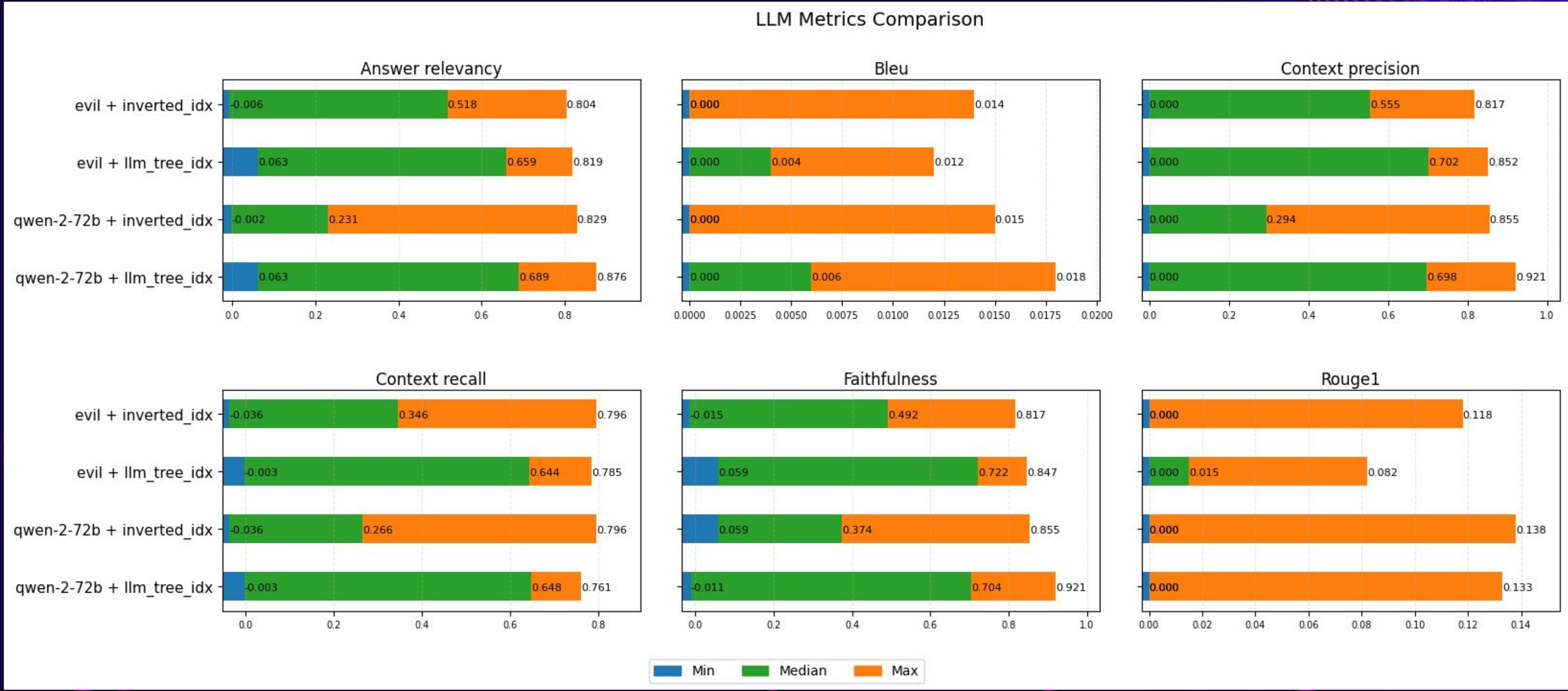
Results

Elapsed Time

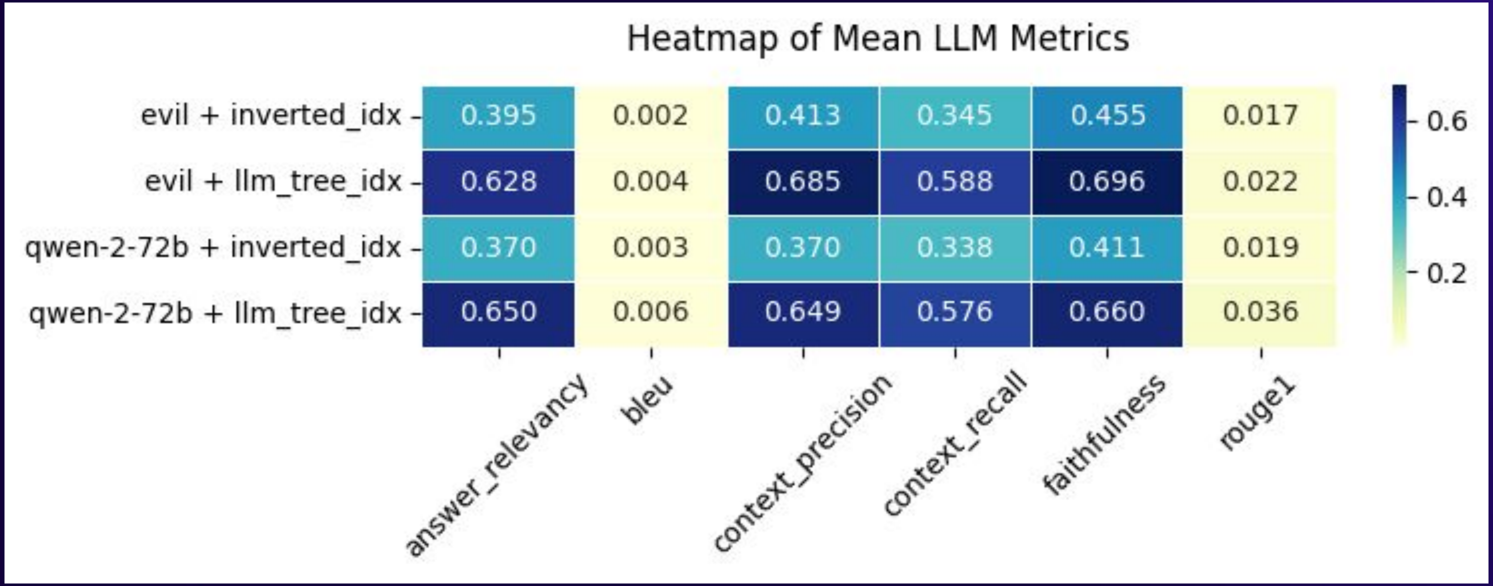




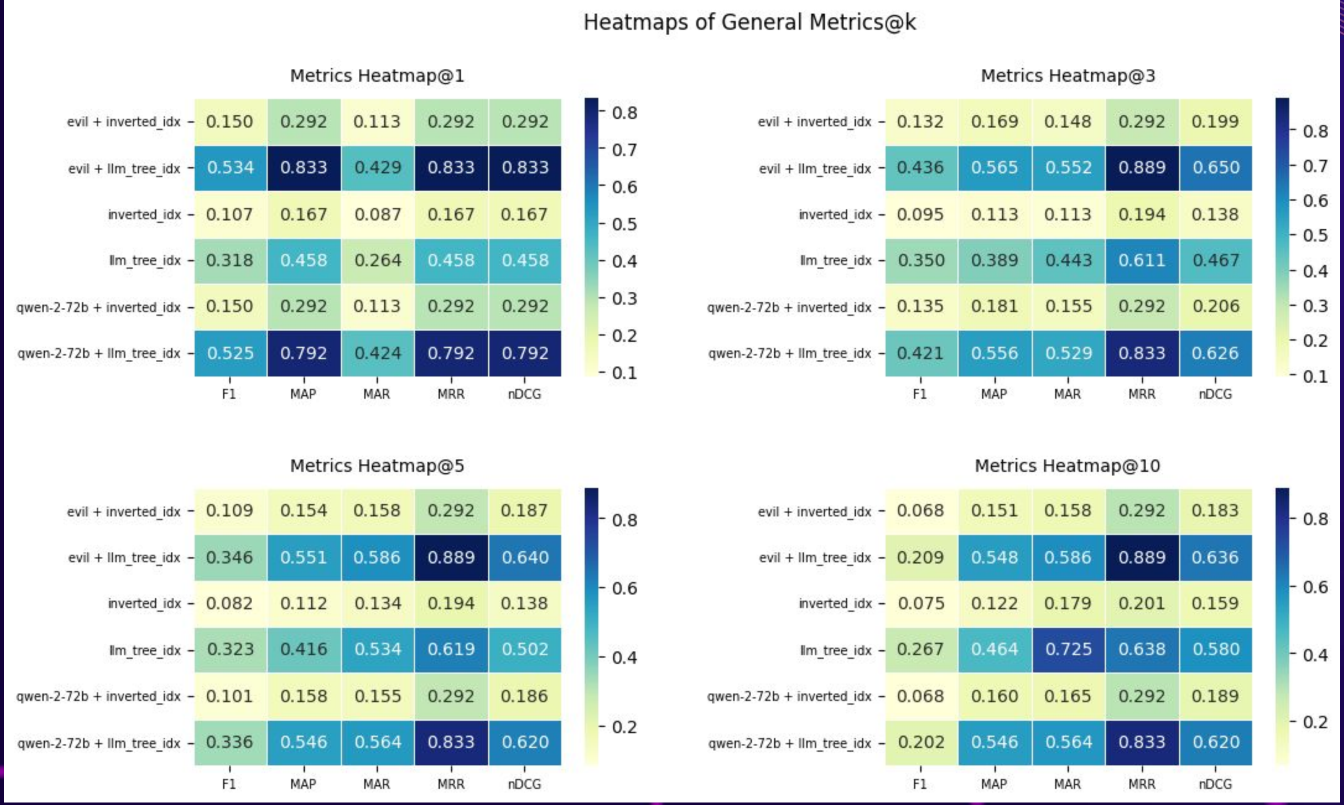
LLM-specific results



LLM-specific results (2)



Ranking results



07

Demo time!

Merci!

Any questions?