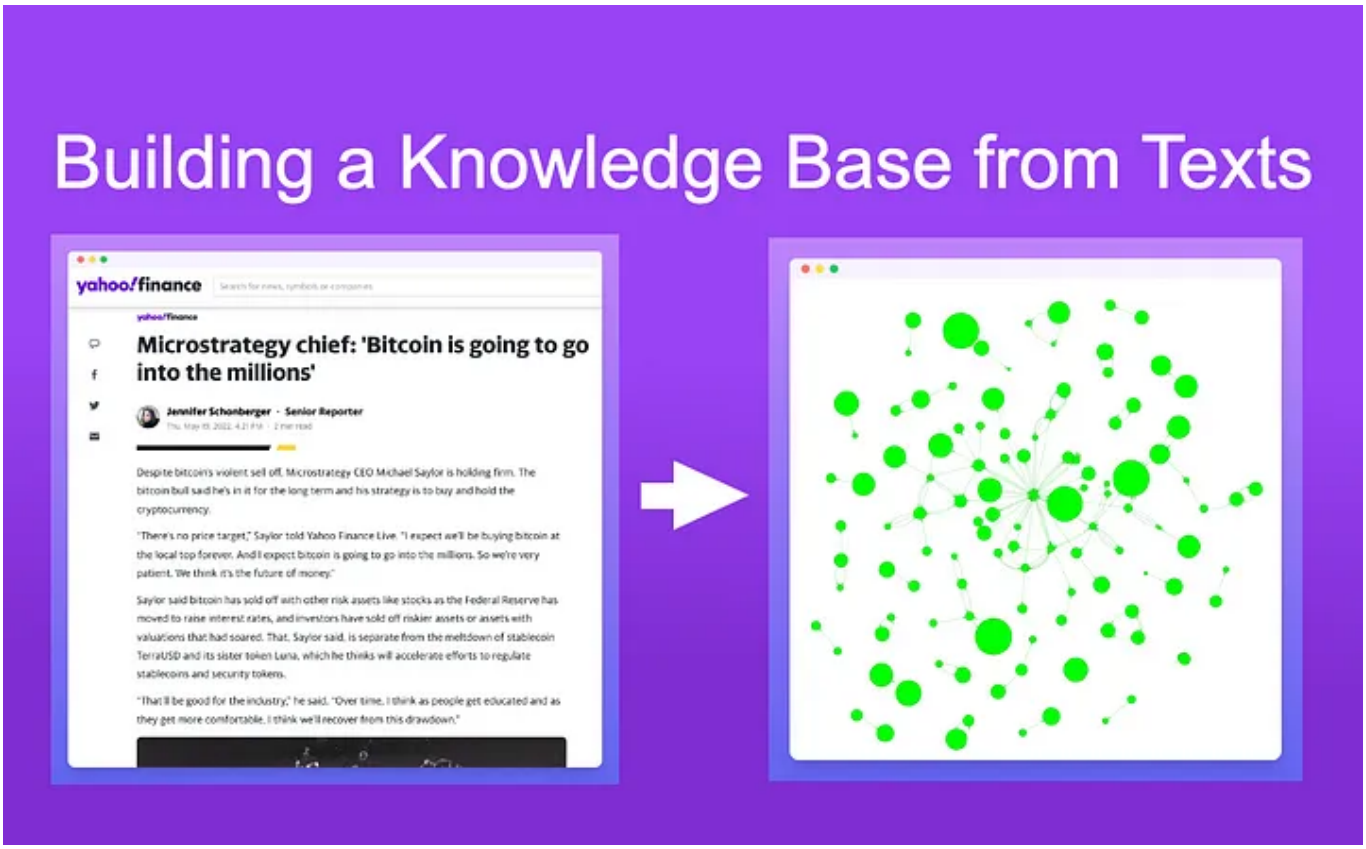
Building a Knowledge Base from Texts: a Full Practical Example

Implementing a pipeline for extracting a Knowledge Base from texts or online articles

Fabio Chiusano · Following

Published in NLPlanet · 12 min read · May 24, 2022

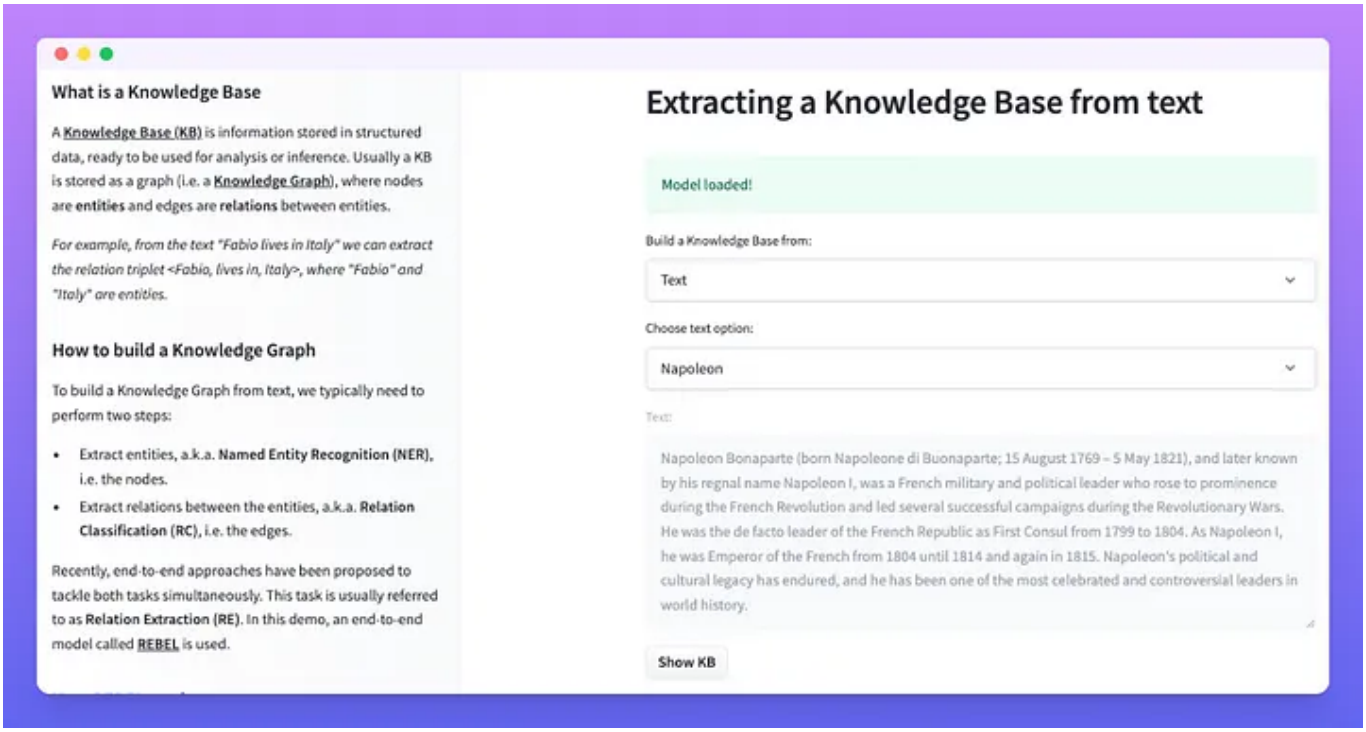


> *Hello fellow NLP enthusiasts! In this article, we see how to implement a pipeline for extracting a Knowledge Base from texts or online articles. We'll talk about Named Entity Recognition, Relation Extraction, Entity Linking, and other common steps done when building Knowledge Graphs. You can try the final demo on this Hugging Face Space and check the code on Colab. Enjoy!* 😄

Ready to start? This is what we are going to do:

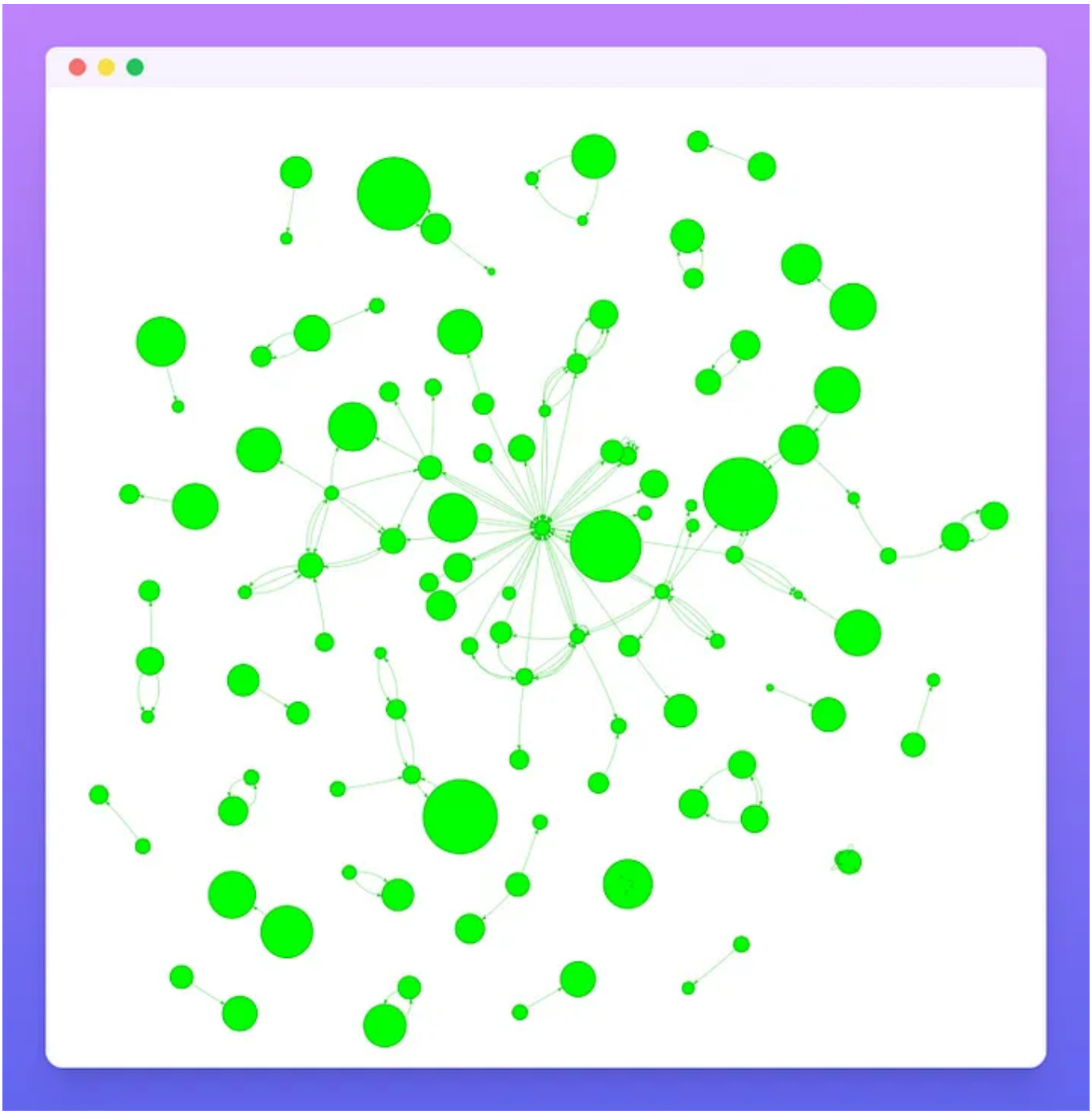
1. Learn what are knowledge bases and knowledge graphs.
2. Learn how to build knowledge graphs and how the REBEL model works.
3. Implement a full pipeline that extracts relations from text, builds a knowledge graph, and visualizes it.
4. Build an interactive demo with Streamlit and deploy it to Hugging Face Spaces.

To get an early idea of what the final output will be, try the demo on this Hugging Face Space.



The knowledge base extraction pipeline demo.

Here is an example of a knowledge graph extracted from 20 news articles about *"Google"*. At the end of this guide, you'll be able to build knowledge graphs from any list of articles you like.



A knowledge graph extracted from 20 news articles about “Google”.

Detail of a knowledge graph extracted from 20 news articles about “Google”.

Let’s start!

Knowledge Bases and Knowledge Graphs

A Knowledge Base (KB) is information stored as structured data, ready to be used for analysis or inference. Usually, a KB is stored as a graph (i.e. a Knowledge Graph), where nodes are entities and edges are relations between entities.

For example, from the text *"Fabio lives in Italy"* we can extract the relation triplet `<Fabio, lives in, Italy>`, where *"Fabio"* and *"Italy"* are entities.

Extracting relation triplets from raw text is a crucial task in Information Extraction, enabling multiple applications such as populating or validating knowledge bases, fact-checking, and other downstream tasks.

How to build a Knowledge Graph

To build a knowledge graph from text, we typically need to perform two steps:

1. Extract **entities**, a.k.a. Named Entity Recognition (NER), which are going to be the nodes of the knowledge graph.
2. Extract **relations** between the entities, a.k.a. Relation Classification (RC), which are going to be the edges of the knowledge graph.

These multiple-step pipelines often propagate errors or are limited to a small number of relation types. Recently, end-to-end approaches have been proposed to tackle both tasks simultaneously. This task is usually referred to as Relation Extraction (RE). In this article, we'll use an end-to-end model called REBEL, from the paper Relation Extraction By End-to-end Language generation.

How REBEL works

REBEL is a text2text model trained by BabelScape by fine-tuning BART for translating a raw input sentence containing entities and implicit relations into a set of triplets that explicitly refer to those relations. It has been trained on more than 200 different relation types.

The authors created a custom dataset for REBEL pre-training, using entities and relations found in Wikipedia abstracts and Wikidata, and filtering them using a RoBERTa Natural Language Inference model (similar to this model). Have a look at the paper to know more about the creation process of the dataset. The authors also published their dataset on the Hugging Face Hub.

The model performs quite well on an array of Relation Extraction and Relation Classification benchmarks.

You can find REBEL in the Hugging Face Hub.

Implementing the Knowledge Graph extraction pipeline

Here is what we are going to do, progressively tackling more complex scenarios:

1. Load the Relation Extraction REBEL model.
2. Extract a knowledge base from a short text.
3. Extract a knowledge base from a long text.
4. Filter and normalize entities.
5. Extract a knowledge base from an article at a specific URL.

6. Extract a knowledge base from multiple URLs.
7. Visualize knowledge bases.

You can find the complete code in this Colab.

First, we install the required libraries.

We need each library for the following reasons:

- transformers: Load the REBEL mode.
- wikipedia: Validate extracted entities by checking if they have a corresponding Wikipedia page.
- newspaper: Parse articles from URLs.
- GoogleNews: Read Google News latest articles about a topic.
- pyvis: Graphs visualizations.

Let's import all the necessary libraries and classes.

Load the Relation Extraction model

Thanks to the `transformers` library, we can load the pre-trained REBEL model and tokenizer with a few lines of code.

From short text to Knowledge Base

The next step is to write a function that is able to parse the strings generated by REBEL and transform them into relation triplets (e.g. the `<Fabio, lives in, Italy` triplet). This function must take into account additional new tokens (i.e. the `<triplet>` , `<subj>` , and `<obj>` tokens) used while training the model. Fortunately, the REBEL model card provides us with a complete code example for this function, which we'll use as-is.

The function outputs a list of relations, where each relation is represented as a dictionary with the following keys:

- `head` : The subject of the relation (e.g. *"Fabio"*).
- `type` : The relation type (e.g. *"lives in"*).
- `tail` : The object of the relation (e.g. *"Italy"*).

Next, let's write the code for implementing a knowledge base class. Our `KB` class is made of a list of relations and has several methods to deal with adding new relations to the knowledge base or printing them. It implements a very simple logic at the moment.

Last, we define a `from_small_text_to_kb` function that returns a `KB` object with relations extracted from a short text. It does the following:

1. Initialize an empty knowledge base `kb` object.
2. Tokenize the input text.
3. Use REBEL to generate relations from the text.
4. Parse REBEL output and store relation triplets into the knowledge base object.
5. Return the knowledge base object.

Let's try the function with some text about Napoleon Bonaparte from Wikipedia.

The model is able to extract several relations, such as Napoleon's date of birth and date of death, and his participation in the French Revolution. Nice!

From long text to Knowledge Base

Transformer models like REBEL have memory requirements that grow quadratically with the size of the inputs. This means that REBEL is able to work on common hardware at a reasonable speed with inputs of about 512

tokens, which correspond to about 380 English words. However, we may need to extract relations from documents long several thousands of words.

Moreover, from my experiments with the model, it seems to work better with shorter inputs. Intuitively, raw text relations are often expressed in single or contiguous sentences, therefore it may not be necessary to consider a high number of sentences at the same time to extract specific relations. Additionally, extracting a few relations is a simpler task than extracting many relations.

So, how do we put all this together?

For example, we can divide an input text long 1000 tokens into eight shorter overlapping spans long 128 tokens and extract relations from each span. While doing so, we also add some metadata to the extracted relations containing their span boundaries. With this info, we are able to see from which span of the text we extracted a specific relation which is now saved in our knowledge base.

Let's modify the `KB` methods so that span boundaries are saved as well. The relation dictionary has now the keys:

- `head` : The subject of the relation (e.g. *"Fabio"*).
- `type` : The relation type (e.g. *"lives in"*).
- `tail` : The object of the relation (e.g. *"Italy"*).
- `meta` : A dictionary containing meta information about the relation. This dictionary has a `spans` key, whose value is the list of span boundaries (e.g. *[[0, 128], [119, 247]]*) where the relation has been found.

Next, we write the `from_text_to_kb` function, which is similar to the `from_small_text_to_kb` function but is able to manage longer texts by splitting them into spans. All the new code is about the spanning logic and the management of the spans into the relations.

Let’s try it with a longer text of 726 tokens about Napoleon. We are currently splitting the text into spans long 128 tokens.

The text has been split into six spans, from which 23 relations have been extracted! Note that we also know from which text span each relation comes.

Filter and normalize entities

If you look closely at the extracted relations, you can see that there's a relation with the entity *"Napoleon Bonaparte"* and a relation with the entity *"Napoleon"*. How can we tell our knowledge base that the two entities should be treated as the same?

One way to do this is to use the `wikipedia` library to check if *"Napoleon Bonaparte"* and *"Napoleon"* have the same Wikipedia page. If so, they are normalized to the title of the Wikipedia page. If an extracted entity doesn't have a corresponding Wikipedia page, we ignore it at the moment. This step is commonly called Entity Linking.

Note that this approach relies on Wikipedia to be constantly updated by people with relevant entities. Therefore, it won't work if you want to extract entities different from the ones already present in Wikipedia. Moreover, note that we are ignoring *"date"* (e.g. the `15 August 1769` in `<Napoleon, date of birth, 15 August 1769>`) entities for simplicity.

Let's modify our `KB` code:

- The `KB` now stores an `entities` dictionary with the entities of the stored relations. The keys are the entity identifiers (i.e. the title of the corresponding Wikipedia page), and the value is a dictionary containing the Wikipedia page `url` and its `summary`.
- When adding a new relation, we now check its entities with the `wikipedia` library.

Let’s extract relations and entities from the same text about Napoleon:

All the extracted entities are linked to Wikipedia pages and normalized with their titles. *"Napoleon Bonaparte"* and *"Napoleon"* are now both referred to with *"Napoleon"*!

From article at URL to Knowledge Base

Let's go another step further. We want our knowledge base to manage the addition of relations and entities from articles from around the web, and to keep track of where each relation comes from.

To do this, we need to modify our `KB` class so that:

- Along with `relations` and `entities`, `sources` (i.e. articles from around the web) are stored as well. Each article has its URL as key and a dictionary with keys `article_title` and `article_publish_date` as value. We'll see later how to extract these two features.
- When we add a new relation to our knowledge base, the relation `meta` field is now a dictionary with article URLs as keys, and another dictionary containing the `spans` as value. In this way, the knowledge base keeps track of all the articles from which a specific relation has been extracted. This information can be an indicator of the quality of an extracted relation.

Next, we modify the `from_text_to_kb` function so that it prepares the relation `meta` field taking into account article URLs as well.

Last, we use the `newspaper` library to download and parse articles from URLs and define a `from_url_to_kb` function. The library automatically extracts the article text, title, and publish date (if present).

Let's try to extract a knowledge base from the article Microstrategy chief: 'Bitcoin is going to go into the millions'.

The KB is showing a lot of information!

- From the `entities` list, we see that Microstrategy is an American company.
- From the `relations` list, we see that Michael J. Saylor is a founder of the Microstrategy company, and where we extracted such relation (i.e. the article URL and the text span).

- From the `sources` list, we see the title and publish date of the aforementioned article.

From multiple articles to Knowledge Base

We are almost done! Consider this last scenario: creating a knowledge base from multiple articles. We can deal with it by extracting a separate knowledge base from each article and then merging all the knowledge bases together. Let's add a `merge_with_kb` method to our `KB` class.

Then, we use the `GoogleNews` library to get the URLs of recent news articles about a specific topic. Once we have multiple URLs, we feed them to the `from_urls_to_kb` function, which extracts a knowledge base from each article and then merges them together.

Let’s try extracting a knowledge base from three articles from Google News about *“Google”*.

The knowledge bases are getting bigger! We got 10 entities, 10 relations, and 3 sources. Note that we know from which article each relation comes.

Visualize the Knowledge Base

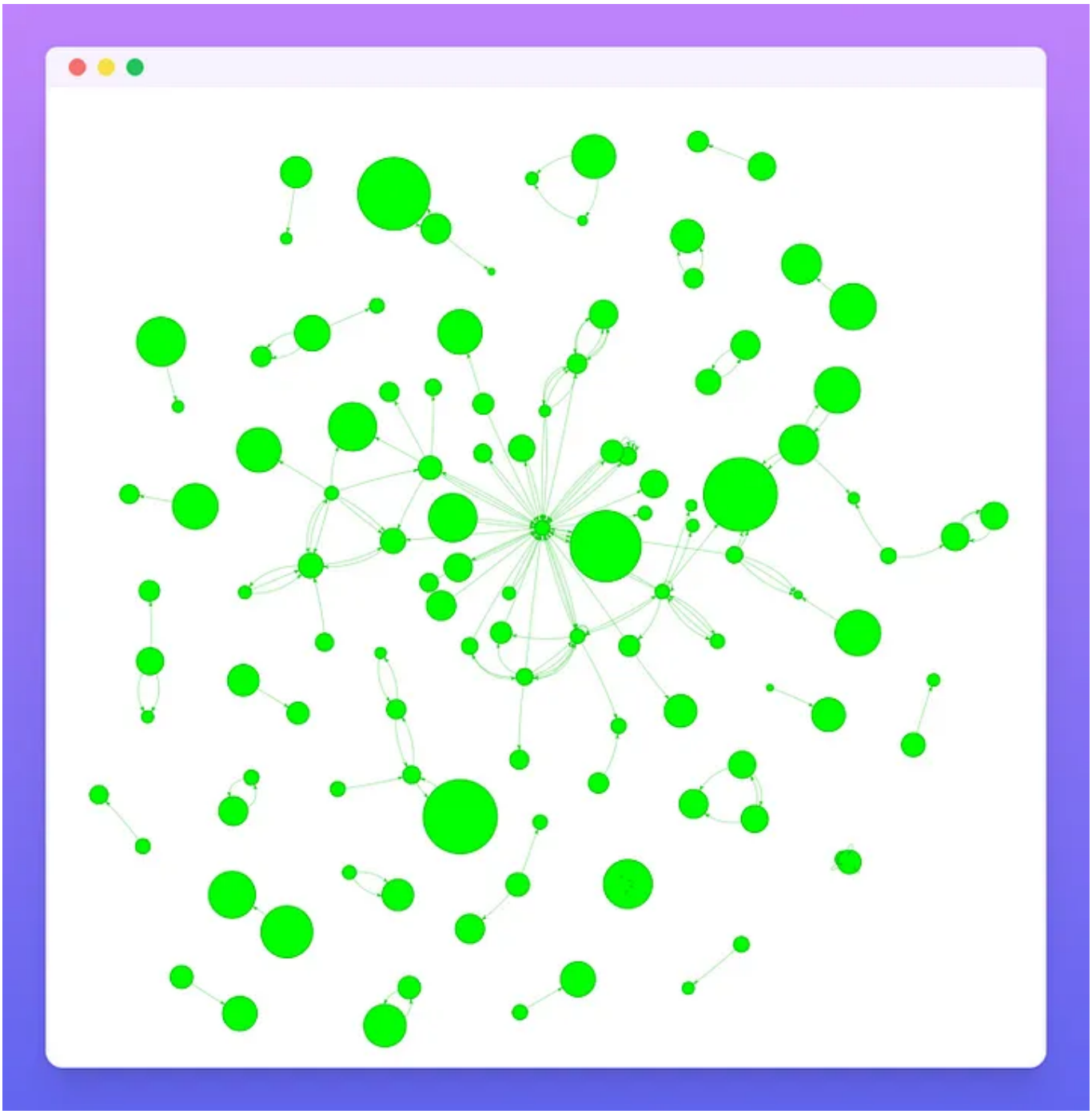
Congratulations if you've read this far, we're done with the scenarios! Let's visualize the output of our work by plotting the knowledge bases. As our knowledge bases are graphs, we can use the `pyvis` library, which allows the creation of interactive network visualizations.

We define a `save_network_html` function that:

1. Initialize an empty directed `pyvis` network.
2. Add the knowledge base entities as nodes.
3. Add the knowledge base relations as edges.
4. Save the network in an HTML file.

Let's try the `save_network_html` function with a knowledge base built from 20 news articles about *"Google"*.

This is the resulting graph:



A knowledge graph extracted from 20 news articles about “Google”.

Zooming into the graph, we see many relations extracted with the *Google* entity.

Detail of a knowledge graph extracted from 20 news articles about “Google”.

Remember that, even though they are not visualized, the knowledge graph saves information about the provenience of each relation (e.g. from which articles it has been extracted and other metadata), along with Wikipedia data about each entity. Visualizing knowledge graphs is useful for debugging purposes, but their main benefits come when used for inference.

You can find the complete code in this Colab.

Build an interactive demo with Streamlit and deploy it to Hugging Face Spaces

The code works fine in our Jupyter notebook, but what if we want to let other non-technical people try our knowledge base extraction pipeline? We can build a small demo using Streamlit.

Streamlit is a Python library that allows data scientists to easily create small interactive demos so that other people can test their machine learning models or see their data analyses. Check out the Streamlit Gallery to learn what can be done with the library.

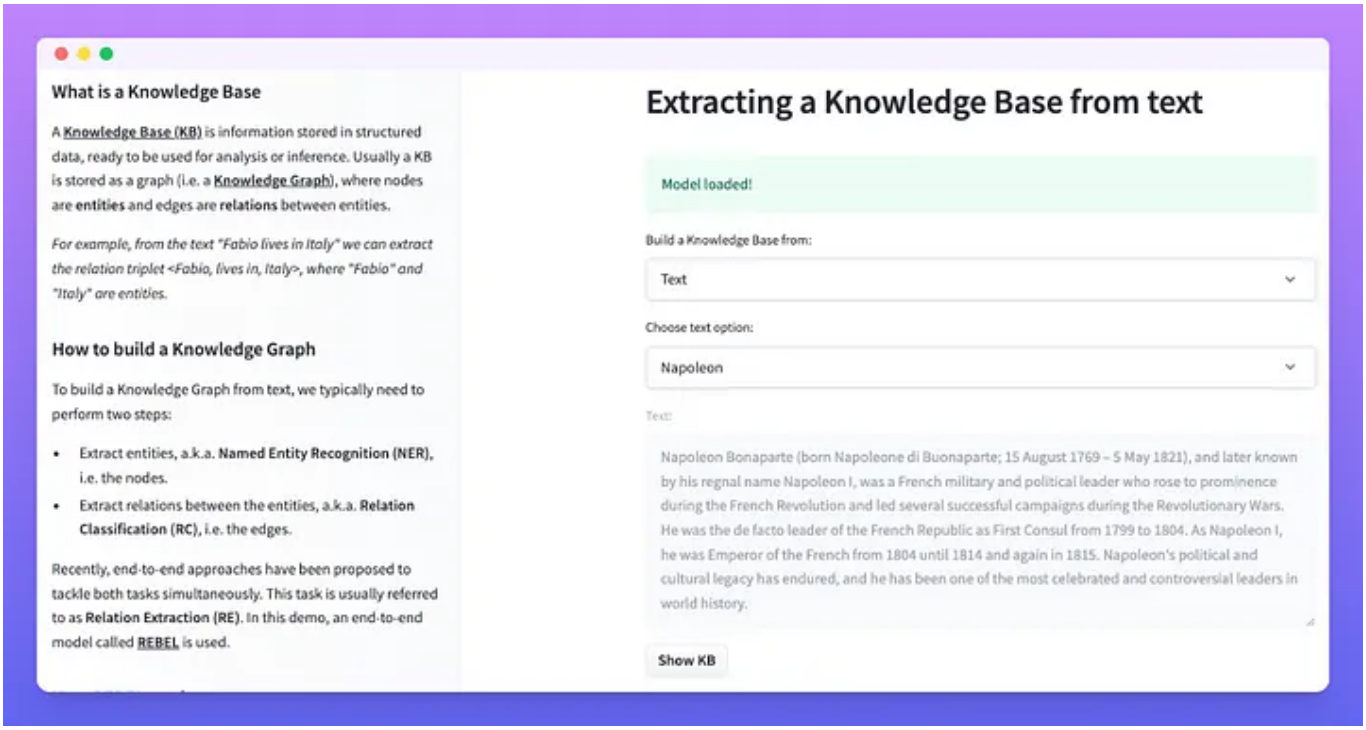
I'm not going into code details with Streamlit, but rest assured that it's very easy to use and quick to learn. I suggest reading the Streamlit documentation, it takes a couple of hours.

Have a look at this repo to see the complete code of our Streamlit application. There are two main components:

- The `app.py` file: Contains the Streamlit code of the app. It's where we create some interactive components and write the logic to connect them with the knowledge base extraction pipeline.
- The `requirements.py` file: Contains all the Python libraries used in the Streamlit application. It's not necessary to add the Streamlit library to this file.

Once we have tested locally our Streamlit application and everything works as expected, we need to upload it to a new Hugging Face Space. This step is just a matter of copying the files of your Streamlit applications to a new repository created on your Hugging Face account. Refer to this article to learn how to host your Streamlit projects in Hugging Face Spaces.

This is the resulting Hugging Face Space. There are some knowledge base examples already loaded into memory, but it's also possible to extract knowledge bases from custom texts and URLs. Try it out!



The knowledge base extraction pipeline demo.

Building a small demo with Streamlit is very easy and, in conjunction with Hugging Face Spaces, allows other people to test your work without needing the knowledge of how to run a Jupyter notebook.

Considerations and next steps

If you look closely at the extracted knowledge graphs, some relations are false. Indeed, relation extraction models are still far from perfect and require further steps in the pipeline to build reliable knowledge graphs from text. Consider this demo as a starting step!

Possible next steps are:

- Manage *"date"* entities in the `KB` class.
- Use sentence boundaries as span boundaries.
- Test other NER and Relation Classification models, such as OpenNRE.
- Add a function that computes a quality metric for each relation, taking into account the relation sources.
- Integrate the `KB` class with other open-source knowledge graphs, such as KBpedia.

Thank you for reading! If you are interested in learning more about NLP, remember to follow NLPlanet on Medium, LinkedIn, Twitter, and join our new Discord server!

NLP | Artificial Intelligence | Data Science | Machine Learning | Knowledge Graph

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

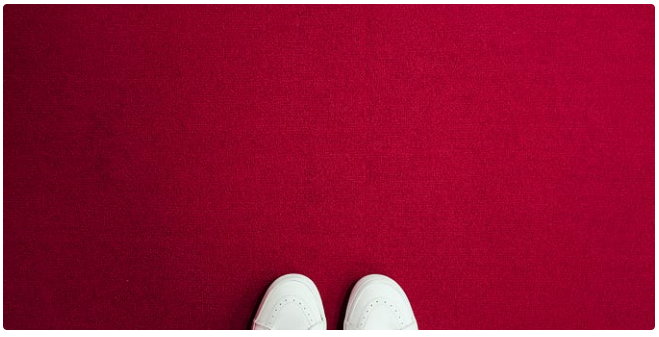
Written by Fabio Chiusano

Following

3.7K Followers · Editor for NLPlanet

Freelance data scientist — Top Medium writer in Artificial Intelligence

More from Fabio Chiusano and NLPlanet



Fabio Chiusano in NLPlanet

Two minutes NLP—Learn the ROUGE metric by examples

ROUGE-N, ROUGE-L, ROUGE-S, pros and cons, and ROUGE vs BLEU

5 min read · Jan 19, 2022

126 2

Fabio Chiusano in NLPlanet

Weekly AI and NLP News—September 11th 2023

The 100 most influential people in AI, Falcon 180B, and an open-source Code Interpreter

4 min read · Sep 11

41 3

Fabio Chiusano in NLPlanet

Weekly AI and NLP News—September 25th 2023

DALL·E 3, AlphaMissense, and LoRA for finetuning LLMs for longer context windows

5 min read · Sep 25

63 1

Fabio Chiusano in NLPlanet

Weekly AI and NLP News—September 5th 2023

ChatGPT Enterprise, RLAIF, and expanding context window up to 126k tokens

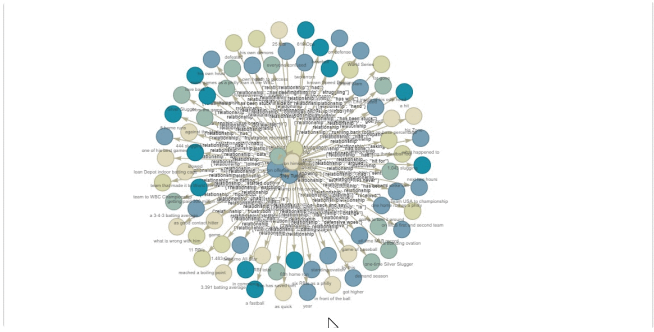
4 min read · Sep 5

42

See all from Fabio Chiusano

See all from NLPlanet

Recommended from Medium



Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

✨ · 17 min read · 4 days ago

501 4

0xdevshah in AI Skunks

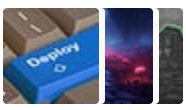
Knowledge Graphs: A Comprehensive Analysis

Knowledge graphs are becoming increasingly important in NLP due to their ability to mode...

8 min read · Apr 9

63

Lists



Predictive Modeling w/ Python

20 stories · 452 saves

Natural Language Processing

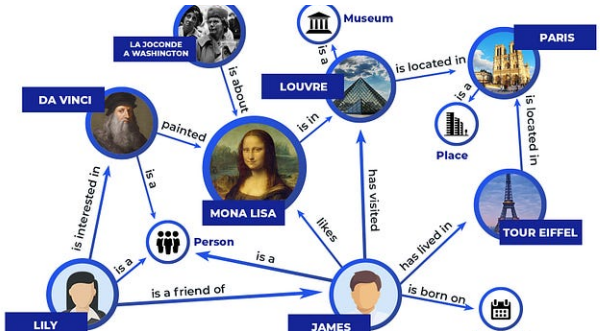
669 stories · 283 saves

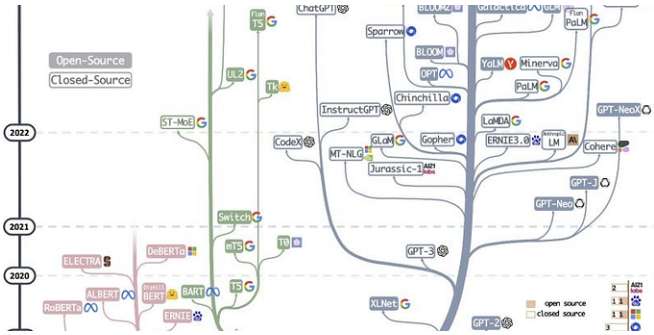
Practical Guides to Machine Learning

10 stories · 519 saves

ChatGPT prompts

24 stories · 459 saves





Alla Chepurova in DeepPavlov

Improving Knowledge Graph Completion with Generative LM...

Combining both internal LM knowledge and external data from KG

13 min read · Sep 5

36

Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

David Shapiro

A Pro's Guide to Finetuning LLMs

Large language models (LLMs) like GPT-3 and Llama have shown immense promise for...

12 min read · Sep 23

283 6

Ryan Nguyen in Towards AI

So, You Want To Improve Your RAG Pipeline

Ways to go from prototype to production with LlamaIndex

✦ · 9 min read · Sep 27

176 2

See more recommendations