Retrieval Augmented Generation

# Factual Claim Verification using RAG (Retrieval-Augmented Generation)

## Context

This project is developed as part of the advanced training in the Master in Data Science at the UCC, specifically within the Natural Language Processing (NLP) course. The work integrates concepts of Transformer architectures, prompt engineering, and information retrieval systems.

### Introduction

In the era of information overload, the ability to distinguish between factual truth and misinformation is one of the greatest challenges for AI. This notebook aims to develop and evaluate a Fact-Checking system using the FEVER (Fact Extraction and VERification) dataset and the Llama-3.2-3B-Instruct large language model.

The project documents the evolution of a model from "pure memory" toward "evidence-based reasoning" through two distinct phases:

1. **Baseline Evaluation (Zero-shot)**: We analyze the model's intrinsic ability to verify 500 claims based solely on its pre-trained knowledge. This stage identifies classic LLM limitations, such as a skeptical bias towards refutation when specific data is missing.
2. **RAG Implementation (Retrieval-Augmented Generation)**: We build an architecture that connects the model to an external Wikipedia knowledge base. By utilizing LangChain for intelligent text segmentation and FAISS for semantic vector search, we enable the model to act as a documented verifier.

### Technical Objectives

- **Data Infrastructure Design**: Implement chunking and overlap strategies to optimize the retrieval of factual evidence.
- **Hallucination Mitigation**: Evaluate how access to external sources reduces false positives and strengthens model reliability.
- **Metric Analysis**: Perform a comparative study of Precision, Recall, and F1-Score to validate the net impact of the RAG architecture against the traditional zero-shot approach.

---

## Libraries

```
!pip install -q langchain-huggingface langchain-community faiss-cpu wikipedia
```

```
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.5/2.5 MB 47.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 23.8/23.8 MB 118.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 73.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.7/64.7 kB 6.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.0/51.0 kB 5.0 MB/s eta 0:00:00
  Building wheel for wikipedia (setup.py) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. Th
google-colab 1.0.0 requires requests==2.32.4, but you have requests 2.32.5 which is incompatible.
```

```
import os
import pandas as pd
import numpy as np
import torch
from tqdm import tqdm

# Transformers and LLM
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

# Metrics
from sklearn.metrics import classification_report, precision_recall_fscore_support

# LangChain and RAG components
from langchain_huggingface import HuggingFaceEmbeddings
```

```
from langchain_community.vectorstores import FAISS
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WikipediaLoader

# Warning management
import warnings
warnings.filterwarnings('ignore')

# Check GPU availability
device = 0 if torch.cuda.is_available() else -1
print(f"Device for inference: {'GPU' if device == 0 else 'CPU'}")
```

```
Device for inference: GPU
```

# Introduction

In the current context of digital misinformation, the ability to verify facts automatically (Automated Fact-Checking) is one of the most critical areas of Natural Language Processing (NLP). For this task, we will work with FEVER, one of the reference datasets for research in claim verification.

FEVER (Fact Extraction and VERification) is a large-scale dataset consisting of more than 185,000 claims generated from Wikipedia. Unlike simple text classification tasks, FEVER requires complex reasoning that combines two NLP sub-tasks:

- Information Retrieval: Finding relevant documents that contain the evidence.
- Natural Language Inference: Determining whether the retrieved evidence supports or refutes the claim.

Each entry in the dataset presents a claim and must be classified into one of the following three labels based exclusively on the provided evidence (Wikipedia articles):

- Supported: The evidence contains sufficient information to confirm that the claim is true.
- Refuted: The evidence contains information that contradicts the claim (it is false).
- Not Enough Info: There is no evidence in the given knowledge base to confirm or deny the claim.

In this task, we will focus only on two classes: Supported and Refuted.

# 1 - Baseline: Zero-shot LLM as Classifier

Load the dataset `df_sample_fever.csv`. This dataset corresponds to a random sample of 500 claims. You should only use the **claim** and **label** fields.

Use the `meta-llama/Llama-3.2-3B-Instruct` model to obtain predictions for each claim (get SUPPORTS or REFUTES label). Justify the prompt design.

Calculate the Precision, Recall, and F1-score metrics for the analyzed set (overall and per class). Comment on the results.

**Observations**

- See model at: https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct
- To use this model, you must register on Hugging Face and request permission. When importing it, you must enter your Hugging Face token. Review Tutorial 3.
- For this question, you must work with GPU.
- Processing time for the 500 claims is approximately 5 minutes.

```
import pandas as pd
pd.read_csv("/content/df_sample_FEVER.csv")
```

| | id | verifiable | label | claim |
|---|---|---|---|---|
| 0 | 198802 | VERIFIABLE | SUPPORTS | An American producer was the writer of the pil... |
| 1 | 145296 | VERIFIABLE | SUPPORTS | Kate Nash was dropped from her label. |
| 2 | 93897 | VERIFIABLE | SUPPORTS | Rope was released. |

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
from huggingface_hub import login

login(token="")

model_id = "meta-llama/Llama-3.2-3B-Instruct"

tokenizer = AutoTokenizer.from_pretrained(model_id)
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.bfloat16,
    device_map="auto"
)

# Text generation pipeline optimized for classification
llm_classifier = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=10,
    temperature=0.01, # Low temperature for deterministic output
    do_sample=False
)
```

```
tokenizer_config.json: 100%                 54.5k/54.5k [00:00<00:00, 1.04MB/s]
tokenizer.json: 100%                        9.09M/9.09M [00:00<00:00, 31.3MB/s]
special_tokens_map.json: 100%               296/296 [00:00<00:00, 39.4kB/s]
config.json: 100%                           878/878 [00:00<00:00, 97.6kB/s]
`torch_dtype` is deprecated! Use `dtype` instead!
model.safetensors.index.json: 100%          20.9k/20.9k [00:00<00:00, 2.40MB/s]
Fetching 2 files: 100%                      2/2 [01:09<00:00, 69.92s/it]
model-00001-of-00002.safetensors: 100%      4.97G/4.97G [01:09<00:00, 81.8MB/s]
model-00002-of-00002.safetensors: 100%      1.46G/1.46G [01:02<00:00, 8.82MB/s]
Loading checkpoint shards: 100%             2/2 [00:25<00:00, 11.66s/it]
generation_config.json: 100%                189/189 [00:00<00:00, 17.2kB/s]
Device set to use cuda:0
The following generation flags are not valid and may be ignored: ['temperature', 'top_p']. Set `TRANSFORMERS_V
```

```
import pandas as pd
from sklearn.metrics import classification_report

def final_robust_extraction(output_item):
    """
    Extracts the label by checking the structure of the pipeline output.
    """
    try:
        # In newer pipelines, the output is a list where the last element
        # is the assistant's dictionary: {'role': 'assistant', 'content': '...'}
        generated_text = ""

        if isinstance(output_item, list):
            generated_text = output_item[-1]['generated_text']
            # If generated_text is still a list (conversation), get the last content
            if isinstance(generated_text, list):
                generated_text = generated_text[-1]['content']

        generated_text = str(generated_text).upper()

        if "SUPPORTS" in generated_text:
            return "SUPPORTS"
```

```
            if "REFUTES" in generated_text:
                return "REFUTES"
            return "UNKNOWN"
        except Exception:
            return "ERROR"

    # 1. Apply the corrected extraction
    df_baseline['prediction'] = [final_robust_extraction(res) for res in raw_results]

    # 2. Check the distribution to ensure we don't have only ERROR/UNKNOWN
    print("New Prediction Distribution:")
    print(df_baseline['prediction'].value_counts())

    # 3. Calculate Metrics (Precision, Recall, F1)
    print("\n### 1 - Baseline Zero-shot Final Results ###")
    # We only evaluate against known labels SUPPORTS and REFUTES
    print(classification_report(df_baseline['label'], df_baseline['prediction'], labels=['SUPPORTS', 'REFUTES']))
```

```
New Prediction Distribution:
prediction
REFUTES     351
SUPPORTS    149
Name: count, dtype: int64

### 1 - Baseline Zero-shot Final Results ###
              precision    recall  f1-score   support

    SUPPORTS       0.79      0.47      0.59       250
     REFUTES       0.62      0.88      0.73       250

    accuracy                           0.67       500
   macro avg       0.71      0.67      0.66       500
weighted avg       0.71      0.67      0.66       500
```

## Analysis and Comments:

**When running the baseline model for the first time, a couple of technical issues arose** that are common when working with Llama-type architectures and chat pipelines. The first difficulty was a complete disconnect between what the model generated and what the code attempted to process. Since Llama is a decoder-only model, leaving padding on the right causes the model to try to predict the next sequence based on empty tokens, which generates inconsistencies. It was necessary to move the padding to the left so that the claim content would be adjacent to the beginning of the assistant's response. Additionally, when using chat templates, the model returned a message structure instead of plain text, which caused the code to fail when attempting to process a list as if it were a string. The processing had to be modified to correctly extract the generated content.

**Once these errors were resolved, the results showed** that the model exhibits a notable bias toward the REFUTES classification, with a recall of 0.88 in this category, while in SUPPORTS it barely reaches 0.47. This behavior can be explained by the fact that, lacking external evidence such as Wikipedia, the model tends to classify as false those claims about which it has no absolute certainty, rather than risking validating them incorrectly. It is, in a sense, hallucinating refutations due to lack of context.

**The overall accuracy of 67% demonstrates that the model cannot operate solely with its internal knowledge. It is essential to implement the RAG system to provide it with verifiable evidence and correct this bias toward excessive skepticism.**

```
Empieza a programar o a crear código con IA.
```

## 2 - Creación de base de conocimiento (10 puntos)

## 2 - Knowledge Base Creation

Use the `meta-llama/Llama-3.2-3B-Instruct` model to act as a Named Entity Recognition (NER) extractor. The goal is to obtain the exact search term for Wikipedia.

For each claim in the dataset, extract the main entity (subject) mentioned. Design an appropriate prompt for this task and justify your design choices.

Once you have extracted the entities, use the Wikipedia API to download the content of the corresponding articles. Implement the following strategy:

- Use `RecursiveCharacterTextSplitter` to segment the articles into smaller chunks.
- Configure a `chunk_size` of 600 characters with an `overlap` of 60 characters.

- Create vector embeddings for each chunk using `HuggingFaceEmbeddings`.
- Store the chunks in a FAISS vector database for efficient retrieval.

Document the number of unique entities retrieved and the total number of document chunks created. Justify the segmentation parameters chosen (chunk size and overlap).

**Observations**

- Some entities may not be retrievable from Wikipedia. This is expected and should be documented.
- The Wikipedia API may fail for certain queries. Implement appropriate error handling.
- Processing time will vary depending on the number of entities and API response times.

## 1 Extract_entities_batch

For entity extraction, I will use the Llama model as a Named Entity Recognition (NER) extractor. The goal is to obtain the exact search term we need to query Wikipedia.

**Prompt Design Justification:**

The prompt is designed to be simple and direct, instructing the model to extract only the main subject or entity from each claim without any additional explanation or context. This approach ensures:

- **Precision**: By requesting only the entity name, we avoid receiving unnecessary text that would complicate Wikipedia searches.
- **Consistency**: A structured prompt helps maintain uniform output format across all claims.
- **Efficiency**: Minimal output reduces processing time and token usage.

The prompt explicitly asks for "just the name, nothing else" to prevent the model from generating explanations or additional commentary that would interfere with the Wikipedia API queries.

```
def extract_entities_batch(claims, pipeline_model):
    """
    Uses Llama-3.2 to extract the main entity from each claim using a robust extraction logic.
    """
    extraction_prompts = [
        [
            {"role": "system", "content": "Extract the main subject or entity from the claim. Answer ONLY wi
            {"role": "user", "content": f"Claim: {claim}"}
        ]
        for claim in claims
    ]

    print(f"Extracting entities for {len(claims)} claims...")
    raw_outputs = pipeline_model(extraction_prompts, batch_size=16, max_new_tokens=15)

    entities = []
    for output in raw_outputs:
        try:
            # Navigate the list to find the content of the assistant's message
            # The structure is usually output[-1]['generated_text'] which is a list of dicts
            assistant_msg = output[-1]['generated_text']
            # If it's a list, get the content of the last message
            entity_name = assistant_msg[-1]['content'].strip() if isinstance(assistant_msg, list) else assis
            entities.append(entity_name)
        except Exception:
            entities.append("None")

    return entities

# Execution
df_baseline['entity'] = extract_entities_batch(df_baseline['claim'].tolist(), llm_classifier)

# Review the extracted entities
print(df_baseline[['claim', 'entity']].head(10))
```

```
Extracting entities for 500 claims...
                                               claim               entity
0  An American producer was the writer of the pil...    American producer
1              Kate Nash was dropped from her label.            Kate Nash
2                                Rope was released.                 Rope
3               Jack Nicholson stars in The Shining.       Jack Nicholson
4                Manchester City F.C. was purchased.  Manchester City F.C
5                       The Punisher is a character.             Punisher
6  The name Minneapolis is a combination of a Dak...          Minneapolis
7                       Mukesh Ambani has a brother.        Mukesh Ambani
8  Jamie Foxx's time-point of birth is in the 20t...           Jamie Foxx
```

```
9  Bill Clinton is a politician from the United S...         Bill Clinton
```

## 2. build_wikipedia_knowledge_base

```
from langchain_community.document_loaders import WikipediaLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from tqdm import tqdm

def build_wikipedia_knowledge_base(entities):
    """
    Fetches Wikipedia articles for a list of entities and splits them into chunks.
    """
    unique_entities = list(set(entities))
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=600,
        chunk_overlap=60,
        length_function=len
    )

    knowledge_data = []
    print(f"Fetching Wikipedia content for {len(unique_entities)} unique entities...")

    for entity in tqdm(unique_entities):
        try:
            # We fetch only the top 1 result per entity as per instructions
            loader = WikipediaLoader(query=entity, load_max_docs=1)
            docs = loader.load()

            if docs:
                # Split the document into manageable chunks
                chunks = text_splitter.split_documents(docs)
                for chunk in chunks:
                    knowledge_data.append({
                        "entity": entity,
                        "content": chunk.page_content,
                        "source": chunk.metadata.get('source', 'unknown')
                    })
        except Exception:
            # Skip if the entity is not found or API fails
            continue

    return pd.DataFrame(knowledge_data)

# This process may take around 15-20 minutes
df_knowledge_base = build_wikipedia_knowledge_base(df_baseline['entity'].tolist())

# Save the chunks to avoid repeating the search later
df_knowledge_base.to_csv("wikipedia_chunks.csv", index=False)
print(f"\nKnowledge base created with {len(df_knowledge_base)} chunks.")
```

```
Fetching Wikipedia content for 469 unique entities...
 95%|████████▌ | 445/469 [10:17<00:28,  1.17s/it]
```

## 3. FAISS index

```
# !pip install -q langchain-core
```

```
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document

# 1. Initialize the embedding model
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

# 2. Convert our DataFrame chunks into Document objects
print("Converting dataframe to documents...")
documents = [
    Document(
        page_content=str(row['content']),
        metadata={"source": row['source'], "entity": row['entity']}
    )
    for _, row in df_knowledge_base.iterrows()
]
```

```
# 3. Create and save the FAISS index
print(f"Indexing {len(documents)} chunks into FAISS. Please wait...")
vector_store = FAISS.from_documents(documents, embedding_model)

# 4. Save locally
vector_store.save_local("faiss_fever_index")

print("✅ FAISS Vector Store is now ready with the correct imports!")
```

```
modules.json: 100%                                                  349/349 [00:00<00:00, 13.3kB/s]
config_sentence_transformers.json: 100%                             116/116 [00:00<00:00, 3.55kB/s]
README.md:         10.5k/? [00:00<00:00, 541kB/s]
sentence_bert_config.json: 100%                                     53.0/53.0 [00:00<00:00, 2.72kB/s]
config.json: 100%                                                   612/612 [00:00<00:00, 26.8kB/s]
model.safetensors: 100%                                             90.9M/90.9M [00:01<00:00, 68.2MB/s]
tokenizer_config.json: 100%                                         350/350 [00:00<00:00, 12.9kB/s]
vocab.txt:         232k/? [00:00<00:00, 3.17MB/s]
tokenizer.json:         466k/? [00:00<00:00, 5.81MB/s]
special_tokens_map.json: 100%                                       112/112 [00:00<00:00, 3.14kB/s]
config.json: 100%                                                   190/190 [00:00<00:00, 4.23kB/s]
Converting dataframe to documents...
Indexing 4670 chunks into FAISS. Please wait...
✅ FAISS Vector Store is now ready with the correct imports!
```

## Analysis and Comments:

The first important decision was to use RecursiveCharacterTextSplitter instead of a simple splitter that cuts text every N characters arbitrarily. This splitter works hierarchically: it attempts to divide first by paragraphs, then by sentences, and finally by words. This is crucial because it ensures that evidence is not fragmented incoherently. If a sentence that says "Jack Nicholson was not born in..." ends in one fragment and the continuation "...but in New Jersey" remains in another, the model would lose the complete context. ***This approach keeps ideas intact within each block.***

I then configured a chunk_size of 600 characters with an overlap of 60. The 600 characters are sufficient to capture between three and four sentences from Wikipedia, which represents an ideal balance: they contain enough information to provide context without introducing unnecessary noise that would distract the model. FEVER claims are very atomic, focused on a single idea, so larger fragments would be counterproductive. ***The 10% overlap functions as a safety mechanism: if an important entity appears at the end of a fragment, the overlap guarantees that it will also be present at the beginning of the next one, preserving semantic continuity.***

The choice of FAISS over traditional keyword search methods responds to the need to perform searches by meaning and not just by exact lexical matches. FAISS allows that, if the claim mentions "The movie was a success" and Wikipedia contains "The film was a blockbuster", the system recognizes the semantic equivalence thanks to the proximity of their vector representations in the embedding space.

It is important to mention the limitations encountered. As the assignment indicates, I could not retrieve some entities, going from 500 claims to 469 unique entities. This occurs when the model extracts subjects that do not have an exact Wikipedia page or when the API fails in the query. ***However, with 4,670 indexed fragments, I have a sufficiently robust library to significantly improve the model's performance. Ultimately, I went from a system that depended exclusively on its internal memory to one backed by a knowledge base of 4,670 semantically organized documents.***

# 3 - Aplicación de RAG

```
import torch

def get_rag_prediction(claim, vector_db, pipeline_model):
    """
    Retrieves context from FAISS and generates a prediction using the LLM.
    """
    # 1. Retrieval: Search for the 3 most similar chunks
    # k=3 provides enough evidence without overwhelming the model's context window
    related_docs = vector_db.similarity_search(claim, k=3)
    context_text = "\n---\n".join([doc.page_content for doc in related_docs])
```

```
# 2. Prompt Construction
# We explicitly tell the model to prioritize the provided evidence.
messages = [
    {
        "role": "system",
        "content": (
            "You are a factual verification assistant. Use the provided context from Wikipedia "
            "to determine if the claim is supported or refuted. Answer ONLY with the word "
            "'SUPPORTS' or 'REFUTES'. If the information is not enough, use your best judgment "
            "based on the evidence."
        )
    },
    {
        "role": "user",
        "content": f"CONTEXT FROM WIKIPEDIA:\n{context_text}\n\nCLAIM: {claim}\n\nVERDICT:"
    }
]

# 3. Inference
output = pipeline_model(messages, max_new_tokens=10)

# 4. Parsing the response
try:
    raw_content = output[-1]['generated_text']
    prediction = raw_content[-1]['content'].strip().upper() if isinstance(raw_content, list) else raw_co

        if "SUPPORTS" in prediction: return "SUPPORTS"
        if "REFUTES" in prediction: return "REFUTES"
        return "UNKNOWN"
    except:
        return "ERROR"

# Execution over the 500 claims
print(f"Starting RAG inference for {len(df_baseline)} claims...")
rag_results = []

for claim in tqdm(df_baseline['claim']):
    res = get_rag_prediction(claim, vector_store, llm_classifier)
    rag_results.append(res)

df_baseline['rag_prediction'] = rag_results
```

```
Starting RAG inference for 500 claims...
  1%|          | 6/500 [00:06<08:39,  1.05s/it]You seem to be using the pipelines sequentially on GPU. In ord
100%|██████████| 500/500 [09:12<00:00,  1.10s/it]
```

```
from sklearn.metrics import classification_report

# We evaluate the RAG predictions against the original labels
print("\n### 3 - RAG Performance Metrics (Final) ###")
print(classification_report(df_baseline['label'], df_baseline['rag_prediction'], labels=['SUPPORTS', 'REFUTE

# Comparison logic
baseline_acc = (df_baseline['label'] == df_baseline['prediction']).mean()
rag_acc = (df_baseline['label'] == df_baseline['rag_prediction']).mean()

print(f"--- Comparison Summary ---")
print(f"Overall Accuracy Baseline: {baseline_acc:.4f}")
print(f"Overall Accuracy RAG:      {rag_acc:.4f}")
print(f"Net Improvement:           {((rag_acc - baseline_acc) * 100):.2f}%")
```

```
### 3 - RAG Performance Metrics (Final) ###
              precision    recall  f1-score   support

    SUPPORTS       0.95      0.51      0.66       250
     REFUTES       0.67      0.97      0.79       250

    accuracy                           0.74       500
   macro avg       0.81      0.74      0.73       500
weighted avg       0.81      0.74      0.73       500

--- Comparison Summary ---
Overall Accuracy Baseline: 0.6740
Overall Accuracy RAG:      0.7420
Net Improvement:           6.80%
```

## Analysis and Comments:

When comparing the metrics obtained with the RAG system versus the Baseline, significant changes are observed in the model's reasoning capability. The Recall of the SUPPORTS class increased from 0.47 to 0.51, an improvement that, although moderate, is accompanied by a jump in Precision to 0.95. This indicates that when the RAG system confirms a claim as true, it rarely makes mistakes. The model stopped guessing and now only marks something as SUPPORTS when the evidence from Wikipedia is explicit and conclusive.

On the other hand, the Recall of REFUTES reached 0.97, which demonstrates that the model became an extremely effective detector of falsehoods. With the available context, if the data in Wikipedia contradicts the claim, the model refutes it immediately. However, the precision in this class is lower (0.67) because the model tends to classify as REFUTES those cases where the retrieved information is insufficient, adopting a conservative behavior in the face of uncertainty.

The macro F1-score increased from 0.66 to 0.73, which is the most relevant indicator as it balances precision and recall. This increase demonstrates that the RAG system is considerably more reliable and balanced than the model operating in isolation. The vector knowledge base effectively anchored the model to reality, allowing it to base its decisions on verifiable evidence.

The strategy of using FAISS with 600-character fragments proved successful. The 6.80% improvement in overall accuracy (going from 0.6740 to 0.7420) validates that, for fact-checking tasks, the latent knowledge of a 3B parameter LLM is not sufficient. The retrieval of external evidence is what transforms a "probable opinion" into a robust "factual verification".

## Experiment

To verify if the knowledge base truly works.

## Testing the claims

Create a simple function where you can write any claim and the system shows what it found in Wikipedia before giving the answer. This allows you to see if the failure is in the search (FAISS) or in the reasoning (Llama).

```
def test_your_rag(claim):
    # 1. Ver qué recupera FAISS
    docs = vector_store.similarity_search(claim, k=2)

    print(f"\n TEST CLAIM: {claim}")
    print("-" * 30)
    print(" WIKIPEDIA EVIDENCE FOUND:")
    for i, doc in enumerate(docs):
        print(f"Chunk {i+1}: {doc.page_content[:200]}...")

    # 2. Ver qué responde el modelo
    prediction = get_rag_prediction(claim, vector_store, llm_classifier)
    print("-" * 30)
    print(f" MODEL VERDICT: {prediction}")

# --- PRUEBAS SUGERIDAS ---
# 1. Prueba un hecho real que debería estar en tus chunks
test_your_rag("Jack Nicholson is an actor.")

# 2. Prueba a cambiar un detalle (debería decir REFUTES)
test_your_rag("Jack Nicholson is a professional soccer player for Manchester City.")

# 3. Prueba algo que NO esté en tu base de datos (para ver cómo reacciona)
test_your_rag("The moon is made of green cheese.")
```

```
 TEST CLAIM: Jack Nicholson is an actor.
------------------------------
 WIKIPEDIA EVIDENCE FOUND:
Chunk 1: John Joseph Nicholson (born April 22, 1937) is an American actor and filmmaker. Nicholson is widely
Chunk 2: John Joseph Nicholson was born on April 22, 1937, in Neptune City, New Jersey, the son of a showgirl
------------------------------
 MODEL VERDICT: SUPPORTS

 TEST CLAIM: Jack Nicholson is a professional soccer player for Manchester City.
------------------------------
 WIKIPEDIA EVIDENCE FOUND:
Chunk 1: John Joseph Nicholson (born April 22, 1937) is an American actor and filmmaker. Nicholson is widely
Chunk 2: Donald Rose; 1909–1997) in 1936, before realizing that he was already married. Biographer Patrick Mc
------------------------------
 MODEL VERDICT: REFUTES
```

```
 TEST CLAIM: The moon is made of green cheese.
------------------------------
 WIKIPEDIA EVIDENCE FOUND:
Chunk 1: female. In PIE mythology, the Moon, which is a male figure, was seen as forming a pair—usually wedlo
Chunk 2: The original PIE moon deity has been reconstructed as *Mehɪnot (from which 'Mene', Selene's byname, 
------------------------------
 MODEL VERDICT: REFUTES
```

## Contradiction stress experiment and extreme ambiguity experiment.

These are key to see if the model really "reads" or if it gets carried away by what it already knows.

```
# --- EXPERIMENTO DE ESTRÉS (CAMBIO DE CONTEXTO) ---
# Intentaremos engañar al modelo con un hecho histórico real pero alterado.
# Bill Clinton fue el 42º presidente, vamos a decirle que fue el primero.
print("EXPERIMENT 2: STRESS TEST (CONTEXT CONTRADICTION)")
test_your_rag("Bill Clinton was the 1st President of the United States.")

print("\n" + "="*50 + "\n")

# --- EXPERIMENTO 3: AMBIGÜEDAD Y FALTA DE ENTIDAD ---
# Usaremos una afirmación muy genérica para ver qué recupera FAISS
# y cómo reacciona el modelo ante la falta de datos específicos.
print("EXPERIMENT 3: EXTREME AMBIGUITY")
test_your_rag("The individual is a high-ranking politician.")
```

```
EXPERIMENT 2: STRESS TEST (CONTEXT CONTRADICTION)

 TEST CLAIM: Bill Clinton was the 1st President of the United States.
------------------------------
 WIKIPEDIA EVIDENCE FOUND:
Chunk 1: William Jefferson  Clinton (né Blythe III; born August 19, 1946) is an American politician and lawye
Chunk 2: As a trusted member of President Washington's first cabinet, Hamilton served as the first U.S. secre
------------------------------
 MODEL VERDICT: REFUTES

==================================================

EXPERIMENT 3: EXTREME AMBIGUITY

 TEST CLAIM: The individual is a high-ranking politician.
------------------------------
 WIKIPEDIA EVIDENCE FOUND:
Chunk 1: == Politics ==...
Chunk 2: guided by the parliamentarian. In the early 1920s, the practice of majority and minority parties ele
------------------------------
 MODEL VERDICT: SUPPORTS
```

***NOTE:***

Through the stress tests, it was demonstrated that the RAG system transforms the model into a logical verifier capable of detecting numerical contradictions (such as the Bill Clinton case). However, it was identified that ambiguity in the query is the system's main weakness: if the vector database retrieves generic information due to lack of a clear subject, the model loses its critical capacity and tends to support vague claims. This underscores that the effectiveness of RAG does not depend only on the LLM, but on the precision in entity extraction and the quality of the retrieved fragments.

# 4 Conclusions

## Comparative Results Table

| Metric | Baseline (Zero-shot) | RAG (Wikipedia + FAISS) | Absolute Difference |
|---|---|---|---|
| Overall Accuracy | 67.40% | 74.20% | +6.80% |
| Precision (SUPPORTS) | 0.79 | 0.95 | +0.16 |
| Recall (SUPPORTS) | 0.47 | 0.51 | +0.04 |
| Recall (REFUTES) | 0.88 | 0.97 | +0.09 |
| F1-Score (Macro) | 0.66 | 0.73 | +0.07 |

## Analysis of Observed Differences

### In which cases does RAG outperform the Baseline?

RAG significantly outperforms the Baseline in the reliability of true claims. While the Baseline achieved a precision of 0.79, RAG reached 0.95. This means that when the RAG system classifies something as true, there is 95% confidence in its verdict. The Baseline failed because it attempted to recall specific data from its training, while RAG functions as a proof validator that contrasts directly with the original source.

**Is there any scenario where RAG could fail or not provide value?**

There are three main limitations. First, semantic ambiguity: as observed in the manual experiments, if the claim is vague (for example, "The individual..."), the retrieval system may bring generic information that the model incorrectly accepts as valid. Second, dependency on entity extraction: the system critically depends on the NER process from Step 2. If Llama fails to correctly identify the subject, RAG has nothing to search for and remains limited to its internal memory. Third, very recent facts: if the vector database is not updated, RAG may refute true facts simply because they do not appear in its library, acting under "administrative silence."

## Conclusion on the Importance of Information Retrieval

The conclusion is clear: memory is not enough for truth. In fact-checking tasks, relying solely on a model's pre-trained weights is inadequate because models tend to be skeptical out of ignorance or hallucinatory out of creativity. The implementation of the RAG architecture proved to be the necessary bridge between the model's linguistic capacity and factual precision. By adding an external retrieval layer with FAISS, I achieved that a model with only 3B parameters behaves like a documented expert, drastically reducing hallucinations and improving overall Accuracy by 6.8%. RAG is not just a technical improvement, but a fundamental necessity for any AI system that must interact with verifiable factual information.