## **1. Traditional Information Retrieval Approach** Don Krasniqi 121410189

### **Method Overview**

For the traditional Information Retrieval (IR) model, we implemented a retrieval pipeline based on **BM25**, a classic bag-of-words ranking function that scores documents based on the frequency and rarity of query terms, while accounting for term saturation and document length normalization. BM25 remains a strong baseline in many IR tasks.

We used the rank\_bm25 library, a pure Python implementation of the BM25Okapi algorithm. BM25 ranks documents higher if they contain more of the query terms (especially rare ones) and penalizes very long documents.

Our implementation performs the following steps:

**1. Document Preprocessing:**

* Concatenated the document **title** and **abstract** from the provided CORD-19 collection.
* Tokenized text using NLTK's word\_tokenize.
* Built a corpus of tokenized documents in memory.

**2. Index Building:**

* Constructed a BM25 index using the preprocessed corpus.

**3. Claim Processing and Retrieval:**

* Each tweet (claim) is tokenized similarly.
* The BM25 index is queried using the processed claim.
* The top-k relevant documents (default: **k=5**) are returned based on their BM25 scores.

The full implementation is located in our Jupyter notebook under the Task 1 section and supports integration with the full claim-source retrieval pipeline.

### **Why It Qualifies as a Traditional IR Approach**

The BM25 model is a canonical example of traditional information retrieval because:

* It does **not** rely on neural networks or learned representations.
* It uses **term frequency (TF)** and **inverse document frequency (IDF)** statistics computed directly from the corpus.
* The retrieval process is based entirely on **lexical overlap** between the query and the documents — no embeddings or semantic models are used.
* The approach is **unsupervised** and parameter-free, apart from optional hyperparameters like k1 and b.

This makes it a textbook example of a traditional IR system in the context of this project.

### **Evaluation Setup & Performance**

Since we missed the official CLEF deadline and could not submit results to Codalab, we implemented our **own local evaluation setup**:

* We **merged the official train and dev splits** from the dataset.
* To avoid data leakage and overfitting, we removed any **duplicate queries**.
* We performed a **random 80/20 split** to create new **train** and **test** sets.
* The BM25 model was evaluated **only on the new test set**.
* The main metric used for evaluation was **MRR@5**, consistent with the CLEF task guidelines.

### **Results**

On our local test split, the BM25-based system achieved an **MRR@5 of 0.5460**, demonstrating solid baseline performance. This provides a strong foundation for comparison against the team’s neural IR approaches, which aim to improve over this traditional method.

## **2. Neural IR Model**

[insert text here]

## **3. Neural Re-Ranker**

Rita Selimi 12332281

### **Method Overview**

To boost the accuracy of our retrieval results, we built a neural re-ranking system using a fine-tuned CrossEncoder. Unlike traditional methods, CrossEncoders take both the query and document together and look at how they relate, which helps them better understand the context and relevance between them.

We started with the cross-encoder/ms-marco-MiniLM-L-12-v2, a lightweight but surprisingly strong model built for semantic relevance tasks. We then fine-tuned it, tailoring the model more specifically to our domain.

**Training Data Generation**  
To train our CrossEncoder, we needed a dataset of query-document pairs labeled as either relevant or not. We created this using the tweet-document pairs from the training split:

* For each tweet, we added one **positive pair** by matching it with its correct document and labeling it with 1.0.
* For the **negative pair**, we used our BM25 index to retrieve the top 20 candidate documents.
* We skipped the gold document and picked the highest-ranked non-matching one, labeling it 0.0.

This gave us a balanced training set where each tweet had exactly one positive and one negative example. All examples were wrapped in InputExample format to prepare for training.

**Fine-Tuning**   
To train the model, we used the CrossEncoder.fit() function provided by the SentenceTransformers library with the following setup:

* Batch size of 16 to ensure stable gradient updates without overloading memory.
* Only 1 training epoch, since our dataset was relatively small and we wanted to avoid overfitting.
* Warmup steps set to 100 to let the model adjust gradually before full training kicks in.
* The model was saved to ./finetuned\_crossencoder for later use.

We reloaded the saved model from disk for inference during evaluation.

**Re-Ranking Process**   
To test how well our fine-tuned model works, we used it to re-rank the documents retrieved by BM25. Here’s how it went:

* For each tweet in the test set, we pulled the top 50 documents using our BM25 index.
* Then, for each of those 50 documents, we paired the tweet with the document (title + abstract) and passed them together into the CrossEncoder.
* The model gave us a relevance score for each pair.
* We used these scores to sort the documents from most to least relevant.
* Finally, we kept only the top 5 documents per tweet to evaluate how good the model was at ranking the true document near the top.

**Why It Qualifies**

This approach qualifies as a neural re-ranking method because it doesn’t just look at word overlap, it actually learns what makes a document relevant to a tweet. Instead of relying on basic keyword matching, it uses a deep language model to understand the meaning and context of both the tweet and the document. Since we fine-tuned it on our own dataset, the model learned how tweets and biomedical abstracts usually relate, which helped a lot with ranking the most relevant documents higher.

**Evaluation Setup & Performance**

We used the same local evaluation setup described earlier (80/20 train-test split, deduplicated queries). The main metric was again MRR@5.

The CrossEncoder re-ranking significantly improved performance:

* BM25-only MRR@5: **0.5460**
* Fine-tuned CrossEncoder MRR@5: **0.6350**

This shows the clear benefit of leveraging supervised neural modeling over lexical BM25 scoring.

**Model Selection and Experiments**

We tried out two different CrossEncoder models:

* cross-encoder/ms-marco-MiniLM-L-6-v2: it is smaller and runs faster, but we saw lower performance, around 0.58 MRR@5.
* cross-encoder/ms-marco-MiniLM-L-12-v2: this one performed better overall, so we stuck with it as our final choice.

We also experimented with how many BM25 candidates to re-rank (topk). We tested 20, 50, and 100. Topk=50 gave us the best trade-off, it was deep enough to include relevant documents but not too expensive to compute.

Challenges Faced

* We initially encountered environment compatibility issues with tensorflow and keras, due to version conflicts with Transformers. This was solved by enforcing TRANSFORMERS\_NO\_TF=1 and avoiding TensorFlow dependencies.
* Pushing the fine-tuned model to GitHub failed due to large file limits (>100MB). We addressed this by excluding the model folder using .gitignore and documenting it locally instead.

### **Results**

Our neural re-ranking pipeline successfully enhanced retrieval effectiveness by integrating the previous models for recall with a fine-tuned CrossEncoder for precision. Fine-tuning yielded a measurable and significant improvement in MRR@5 over the baseline, with a result of 0.6352, justifying the added complexity.