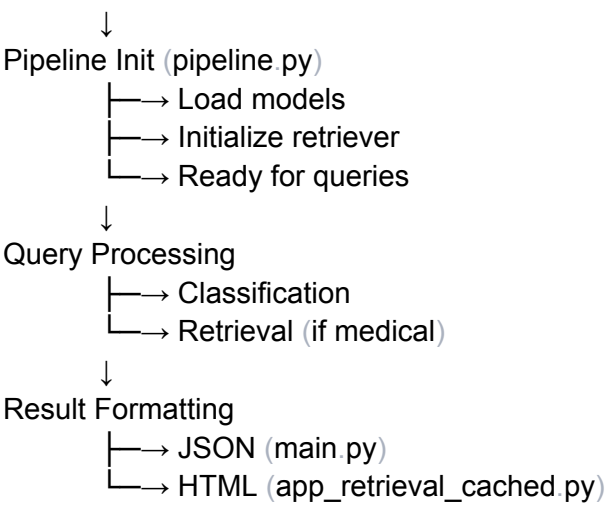
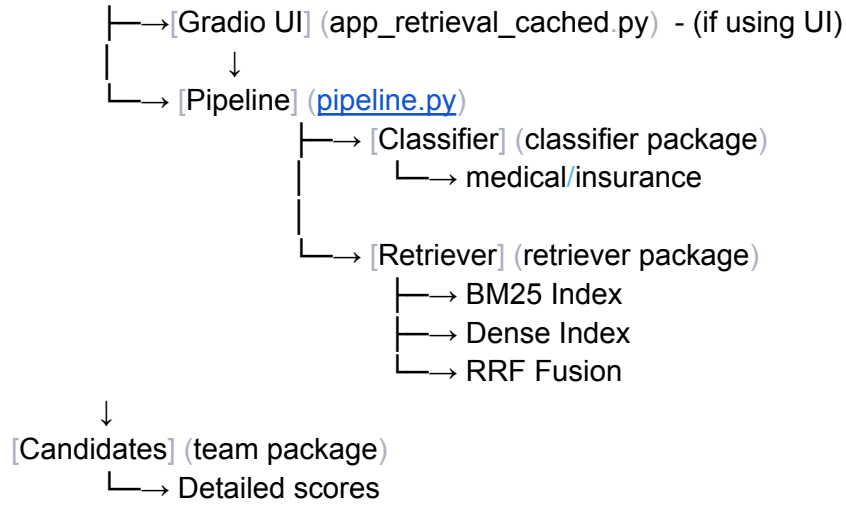
Header/Arch

This new tool is a WebMD style system designed for usage in patient care portals (inquiries). Medical inquiry classification for efficient patient care schedule optimization. If you use MyChart or any other app to make appointments with your doctor you may not know that you can chat with your doctor directly. Usually a member of staff will read it, assess whether they can answer, and then either answer or have the doctor look. This could be replaced with an AI system that can identify the type of inquiry, and if medical returns relevant documents based upon the inquiry. Although not integrated now, the idea is to serve those relevant documents to the doctors that the patients would see as well, so the doctor and patient can be starting on the same page.

Data workflow:



Main Flow:



↓
Formatted Results

Classifier

Classifier Package Documentation

Package Overview

The `classifier` package implements a healthcare query classification and routing system that distinguishes between medical and insurance-related queries, then routes medical queries to appropriate departments based on symptom analysis.

Additional Folder: reason

Files:

- `__init__.py`
- `config.py`
- `head.py`
- `infer.py`
- `modelcard_template.md`
- `query_router.py`
- `train.py`
- `utils.py`

Usage

To perform training, run the following script:

```
python3 -m classifier.train
```

The model will be automatically saved and loaded into main
No other files need to be run directly.

config

Purpose: Environment configuration and API token management

Functions

`load_env()`

Loads environment variables from an `env.list` file in the project root.

Behavior:

- Reads `env.list` line by line
- Skips empty lines and comments (lines starting with `#`)
- Parses `KEY=VALUE` pairs and loads them into `os.environ`

Variables

- **`HF_TOKEN`**: HuggingFace API token loaded from environment variables, used for model downloads and uploads

head

Purpose: Neural network classification head for medical/insurance query classification

Class: `ClassifierHead`

A PyTorch neural network module that serves as the classification layer on top of sentence embeddings.

Inheritance

- `nn.Module`: PyTorch neural network base class
- `PyTorchModelHubMixin`: Enables easy HuggingFace Hub integration

Architecture

Input: 768-dimensional sentence embeddings (from EmbeddingGemma-300M)

Layers:

1. Linear (768 → 512) + ELU activation + 50% Dropout
2. Linear (512 → 512) + ELU activation + 50% Dropout
3. Linear (512 → num_classes)
4. Softmax for probability output

Methods

`__init__(num_classes: int, embedding_dim: int = 768)`

Initializes the classification head.

Parameters:

- `num_classes`: Number of output categories (2 for medical/insurance)
- `embedding_dim`: Dimension of input embeddings (default: 768)

`forward(features: Dict[str, torch.Tensor]) -> Dict[str, torch.Tensor]`

Standard PyTorch forward pass for training.

Parameters:

- `features`: Dictionary containing `'sentence_embedding'` key with embeddings

Returns:

- Dictionary with `'logits'` key containing raw classification scores

---

`predict(embeddings: torch.Tensor) -> torch.Tensor`

Predicts class labels from embeddings.

Parameters:

- `embeddings`: Tensor of shape `[batch_size, embedding_dim]`

Returns:

- Integer labels tensor of shape `[batch_size]`

`predict_proba(embeddings: torch.Tensor) -> torch.Tensor`

Predicts class probabilities from embeddings.

Parameters:

- `embeddings`: Tensor of shape `[batch_size, embedding_dim]`

Returns:

- Probability tensor of shape `[batch_size, num_classes]` (sums to 1.0 per row)

Note: Automatically switches to eval mode and back to training mode

---

`get_loss_fn() -> nn.Module`

Returns the appropriate loss function for training.

Returns:

- `nn.CrossEntropyLoss()` instance

HuggingFace Hub Metadata

- **Repository:** `davidgray/health-query-triage`

- **Pipeline tag:** `text-classification`
- **Tags:** `medical`, `classification`

infer

Purpose: Inference pipeline for classifying healthcare queries

Functions

`classifier_init(checkpoint_path: str | None, model_id: str | None) -> tuple`

Initializes the embedding model and classifier head.

Parameters:

- `checkpoint_path`: Path to local checkpoint directory (optional)
- `model_id`: HuggingFace model ID (default: `CLASSIFIER_NAME`)

Returns:

- Tuple of `(embedding_model, classifier_head)`

Behavior:

- If `checkpoint_path` provided: loads latest checkpoint from directory
- Otherwise: loads from HuggingFace Hub using `model_id`

`predict_query(text: list[str], embedding_model: SentenceTransformer, classifier_head: ClassifierHead) -> dict`

Runs the complete inference pipeline on one or more queries.

Parameters:

- `text`: List of query strings
- `embedding_model`: Initialized SentenceTransformer model
- `classifier_head`: Initialized ClassifierHead model

Returns: Dictionary containing:

- `'prediction'`: List of predicted category labels
- `'confidence'`: Confidence scores for predictions
- `'probabilities'`: Full probability distribution across all classes

Pipeline:

1. Sets models to evaluation mode
2. Encodes text to embeddings using SentenceTransformer
3. Computes probabilities using classifier head
4. Extracts predicted labels and confidence scores

`test(local: bool = False)`

Demonstrates the inference system with example queries.

Parameters:

- `local`: If True, uses local checkpoint; otherwise uses HuggingFace model

train

Purpose: Training pipeline for the medical/insurance classifier

Data Preparation Functions

`even_split(prefix: str, target: int, splits: int, total: int) -> str`

Creates evenly distributed dataset split strings for HuggingFace datasets.

Parameters:

- `prefix`: Split name (e.g., "train")
- `target`: Target number of samples
- `splits`: Number of split chunks
- `total`: Total dataset size

Returns:

- Dataset split string like `"train[0:500]+train[1000:1500]+..."`

Purpose: Efficiently sample large datasets by taking distributed chunks

---

`get_model_train_test() -> tuple`

Loads and prepares training datasets.

Datasets Used:

1. **MIRIAD (Medical)**
   - Source: `tomaarsen/miriad-4.4M-split`
   - Training samples: 50,000 (from 4.47M total)
   - Label: "medical"
   - Fields: question text
2. **InsuranceQA (Insurance)**
   - Source: `deccan-ai/insuranceQA-v2`
   - Training samples: 5,000 (from 21,300 total)
   - Label: "insurance"
   - Fields: input text

Process:

1. Loads datasets with even splits
2. Adds static labels ("medical" or "insurance")

3. Interleaves datasets randomly
4. Removes duplicate texts
5. Creates train/test/validation splits

Returns:

- `embedding_model`: SentenceTransformer model
- `classifier`: ClassifierHead model
- `train_ds`, `test_ds`, `validation_ds`: Dataset splits
- `CATEGORIES`: Label names

Training Functions

`label_to_int(embedding_model, label_names: list) -> callable`

Creates a custom collate function for DataLoader.

Returns:

- Collate function that:
    1. Extracts texts and labels from batch
    2. Encodes texts to embeddings using the embedding model
    3. Converts string labels to integers
    4. Returns dictionary with embeddings and labels on correct device

---

`train_loop(dataloader, model, loss_fn, optimizer, batch_size=64, epochs=10) -> tuple`

Executes one training epoch.

Returns:

- `total_loss`: Cumulative loss for the epoch
- `batch_losses`: List of loss values per batch

Process:

1. Sets model to training mode
2. For each batch:
    - Zero gradients

    - Forward pass (embeddings → logits)
    - Calculate loss
    - Backward pass
    - Update weights
3. Logs progress every 100 batches

`test_loop(dataloader, model, loss_fn) -> tuple`

Evaluates model performance on validation/test set.

Returns:

- `avg_loss`: Average loss across all batches
- `accuracy`: Classification accuracy

Process:

1. Sets model to evaluation mode
2. Disables gradient computation
3. Computes predictions and metrics

Model Card Functions

`generate_model_card(save_dir: str, accuracy: float, loss: float, epoch: int)`

Generates a HuggingFace model card from template.

Populates:

- Model description and metadata
- Training data information
- Performance metrics
- Base model information

`push_model_card(save_dir: str, repo_id: str, token: str)`

Uploads model card to HuggingFace Hub.

Main Training Function

`train(push_to_hub: bool = False)`

Complete training pipeline.

Configuration:

- Batch size: 64
- Epochs: 1 (with early stopping)
- Learning rate: 1e-3
- Weight decay: 1e-5
- Optimizer: Adam
- Early stopping patience: 1 epoch

Process:

1. Creates checkpoint directory with timestamp
2. Loads datasets and models
3. Creates DataLoaders with custom collate function
4. Training loop with validation
5. Early stopping based on validation loss
6. Final test set evaluation
7. Saves model, history CSV, and loss plot
8. Optional push to HuggingFace Hub

Outputs:

- `checkpoints/YYYYMMDD_HHMMSS/`: Saved model and artifacts
- `history.csv`: Training metrics per epoch
- `loss.png`: Training loss plot

utils

Purpose: Shared utilities and configuration for the classification system

Constants

Model Configuration

- **`MODEL_NAME`**: `"sentence-transformers/embeddinggemma-300m-medical"`
  - Base embedding model (768-dimensional output)
- **`CLASSIFIER_NAME`**: `"davidgray/health-query-triage"`
  - HuggingFace repository for trained classifier
- **`CATEGORIES`**: `["medical", "insurance"]`
  - Classification labels

File System

- **`CHECKPOINT_PATH`**: `"classifier/checkpoints"`
  - Directory for saving model checkpoints
- **`DATETIME_FORMAT`**: `"%Y%m%d_%H%M%S"`
  - Timestamp format for checkpoint directories

Device Configuration

- **`DEVICE`**: Automatically detected compute device
  - Priority: Accelerator (MPS/CUDA) → CPU
  - Uses modern PyTorch accelerator API with fallback for older versions

---

Functions

`get_models(model_id: str | None = None, num_labels: int = 2) -> tuple`

Loads the embedding model and classification head.

Parameters:

- `model_id`: HuggingFace model ID or local path (optional)
- `num_labels`: Number of classification categories (default: 2)

Returns:

- Tuple of `(embedding_model, classifier_head)`

SentenceTransformer Prompts: The embedding model is configured with three prompt types:

1. **classification**: `"task: classification | query: "`
2. **retrieval (query)**: `"task: search result | query: "`
3. **retrieval (document)**: `"title: {title | "none"} | text: "`

Default prompt: `classification`

Behavior:

- If `model_id` provided: loads pretrained classifier from that location
- Otherwise: initializes new ClassifierHead with random weights
- Moves both models to the appropriate device (GPU/MPS/CPU)

`get_latest_checkpoint(checkpoint_path: str) -> str`

Returns path to the most recent checkpoint in a directory.

Parameters:

- `checkpoint_path`: Directory containing timestamped checkpoint folders

Returns:

- Full path to the latest checkpoint (sorted alphabetically by timestamp)

Error Handling

The module includes comprehensive error handling:

- Catches model loading failures
- Provides helpful error messages
- Falls back to older PyTorch device detection if needed
- Raises `RuntimeError` with guidance if model download fails

Adapters

Adapter Package Documentation

Package Overview

The `adapter` package automates the download and preprocessing of multiple medical and scientific datasets into a unified JSONL format. It's designed to build a comprehensive corpus for medical question-answering and information retrieval tasks.

Usage

Pubmed is pulled into build_corpora, and does not need to be called directly although it could be directly called.
To use build_corpora to build the corpuses, run the following:

```
python3 adapters/build_corpora.py
```

build_corpora

Build_corpora.py:

This script automates the download and preprocessing of multiple medical and scientific datasets into a unified JSONL format. It's designed to build a comprehensive corpus for medical question-answering and information retrieval tasks.

Output Structure

All datasets are saved to `data/corpora/` as JSONL files (JSON Lines format), where each line is a valid JSON object.

All JSONL files follow this structure:

{"id": "dataset:identifier", "field1": "value1", ...}

{"id": "dataset:identifier", "field1": "value1", ...}

Datasets

1. LasseRegin Medical Q&A (`medical_qa.jsonl`)

Source: iCliniq medical question-answer pairs from GitHub repository

Fields:

- `id`: Unique identifier (format: `icliniq:{index}`)
- `title`: Question title
- `question`: Full question text
- `answer`: Medical professional's answer
- `source`: Always "icliniq"

Data Source: Raw JSON from LasseRegin's medical-question-answer-data repository

2. MIRIAD-4.4M (`miriad_text.jsonl`)

Source: MIRIAD (Medical Information Retrieval in Abdominal Diseases) dataset via HuggingFace

Configuration:

- Default sample size: 1,000 records (configurable)

- Uses test split with shuffling (seed=42)

Fields:

- `id`: Unique identifier (format: `miriad:{index}`)
- `title`: Document title
- `question`: Query or question
- `answer`: Relevant passage text

3. PubMed Abstracts (`pubmed.jsonl`)

Source: NCBI PubMed 2025 Annual Baseline

Configuration:

- Maximum records: 500,000 (configurable)
- Filters out entries without title or abstract

Fields:

- `id`: Unique identifier (format: `pubmed:{PMID}`)
- `title`: Article title
- `text`: Combined title and abstract

4. UniDoc-Bench (`unidoc_qa.jsonl`)

Source: Salesforce UniDoc-Bench dataset for document understanding

Configuration:

- Maximum items: 1,000 (configurable)
- Uses test split

Fields:

- `id`: Unique identifier (format: `unidoc:{index}`)
- `title`: Generated from domain field (e.g., "Medical PDF")
- `question`: Question or query about the document
- `answer`: Ground truth answer
- `pdf_path`: Path to source PDF document

Functions

`write_jsonl(path, rows)`

Utility function that writes a list of dictionaries to a JSONL file with progress feedback.

`build_lasseregin()`

Downloads and processes iCliniq Q&A data from GitHub.

`build_miriad(sample_size=200_000)`

Loads MIRIAD dataset from HuggingFace, shuffles, and samples specified number of records.

`build_pubmed(max_records=500_000)`

Streams PubMed abstracts, combining titles and abstracts while filtering empty entries.

`build_unidoc(max_items=1000)`

Loads UniDoc-Bench test set with document Q&A pairs and PDF references.

pubmed

Purpose: Downloads, parses, and indexes PubMed medical literature abstracts

Overview

This module provides utilities for working with the PubMed dataset from NCBI (National Center for Biotechnology Information). It handles the complete workflow of downloading compressed XML files, parsing medical article metadata, and optionally building search indexes.

The `pubmed.py` module provides:

1. **Automated Download:** Fetches PubMed XML files from NCBI FTP
2. **Efficient Parsing:** Extracts article metadata from compressed XML
3. **JSONL Output:** Creates search-ready document format
4. **Resumable Operation:** Skips existing files for interrupted downloads
5. **Low Memory Usage:** Streams data file-by-file
6. **Progress Feedback:** Visual progress bars for long operations

Constants

- **`PUBMED_DATASET_BASE_URL`**: `"https://ftp.ncbi.nlm.nih.gov/pubmed/baseline"`
    - FTP base URL for PubMed baseline dataset
    - Contains regularly updated medical literature
- **`PUBMED_FILE_LIMIT`**: `10`
    - Default limit for number of files to download
    - Each file contains ~30,000 articles

Function: `get_pubmed_dataset_size`

Determines the total number of available PubMed XML files.

Returns:

- Number of unique PubMed XML files available on FTP server
- Returns `0` on error

Process:

1. **Fetch FTP Directory Listing**
2. **Parse Filenames**
    - Matches pattern: `pubmed25n0001.xml.gz`

    - Uses negative lookahead to exclude `.md5` files
3. **Count Unique Files**

Error Handling:

- Catches all exceptions
- Prints error message
- Returns `0` to allow graceful degradation

Function: `download_pubmed_xml`

Downloads PubMed XML files from NCBI FTP server.

Parameters:

- `output_dir`: Directory to save downloaded XML files
- `num_files`: Number of files to download (default: 1)
- `year`: Two-digit year for filename pattern (default: '25' for 2025)

Returns:

- List of file paths to downloaded XML files

Process:

1. Create Output Directory
2. Get Dataset Size
3. Download Files with Progress Bar

Performance:

- Each file: ~100-200 MB compressed
- ~30,000 articles per file
- Download time: ~5-30 seconds per file (network dependent)

Function: `parse_pubmed_to_jsonl`

Parses PubMed XML files and extracts article metadata to JSONL format.

Parameters:

- `xml_files`: List of paths to gzipped XML files
- `output_jsonl`: Output path for JSONL file

Extracted Fields:

Field	XML Path	Description
`id`	`.//PMID`	PubMed ID (unique identifier)
`title`	`.//ArticleTitle`	Article title
`contents`	Title + Abstract	Combined text for indexing

XML Structure:

```
<PubmedArticle>
    <PMID>12345678</PMID>
     <Article>
        <ArticleTitle>Title text here</ArticleTitle>
        <Abstract>
            <AbstractText>Abstract text here</AbstractText>
        </Abstract>
    </Article>
 </PubmedArticle>
```

Output Format (JSONL):

```
{"id": "12345678", "title": "Article title", "contents": "Article title Abstract text"}
```

{"id": "12345679", "title": "Another title", "contents": "Another title Another abstract"}

Process:

1. Open Output File
2. For each XML file:
    - Extract articles
    - Handle missing fields
    - Write to JSONL

Function: `download_pubmed`

Main entry point combining download and parsing operations.

Parameters:

- `output_jsonl`: Path for output JSONL file
- `num_files`: Number of XML files to download and process (default: 1)

Returns:

- None (writes to file)

Process:

1. **Check if Already Downloaded:**
    - Skips download if output file exists
    - Enables idempotent execution
2. **Determine XML Directory:**
    - Places XML files in sibling directory
    - Keeps XML files separate from final output
3. **Download and Parse**

Function: `build_index_cmd`

Generates command for building Lucene search index using Pyserini.

Parameters:

- `input_file`: Path to JSONL corpus file
- `index_dir`: Output directory for index

Returns:

- List of command arguments for subprocess execution

Function: `build_index`

Builds a Lucene search index using Pyserini (optional feature).

Parameters:

- `input_file`: Path to JSONL corpus file
- `index_dir`: Output directory for index
- `cmd_generator`: Function to generate indexing command (default: `build_index_cmd`)

Returns:

- None (creates index on disk)

Process:

1. Check Existing Index
2. Create Directory
3. Generate and Run Command

Function: `main`

Standalone main function for direct script execution (not used by `build_corpora.py`).

Parameters:

- `base_data_dir`: Base directory for data files (default: "data")
- `base_index_dir`: Base directory for indexes (default: "indexes")
- `num_files`: Number of files to download (default: 1)

Process:

1. Downloads PubMed corpus to `data/pubmed/corpus.jsonl`
2. Builds Lucene index to `indexes/pubmed/`

Usage:

```
# Direct execution (not typically used)
python pubmed.py
```

Note: When integrated with `build_corpora.py`, only the `download_pubmed()` function is called, and indexing is skipped.

Retriever

Retriever Package Documentation

Package Overview

The `retriever` package implements a hybrid search system that combines sparse (BM25) and dense (semantic) retrieval methods with optional reranking. It provides efficient document indexing, caching, and retrieval for medical and insurance-related queries.

Usage

Nothing in this package needs to be run directly. The `retriever` package is imported by pipeline (main folder), and pipeline is imported by main. This package is also imported into app_retrieved_cache.

data_schemas

Purpose: Data structures for document representation

Class: `Doc`

A dataclass representing a searchable document in the retrieval system.

Fields

- `id` (str): Unique document identifier
- `text` (str): Main document content for searching and display
- `title` (str | None): Optional document title or category
- `meta` (dict | None): Optional metadata dictionary containing additional fields from source

index_bm25

Purpose: BM25 (Best Matching 25) sparse retrieval index

Class: `BM25Index`

Implements the BM25Okapi algorithm for keyword-based document retrieval.

Parameters:

- `docs`: List of Doc objects to index

Process:

1. Tokenizes all document texts using the `tokenize()` utility
2. Builds BM25Okapi index from tokenized corpus

Attributes

- **`docs`**: Original document list
- **`corpus_tokens`**: List of tokenized documents
- **`bm25`**: BM25Okapi index object

Methods

```
search(query: str, k: int = 50) -> list[tuple[Doc, float]]
```

Searches the BM25 index for relevant documents.

Parameters:

- `query`: Search query string
- `k`: Number of top results to return (default: 50)

Returns:

- List of tuples: `[(Doc, score), ...]` sorted by relevance (highest first)

Score Interpretation:

- Higher scores indicate better keyword match
- Scores are relative (not normalized 0-1)
- Based on term frequency and inverse document frequency

index_dense

Purpose: Dense vector retrieval using sentence embeddings with FAISS acceleration

Class: `DenseIndex`

Implements semantic search using sentence transformers and vector similarity.

Parameters:

- `docs`: List of documents to index
- `model_name`: HuggingFace model identifier (default: all-MiniLM-L6-v2)
- `batch_size`: Batch size for encoding (default: 64)
- `embedding_model`: Pre-initialized model (optional, overrides model_name)
- `cache_dir`: Directory for caching embeddings (default: .cache/embeddings)

Key Features

Intelligent Caching System:

- **Full Cache**: Saves complete embedding matrix when indexing completes
- **Partial Cache**: Saves progress incrementally during indexing
- **Cache Key**: MD5 hash of document IDs + model name ensures validity
- **Resume Support**: Can resume interrupted indexing from partial cache

Background Indexing:

- Embeddings generated in background thread
- Search available on partial data during indexing
- Non-blocking initialization for faster startup

FAISS Acceleration:

- Uses FAISS IndexFlatIP for fast similarity search (if installed)
- Falls back to NumPy operations if FAISS unavailable
- Inner product search (equivalent to cosine similarity with normalized vectors)

Methods

```
search(query: str, k: int = 50) -> list[tuple[Doc, float]]
```

Searches for semantically similar documents.

Parameters:

- `query`: Search query string
- `k`: Number of results to return (default: 50)

Returns:

- List of tuples: `[(Doc, similarity_score), ...]` sorted by similarity

Score Interpretation:

- Range: [-1, 1] (cosine similarity)
- Higher = more semantically similar
- 1.0 = identical meaning, 0.0 = orthogonal, -1.0 = opposite

Thread Safety:

- Safe to call during background indexing
- Searches available documents even if indexing incomplete
- Returns empty list if called before any documents indexed

`get_progress() -> tuple[int, int]`

Returns indexing progress.

Returns:

- Tuple of `(current_count, total_count)`
- `current_count`: Number of documents currently indexed
- `total_count`: Total number of documents to index

Private Methods for DenseIndex()

`_generate_embeddings()`

Generator that yields embedding batches from cache or computation.

Caching Strategy:

1. Try loading full cache (fastest)
2. Load partial cache + compute remaining
3. Compute all embeddings from scratch

`_ingest_embeddings()`

Background thread that ingests embeddings into the index.

Process:

1. Iterates through embedding batches from generator
2. Adds each batch to FAISS index (if available)
3. Updates ready_count and emb_batches
4. Saves full cache when complete
5. Cleans up partial cache file

`_compute_cache_key(docs, model_name)`

Computes MD5 hash for cache identification.

`_chunks(lst, n)`

Generator that yields list in chunks of size n.

Performance Considerations

Memory:

- Embeddings stored as float32 (4 bytes per dimension)
- Example: 100K docs × 384 dims = ~150MB
- FAISS index adds minimal overhead

ingest

Purpose: Document ingestion from JSONL files

Function: `load_jsonl`

Loads documents from JSON Lines format into Doc objects.

Parameters:

- `path`: File path to JSONL file
- `text_fields`: Tuple of field names to combine into document text (default: ("question", "answer"))

Returns:

- List of `Doc` objects

rrf

Purpose: Reciprocal Rank Fusion for combining multiple rankings

Function: `rrf`

Fuses multiple ranked lists using Reciprocal Rank Fusion algorithm.

Parameters:

- `rank_lists`: List of ranked result lists (each containing Doc-score tuples)
- `k`: Number of final results to return(default: 10)
- `K: RRF constant parameter (default 60)`

Returns:

- List of `Doc` objects

search

Purpose: Main retrieval interface combining BM25, dense search, and optional reranking

Class: `Retriever`

Unified retrieval system implementing hybrid search with result fusion.

Parameters:

- `corpora_config`: Dictionary mapping corpus names to configuration
- `use_reranker`: Whether to use cross-encoder reranking (default: False)
- `embedding_model`: Pre-initialized embedding model (optional)

Attributes

- **`corpora`**: Dictionary of corpus_name → list of Docs
- **`bm25`**: BM25Index instance
- **`dense`**: DenseIndex instance
- **`reranker`**: CrossEncoderReranker instance or None

Methods

`retrieve(query: str, k: int = 10, for_ui: bool = True) -> list`

Main retrieval method implementing hybrid search pipeline.

Parameters:

- `query`: Search query string
- `k`: Number of results to return (default: 10)
- `for_ui`: Return UI-friendly format if True, raw Doc objects if False

Returns:

If `for_ui=True` (default):

[{ "id": "icliniq:123", "title": "Ankle Sprain Treatment", "snippet": "Patient presents with acute ankle pain after...", "score": 0.85, "meta": {...} # *Original JSONL data* }, ...]

If `for_ui=False`:
[(Doc(...), score), ...]

Retrieval Pipeline:

1. **BM25 Search:**
    - Retrieves top 100 results based on keyword matching

    - Fast, captures exact term matches
2. **Dense Search:**
    - Retrieves top 100 results based on semantic similarity
    - Captures meaning and context
3. **Reciprocal Rank Fusion:**
    - Combines BM25 and dense results using RRF algorithm
    - Keeps top 20 (or k, whichever is larger) fused results
    - Documents appearing in both lists ranked higher
4. **Optional Reranking (if enabled):**
    - Cross-encoder reranking of fused results
    - More accurate but slower
    - Returns top-k after reranking
5. **Format Conversion:**
    - Truncates snippets to 300 characters
    - Extracts relevant fields for UI display

utils - retriever

Purpose: Text tokenization utilities for BM25

Function: `tokenize`

Simple tokenizer for BM25 index construction.

Parameters:

- `s`: Input string to tokenize

Returns:

- List of lowercase tokens

Algorithm:

1. **Case Normalization:** Converts to lowercase
2. **Splitting:** Splits on any non-word character (regex `\W+`)
3. **Filtering:** Removes empty strings
4. **Unicode Support:** Handles Unicode characters correctly

team

Team Package Documentation

Package Overview

The `team` package provides interfaces and utilities for candidate document retrieval with detailed scoring information. It extends the base retrieval system by exposing individual component scores (BM25, dense, RRF) for each candidate, enabling downstream ranking and analysis.

Usage

Nothing in this folder needs to be run directly.
candidates imports interfaces
Pipeline imports candidates, and pipeline is imported into main.

candidates

Purpose: Data structures for candidate documents with scoring metadata

Class: `Candidate`

A dataclass representing a retrieved document with detailed scoring information from multiple retrieval components.

Fields

- **`id`** (str): Unique document identifier
- **`title`** (str): Document title or category
- **`text`** (str): Full document text content
- **`meta`** (Dict[str, Any]): Metadata dictionary from original source
- **`bm25`** (float): BM25 (keyword) retrieval score
- **`dense`** (float): Dense (semantic) retrieval score
- **`rrf`** (float): Reciprocal Rank Fusion combined score

Purpose

Unlike the base `Doc` class in the retriever package, `Candidate` includes:

- Component-level scores for interpretability
- Fusion score for ranking
- Complete metadata for downstream processing

Score Interpretation

BM25 Score:

- Range: Unbounded (typically 0-30 for relevant docs)
- Higher = better keyword match
- Based on term frequency and document rarity

Dense Score:

- Range: -1.0 to 1.0 (cosine similarity)
- Higher = more semantically similar
- Normalized embeddings

RRF Score:

- Range: 0.0 to ~0.033 (for K=60)
- Computed as: Σ[1/(K + rank + 1)]
- Higher = better combined ranking

- Balances both retrieval methods

interfaces

Purpose: High-level candidate retrieval with transparent scoring

Function: `get_candidates`

Retrieves candidate documents with detailed component scores for analysis and reranking.

Parameters:

- `query`: Search query string
- `retriever`: Initialized Retriever instance
- `k_retrieve`: Number of candidates to return (default: 50)

Returns:

- List of `Candidate` objects sorted by RRF score (descending)

Retrieval Process:

1. Parallel Component Retrieval:
    - Retrieves at least 100 candidates from each method
    - Ensures sufficient overlap for effective fusion
2. Score Mapping:
    - Creates lookup dictionaries for component scores
    - Used to populate Candidate objects
3. Reciprocal Rank Fusion:
    - Combines rankings using RRF algorithm
    - Retrieves at least 50 fused results
4. RRF Score Computation:
    - Recalculates RRF score from rank positions
    - K constant = 60 (standard RRF parameter)
    - Documents not in a list contribute 0 to that component
5. Candidate Construction:

```
Candidate(
id=doc.id,
title=doc.title or "",
text=doc.text,
meta=doc.meta or {},
bm25=bm_map.get(doc.id, 0.0),
dense=de_map.get(doc.id, 0.0), rrf=rrf_score
)
```

    - Combines document content with all scores
    - Uses 0.0 default for missing scores

6. Final Sorting:
    - Sorts by RRF score (highest first)
    - Baseline ranking for downstream use

Helper Functions

`_default_corpora_config() -> Dict[str, dict]`

Returns default configuration for medical corpora.

Configured Corpora:

Corpus	Path	Text Fields
medical_qa	data/corpora/medical_qa.jsonl	question, answer, title
miriad	data/corpora/miriad_text.jsonl	question, answer, title
pubmed	data/corpora/pubmed.jsonl	contents, title
unidoc	data/corpora/unidoc_qa.jsonl	question, answer, title

`_available(cfg: Dict[str, dict]) -> Dict[str, dict]`

Filters corpus configuration to only include existing files.

Parameters:

- `cfg`: Full corpus configuration dictionary

Returns:

- Filtered configuration with only available corpora

CLI

CLI Package Documentation

Package Overview

The `cli` package provides command-line interfaces for testing and using the healthcare classification system. It includes tools for both primary (medical/insurance) classification and secondary (medical reason) classification with interactive, single-query, and batch processing modes.

When to Use Each

Use `healthcare_classifier_cli.py` when:

- Testing complete end-to-end workflow
- Evaluating full classification pipeline
- Generating routing recommendations
- Processing mixed medical/insurance queries
- Production testing

Use `reason_classifier_cli.py` when:

- Testing reason classifier independently
- Evaluating reason classification accuracy
- Working only with medical queries
- Debugging reason classification
- Faster iteration during development

The CLI package provides:

1. **Complete Classification Pipeline** (`healthcare_classifier_cli.py`):
    - Two-stage classification (primary + reason)
    - Department routing recommendations
    - Production-ready interface
2. **Focused Reason Classification** (`reason_classifier_cli.py`):
    - Independent reason testing
    - Faster iteration
    - Development-friendly
3. **Multiple Interaction Modes:**
    - Interactive for manual testing
    - Single query for quick checks
    - Batch processing for evaluation

4. **Comprehensive Output:**
    - Console display for immediate feedback
    - JSON export for analysis
    - Summary statistics

healthcare_classifier_cli

Purpose: Complete end-to-end healthcare classification CLI with two-stage classification pipeline

Overview

This CLI implements the full healthcare classification workflow:

1. **Primary Classification:** Medical vs Insurance
2. **Secondary Classification:** Medical reason categorization (if medical)
3. **Routing Decision:** Final department assignment

Key Features

- Interactive mode for real-time testing
- Single query classification
- Batch processing from files
- Complete pipeline execution
- Detailed probability outputs
- Department routing recommendations

Function: `classify_healthcare_query`

Main classification function implementing the complete two-stage pipeline.

Parameters:

- `query`: Healthcare query string to classify

Returns:

- Dictionary containing classification results
- Returns `None` if classification fails.

```
{

    'primary_classification': 'medical' or 'insurance',

    'primary_confidence': 0.95,

    'reason_classification': 'PAIN_CONDITIONS',  # None for insurance
```

```
        'reason_confidence': 0.87, # None for insurance
        'routing': 'Pain Management Department'
}
```

Function: `classify_batch_queries`

Processes multiple queries through the complete pipeline.

Parameters:

- `queries_file`: Path to file containing queries (text or JSON)
- `output_file`: Optional path for JSON output

Returns:

- `True` if successful
- `False` otherwise.

Function: `interactive_mode`

Interactive command-line interface for real-time classification testing.

Features:

- Real-time classification
- Immediate feedback
- Example queries provided
- Graceful exit (type 'quit' or Ctrl-C)
- Error recovery

Function: `main`

Command-line argument parser and dispatcher.

Returns:

- `0` on success

- 1 on error

Command-Line Arguments:

Argument	Type	Description
`query`	positional	Single query to classify (optional)
`--batch`	string	File containing queries to process
`--output`	string	Output file for batch results
`--interactive`	flag	Start interactive mode (recommended)

Usage Examples:

```
Interactive Mode (Recommended):
# Activate virtual environment first
source .venv/bin/activate
# Start interactive mode
python cli/healthcare_classifier_cli.py --interactive
```

Single Query:

```
python cli/healthcare_classifier_cli.py "I have heel pain"
```

Batch Processing:

```
# From text file
python cli/healthcare_classifier_cli.py --batch queries.txt
# With output file
python cli/healthcare_classifier_cli.py --batch queries.txt --output
results.json
# From JSON file
python cli/healthcare_classifier_cli.py --batch queries.json --output
results.json
```

Help:

```
python cli/healthcare_classifier_cli.py --help
```

reason_classifier_cli

Purpose: Focused CLI for medical reason classification testing

Overview

This CLI provides a simpler interface focused specifically on reason classification without the primary medical/insurance classification stage. Useful for testing and evaluating the reason classifier independently.

Function: `classify_single_query`

Classifies a single query using only the reason classifier.

Parameters:

- `query`: Medical query string

Returns:

- `True` if successful
- `False` otherwise (on error)

Function: `classify_batch_queries`

Processes multiple queries for reason classification.

Parameters:

- `queries_file`: Path to file with queries (text or JSON)
- `output_file`: Optional output file path

Returns:

- `True` if successful, `False` otherwise

Input Formats:

Same as `healthcare_classifier_cli.py`:

- Text file (one query per line)
- JSON array
- JSON object with "queries" key

Output Format:


```
{
    "queries": [
        "I have heel pain when I walk",
        "My toenail is ingrown",
        "I need routine foot care"
    ],
    "predictions": [
    {
        "category": "PAIN_CONDITIONS",
        "confidence": 0.8734,
        "probabilities": {
            "PAIN_CONDITIONS": 0.8734,
            "INJURIES": 0.0892,
            ...
        }
    },
    {
        "category": "SKIN_CONDITIONS",
        "confidence": 0.9156,
        "probabilities": {...}
    },
    {
        "category": "ROUTINE_CARE",
```

```
            "confidence": 0.8923,
            "probabilities": {...}
        }
    ],
    "summary": {
        "total_queries": 3,
        "categories": {
            "PAIN_CONDITIONS": 1,
            "SKIN_CONDITIONS": 1,
            "ROUTINE_CARE": 1
        }
    }
}
```

Function: `interactive_mode`

Interactive testing interface for reason classification.

Function: `main`

Argument parser and dispatcher for reason classifier CLI.

Command-Line Arguments:

Argument	Type	Description
`query`	positional	Single query to classify (optional)
`--batch`	string	File containing queries to process

`--output`	string	Output file for batch results
`--interactive`	flag	Start interactive mode

Usage Examples:

```
python cli/reason_classifier_cli.py --interactive
```

Single Query:

```
python cli/reason_classifier_cli.py "I have heel pain"
```

Batch Processing:

```
python cli/reason_classifier_cli.py --batch reason_queries.txt --output
results.json
```

Help:

```
python cli/reason_classifier_cli.py --help
```

main

Purpose: Interactive command-line interface for the health query pipeline

Constants

- **`EXIT_COMMANDS`**: `["exit", "quit"]` - Commands to exit the program
- **`PROMPT`**: `"\nQuery> "` - User input prompt string

Function: `main`

Main interactive loop for processing queries.

Parameters:

- `pipeline`: Initialized HealthQueryPipeline instance
- `k`: Number of results to return per query

Interactive Loop:

1. **Display Instructions:**

```
print(f"(Ctrl-D or 'quit' to exit)")
```

2. **Read Query:**
    - Prompts user for input
    - Strips whitespace
    - Exits on empty or exit commands

```
query = input(PROMPT).strip()
```

3. **Show Index Progress:**
    - Displays indexing progress if incomplete
    - Informs user that system is still loading

```
curr, total = pipeline.get_index_progress() if pct < 100: print(f"[Index: {pct}% loaded]")
```

4. **Execute Pipeline:**

```
result = pipeline.predict(query, k=k)
```

    - Runs classification and retrieval
    - Returns structured result
5. **Display Classification:**
    - Shows predicted category
    - Displays confidence scores for all categories

- Formatted as percentages

```
print(f"\nTriaging query as {prediction}")
print(f"\nConfidence:")
for cat, prob in classification["probabilities"].items():
percent = prob * 100 print(f" {cat}: {percent:3.2f}%")
```

6. **Display Retrieval Results:**
   - Only for medical queries
   - Pretty-prints each document as JSON
   - Shows all metadata and scores

```
if "medical" == prediction:
hits = result["retrieval"]
print(f"Found {len(hits)} matching medical documents\n")
for i, hit in enumerate(hits, 1):
print(json.dumps(hit, indent=2, ensure_ascii=False))
```

7. **Handle Non-Medical:**
   - Placeholder for insurance query handling
   - Could be extended for insurance-specific logic

```
else:
print(f"TODO: handle queries of type {prediction}")
```

Command-Line Interface (e.g. if __name__ == "__main__":)

Argument Parser:
Arguments:

- `--k`: Number of results (default: 10)
- `--rerank`: Enable cross-encoder reranking (flag)

Execution:

```
pipeline = HealthQueryPipeline(use_reranker=args.rerank)
pipeline.initialize()
main(pipeline, k=args.k)
```

Usage Examples

Basic Usage:

```
python main.py
# Query> ankle pain after injury
# Triaging query as medical
```

```
# Confidence: # medical: 95.23% # insurance: 4.77%
# Found 10 matching medical documents
# [JSON output...]
```

With Custom k:

```
python main.py --k 5 # Returns top 5 results per query
```

With Reranking:

```
python main.py --rerank # Enables cross-encoder reranking (slower, more accurate)
```

Sample Session:

```
$ python main.py --k 3
(Ctrl-D or 'quit' to exit)

Query> headache with blurred vision

Triaging query as medical

Confidence:
  medical: 98.45%
  insurance: 1.55%

Found 3 matching medical documents

{
  "id": "medical:1234",
  "title": "Headache with Vision Changes",
  "text": "Headache accompanied by vision problems...",
  "bm25": 15.234,
  "dense": 0.876,
  "rrf": 0.0289
 }
 ...

Query> insurance claim status

Triaging query as insurance

Confidence:
  medical: 8.23%
  insurance: 91.77%
```

```
TODO: handle queries of type insurance

Query> quit
Bye!
```

app_retrieval_cached

Purpose: Gradio web interface with aggressive disk caching for instant startup

Overview

This file implements a web-based UI for the medical retrieval system with a sophisticated caching strategy that reduces startup time from 5-8 minutes to ~30 seconds on subsequent launches.

Caching Strategy

Cache Directory:

```
CACHE_DIR = Path("cache")
CACHE_DIR.mkdir(exist_ok=True)
```

Cache Key Generation:

```
def _get_cache_key(corpora_config: Dict[str, dict]) -> str:
    config_str = json.dumps(corpora_config, sort_keys=True)
    return hashlib.md5(config_str.encode()).hexdigest()
```

- Generates MD5 hash from corpus configuration
- Ensures cache invalidation when configuration changes
- Deterministic across runs'

Cached Components:

1. **Documents Cache:** `docs_{cache_key}.pkl`
    - Loaded/parsed documents from JSONL files
    - Saves file I/O and parsing time
2. **BM25 Index Cache:** `bm25_{cache_key}.pkl`
    - Tokenized corpus and BM25 structure
    - Saves tokenization time (~1-2 minutes)
3. **Dense Index Cache:** `dense_{cache_key}.pkl`
    - Pre-computed embeddings (NOT recommended, see note below)
    - Saves embedding computation time (~5-8 minutes)

Class: `CachedRetriever`

Custom retriever implementation with aggressive disk caching.

Process:

1. Generates cache key from configuration
2. Sets cache file paths for all components
3. Loads or builds documents
4. Loads or builds BM25 index
5. Loads or builds dense index

Methods

`_load_or_build_docs() -> List`

Loads documents from cache or rebuilds from corpus files.

Cache Hit:

- Instant loading (~100ms)
- Prints success message

Cache Miss:

- Loads from JSONL files
- Combines all corpora
- Saves to cache

Performance:

- Cache hit: ~100ms
- Cache miss: ~5-15 seconds

`_load_or_build_bm25() -> BM25Index`

Loads BM25 index from cache or rebuilds.

Cache Hit:

- Loads pickled BM25Index object
- Instant restoration

Cache Miss:

- Tokenizes all documents
- Builds BM25 structure
- Saves to cache

Performance:

- Cache hit: ~100ms
- Cache miss: ~30-60 seconds

`_load_or_build_dense() -> DenseIndex`

Loads dense index from cache or rebuilds.

Function: `get_candidates_cached`

Retrieval function using the cached retriever.

Identical to `team.candidates.get_candidates` but:

- Uses module-level `retriever` instance
- Doesn't require passing retriever as parameter
- Tied to specific cached retriever

Process:

1. BM25 search (k=100)
2. Dense search (k=100)
3. Score mapping
4. RRF fusion
5. Candidate construction
6. Sort by RRF

Function: `retrieve_documents`

Main retrieval function for Gradio interface.

Parameters:

- `query`: Search query string
- `num_results`: Number of results to display (default: 5)

Returns:

- HTML string for Gradio display

Output Cases:

1. **Empty Query:**
    - Returns instruction card
    - Blue background with usage examples
2. **No Results:**
    - Returns warning card
    - Yellow background with suggestions
3. **Success:**
    - Returns formatted results with:
        - Header with result count
        - Document cards with gradient headers
        - Question/Answer formatting (if available)
        - Score badges (BM25, Dense, RRF)
        - Truncated snippets (500 chars)
4. **Error:**
    - Returns error card
    - Red background with error message

Gradio Interface

Main Interface Structure:

Components:

1. **Input Section:**
    - Text box for query (2 lines)
    - Slider for number of results (1-10)
    - Primary button to submit
2. **Output Section:**
    - HTML component for formatted results
    - Dynamically updated on submission
3. **Examples:**
    - Pre-filled queries for quick testing
    - Click to populate input field
4. **Documentation:**
    - Markdown sections explaining:
        - System capabilities
        - Model components
        - Technical details
        - Caching benefits

Launch Configuration
Parameters:

- `server_name`: Localhost only (security)
- `server_port`: Fixed port for consistency
- `share=True`: Generates public Gradio link

Access Methods:

- Local: `http://127.0.0.1:7863`
- Public: Generated share link (temporary, 72 hours)

Usage

```
python app_retrieval_cached.py
```

Cache Invalidation:

- Automatic when corpus configuration changes
- Manual: Delete `cache/` directory
- Each configuration gets unique cache files

config (main folder)

Purpose: Centralized configuration management using Pydantic settings

Class: `Settings`

Configuration manager using Pydantic BaseSettings for type-safe configuration with environment variable support.

Fields

Model Configuration:

- `MODEL_NAME` (str): Sentence transformer model identifier
  - Default: `"sentence-transformers/embeddinggemma-300m-medical"`
  - 768-dimensional medical domain embeddings
- `CLASSIFIER_NAME` (str): HuggingFace classifier model ID
  - Default: `"davidgray/health-query-triage"`
  - Trained medical/insurance classifier
- `CATEGORIES` (List[str]): Classification labels
  - Default: `["medical", "insurance"]`
  - Used for label indexing and display

File System Paths:

- `CHECKPOINT_PATH` (str): Local model checkpoint directory
  - Default: `"classifier/checkpoints"`
  - For saving/loading training checkpoints
- `CACHE_DIR` (str): Embedding cache directory
  - Default: `".cache/embeddings"`
  - Stores computed embeddings for fast reloading

Hardware Configuration:

- `DEVICE` (str): Compute device for model inference
  - Auto-detected: CUDA → MPS → CPU
  - Override via environment variable if needed

Corpora Configuration:

- `CORPORA_CONFIG` (Dict[str, dict]): Available document corpora
  - Maps corpus name to path and text fields
  - Configures which fields to extract for indexing

Inner Class: `Config`

Pydantic configuration class.

- **`env_file`**: `".env"`
    - Loads settings from `.env` file if present
    - Environment variables override defaults

pipeline

Purpose: Unified pipeline orchestrating classification and retrieval

Class: `HealthQueryPipeline`

Complete end-to-end pipeline for processing health-related queries.

Parameters:

- `use_reranker`: Enable cross-encoder reranking for better accuracy (default: False)

Initial State:

- All models set to `None`
- `is_initialized` set to `False`
- Lazy initialization on first use

Attributes

- **`use_reranker`** (bool): Whether to use cross-encoder reranking
- **`embedding_model`** (SentenceTransformer | None): Embedding model for classification and retrieval
- **`classifier`** (ClassifierHead | None): Medical/insurance classifier
- **`retriever`** (Retriever | None): Hybrid retrieval system
- **`is_initialized`** (bool): Initialization status flag

Methods

`initialize()`

Loads models and initializes the retrieval system.

Process:

1. **Check Initialization:**
    - Returns immediately if already initialized
    - Prevents duplicate loading
2. **Load Classification Models:**
    - Loads SentenceTransformer embedding model
    - Loads trained ClassifierHead from HuggingFace or local
3. **Initialize Retriever:**
    - Filters to available corpus files
    - Shares embedding model between classification and retrieval
    - Starts background indexing

4. **Mark Complete:**
    - Sets `is_initialized = True`
    - Prints status messages

Error Handling:

- Raises `RuntimeError` if no corpus files found
- Model loading errors propagate to caller

Timing:

- First call: 30 seconds to 8 minutes (depending on cache)
- Subsequent calls: Instant (already initialized)

`predict(query: str, k: int = 10) -> Dict[str, Any]`

Executes the complete pipeline on a query.

Parameters:

- `query`: User's health query string
- `k`: Number of results to return for medical queries (default: 10)

Return Format:

```
{
        "query": "original query text",
        "classification": {
                "prediction": "medical" or "insurance",
                "probabilities": {
                        "medical": 0.85,
                        "insurance": 0.15
                }
        },
        "retrieval": [
                {
                        "id": "doc_id",
                        "title": "document title",
                        "text": "document text",
                        "meta": {...},
                        "bm25": 12.5,
                        "dense": 0.87,
```

```
                "rrf": 0.025
            },
            # ... more results
        ]
}
```

Pipeline Stages:

1. **Auto-Initialize:**
    - Ensures models loaded before prediction
    - Transparent lazy initialization
2. **Classification:**
    - Determines if query is medical or insurance
    - Returns probabilities for both categories
3. **Build Base Result:**
    - Creates response structure
    - Retrieval initially empty
4. **Conditional Retrieval:**
    - Only retrieves documents for medical queries
    - Converts Candidate objects to dictionaries
    - Insurance queries return empty retrieval list

Design Rationale:

- **Unified Interface:** Single method for complete processing
- **Lazy Initialization:** Fast import, slow first use
- **Conditional Retrieval:** Don't search for insurance queries
- **Structured Output:** Consistent JSON-serializable format

`get_index_progress() -> tuple[int, int]`

Returns indexing progress from the retrieval system.

Returns:

- Tuple of `(current_indexed, total_documents)`
- Returns `(0, 0)` if retriever not initialized

Integration Benefits

Single Import:

```
from pipeline import HealthQueryPipeline
```

Shared Models:

- Embedding model used for both classification and retrieval
- Efficient memory usage
- Consistent embeddings across components