PDF Text Extraction Algorithm:

The PDFTextExtractor class is a centralized module designed to extract text from PDF files with detailed positional information, optimized for both single-file and batch processing scenarios. This implementation supports various input types (file path, byte stream, in-memory file-like objects) and applies efficient memory and synchronization strategies suitable for GDPR-compliant environments.

Key Features:

- Supports file path, bytes, or stream input.
- Page-wise and batch-wise text extraction with bounding boxes.
- Thread-safe via timeout locks (TimeoutLock).
- Metadata sanitization and GDPR-aware error handling.
- Parallel batch processing with ParallelProcessingCore.

Core Algorithm Logic:

1. Initialization and Input Handling

The extractor accepts flexible inputs:

```
# Check if the pdf_input is a string (file path).
if isinstance(pdf_input, str):
  # Open the PDF document using PyMuPDF with the file path.
  self.pdf_document = pymupdf.open(pdf_input)
  # Set file_labeling_path attribute with the file path.
  self.file_path = pdf_input


# Check if the pdf_input is already a PyMuPDF Document instance.
elif isinstance(pdf_input, pymupdf.Document):
  # Use the provided PyMuPDF document.
  self.pdf_document = pdf_input
  # Mark the file_labeling_path as a memory document.
  self.file_path = "memory_document"


# Check if the pdf_input is of bytes type.
elif isinstance(pdf_input, bytes):
  # Wrap the bytes in a BytesIO stream.
  buffer = io.BytesIO(pdf_input)
  # Open the PDF document using the stream.
  self.pdf_document = pymupdf.open(stream=buffer, filetype="pdf")
  # Set the file_labeling_path as memory_buffer.
  self.file_path = "memory_buffer"
```

2. Main Extraction Logic

Depending on the PDF size, it processes pages sequentially or in batches:

```
# Determine if page batch processing should be used (for documents with more than 10 pages).
use_paged_processing = total_pages > 10

if use_paged_processing:
  # If using batch processing, extract pages in batches.
  self._extract_pages_in_batches(extracted_data, empty_pages)


else:
  # For smaller documents, process each page sequentially.
  for page_num in range(total_pages):
    # Process a single page.
    self._process_page(page_num, extracted_data, empty_pages)
```

3. Per-Page Word Extraction with Bounding Boxes

Uses PyMuPDF's get_text("words") to obtain coordinates:

```
# Retrieve word information from the page as a list of tuples.
words = page.get_text("words")


# Iterate over each word tuple.
for word in words:


  # Unpack coordinates and text from the tuple.
  x0, y0, x1, y1, text, *_ = word


  # Check if the text is not empty after stripping whitespace.
  if text.strip():


    # Append the word and its coordinates as a dictionary.
    page_words.append({
      "text": text.strip(),
      "x0": x0,
      "y0": y0,
      "x1": x1,
      "y1": y1
    })
```

Response Structure:

The response of the extraction method is a structured dictionary:

```
{
 "pages": [
  {
   "page": 1,
   "words": [
    {"text": "Hello", "x0": 100.2, "y0": 150.0, "x1": 145.6, "y1": 160.0},
    ...
   ]
  },
  ...
 ],
 "total_document_pages": 5,
 "empty_pages": [3],
 "content_pages": 4,
 "metadata": {
  "author": "...",
  "title": "...",
  ...
 }
}
```

Mapping Logic:

- Each **page** maps to a list of **words**, with bounding box coordinates (x0, y0, x1, y1).
- **Empty pages** are tracked separately to optimize downstream processing.
- The **metadata** is sanitized before inclusion.
- Logs and record-keeping support GDPR auditing.

Batch Mode: Parallel PDF Processing:

The method extract_batch_text() allows asynchronous batch extraction:

```
# Use ParallelProcessingCore to process all PDFs in parallel.
results = await ParallelProcessingCore.process_in_parallel(
    pdf_files,
    lambda pdf: process_pdf(pdf),
    max_workers=max_workers
)
```

Each result is a tuple (index, result_dict) enabling parallel-safe, ordered responses.

📜 Summary

This algorithm enables efficient, scalable, and secure PDF text extraction. The inclusion of bounding box mapping allows precise word-level tracking, making it ideal for systems requiring redaction, annotation, or semantic search.

The PDFSearcher class is designed to locate specific terms or phrases within text extracted from PDF files. It supports two main modes of operation:

1. Search by Terms (Keyword Search)

There are **two sub-modes** for searching based on keywords:

a. AI Search Mode (Gemini API)

This mode sends the entire page text along with the search terms to the Gemini LLM for intelligent, context-aware matching.

How It Works:

- The page text is reconstructed using word-by-word mapping.
- A prompt is created using a header, the entity list, user query, and full page text.
- The prompt is sent to the Gemini API.
- The response is parsed, and each returned token is remapped to its position and bounding box in the original text.

```
mapping, page_text = self.build_page_text_and_mapping(words)

prompt = self._build_ai_prompt([combined_query], page_text)

response = await self.gemini_helper.send_request(prompt, ...)

parsed_response = self.gemini_helper.parse_response(response)

matches = self._split_and_remap_entity(entity, mapping, case_sensitive)
```

b. Fallback Word-Based Matching

In this mode, every word is matched directly against the search terms without using AI.

Logic:

- Iterates through each word in the page.
- Matches are checked using case sensitivity (optional).
- Matching words' bounding boxes are collected.

```
# Check whether the cleaned word matches any term in the search set.

if self._word_matches(word_text, search_set, case_sensitive, ai_search=False):

  # Retrieve the bounding box from the word, or construct it from coordinates.

  bbox = word.get("bbox") or {

    "x0": word.get("x0"),

    "y0": word.get("y0"),

    "x1": word.get("x1"),
    "y1": word.get("y1")
  }
  # Append a dictionary of the matching word's bounding box.
  page_matches.append({
    "bbox": bbox,
  })
```

Response Format:

```
{
 "pages": [
  {
   "page": 1,
   "matches": [
    { "bbox": { "x0": ..., "y0": ..., "x1": ..., "y1": ... } }
   ]
  },
  ...
 ],
 "match_count": 5
}
```

2. Search by Bounding Box (BBox Phrase Re-Match)

This mode is used when the system needs to re-identify a phrase based on its bounding box. It is especially useful for **phrase propagation**, redaction, or linked annotations.

a. Step 1: Identify the Phrase Behind a BBox

Finds a candidate phrase by:

- Checking which words fall inside the bounding box center.
- Grouping them into possible phrases.
- Reconstructing each candidate phrase.
- Comparing the computed phrase bbox with the original using tolerance.

```
    # Identify indices of words whose center falls inside the target_bbox
    candidate_indices = [
      idx for idx, m in enumerate(mapping)
      if self._word_center_in_bbox(m["bbox"], target_bbox)
    ]

    if not candidate_indices:
      # Skip this page if no candidates were found
      continue

    # Group the candidate indices into consecutive word groups (phrases)
    groups = self._group_consecutive_indices(candidate_indices)
```

b. Step 2: Re-Search That Phrase Across All Pages

Once a phrase is confirmed, it is searched again across all pages.

- If it's a **single word**, direct text-to-bbox mapping is used.
- If it's a **multi-word phrase**, offsets in full page text are computed and remapped to bounding boxes.

```
full_text, text_mapping = TextUtils.reconstruct_text_and_mapping(page.get("words", []))
matches = TextUtils.recompute_offsets(full_text, candidate_phrase)
bboxes = TextUtils.map_offsets_to_bboxes(full_text, text_mapping, (s, e))
```

Final Output:

```
{
 "pages": [
  { "page": 1, "matches": [ { "bbox": ... } ] },
  ...
 ],
 "target_phrase": "your exact phrase",
 "word_count": 3
}
```

Summary

Mode	**Method Used**	**Use Case**
AI Search	Gemini API with remapping	Context-aware entity extraction
Fallback Search	Word-by-word comparison	Fast, simple term match
BBox Phrase Search	BBox-to-text rematch + propagation	Phrase tracking/redaction across pages

This dual-mode system enables intelligent and precise search operations within PDFs, supporting both classic keyword queries and advanced bounding box-driven workflows.

PDF Redaction Algorithm

The PDFRedactionService class is responsible for securely redacting sensitive content from PDF documents. It supports both **file-based** and **in-memory redaction**, removing both **text and images** based on provided bounding boxes.

The redaction system is **GDPR-compliant** and optimized for **thread safety**, **memory efficiency**, and **batch processing** of large documents.

Redaction Workflow Overview

The primary entry point is:

```
apply_redactions(redaction_mapping: Dict[str, Any], output_path: str, remove_images: bool = False)
```

This function:

1. **Acquires a timeout lock** to ensure thread-safe access.
2. **Draws redaction boxes** on sensitive areas:

   ```
   self._draw_redaction_boxes(redaction_mapping, remove_images)
   ```

3. **Sanitizes the document**:

   ```
   self._sanitize_document()
   ```

4. **Saves the redacted PDF** to file or memory.

_draw_redaction_boxes() Algorithm

This method is the **core of the redaction logic**. It takes the redaction mapping (list of sensitive items per page) and applies redaction rectangles across affected pages.

```
def _draw_redaction_boxes(self, redaction_mapping: Dict[str, Any], remove_images: bool) -> None:
  """
  Process the redaction mapping and apply redaction boxes to the specified pages.

  For large documents, redaction is applied in batches to optimize memory usage.

  Args:
    redaction_mapping: Dictionary with redaction information.
    remove_images: Flag to enable image redaction on each page.
  """
```

```
# Get the total number of pages in the PDF document.
total_pages = len(self.doc)


# Extract a sorted list of unique page numbers from the mapping.
page_numbers = {page_info.get("page") for page_info in redaction_mapping.get("pages", [])
        if page_info.get("page") is not None}
page_numbers = sorted(page_numbers)

try:


  # Check if the number of pages exceeds 10 for batching.
  if len(page_numbers) > 10:


    # Iterate through page numbers in batches.
    for batch_start in range(0, len(page_numbers), self.page_batch_size):


      # Determine the end index for the current batch.
      batch_end = min(batch_start + self.page_batch_size, len(page_numbers))


      # Get the list of page numbers in the current batch.
      batch_page_numbers = page_numbers[batch_start:batch_end]


      # Log the batch processing of redaction.
      log_info(f"[OK] Processing redaction batch for pages {batch_page_numbers}")


      # Process the current batch of pages.
      self._process_redaction_pages_batch(redaction_mapping, batch_page_numbers,
total_pages,  remove_images)


      # Import and invoke garbage collection to free memory.
      import gc
      gc.collect()


  else:
    # If few pages, process them all at once.
    self._process_all_redaction_pages(redaction_mapping, total_pages, remove_images)
```

What Happens per Page?

Each page is passed to _process_redaction_page(), which handles two cases:

1. **Text Redaction via BBox or Boxes**

2. **Optional Image Redaction**

Example: How Redaction Is Applied Per Sensitive Item

Here's a sample from _process_sensitive_item():

```
# Check if the item contains multiple boxes.
if "boxes" in item and isinstance(item["boxes"], list):

  # Iterate over each box in the list.
  for box in item["boxes"]:

    # Create a rectangle using the box coordinates.
    rect = pymupdf.Rect(box["x0"], box["y0"], box["x1"], box["y1"])

    # Add a redaction annotation with an optional entity type label.
    page.add_redact_annot(rect, fill=(0, 0, 0), text=item.get("entity_type"))

    # Log the redaction of the sensitive item.
    log_info(f"[OK] Redacting entity {item.get('entity_type')} on page {page_num}")

# Otherwise, check if a single bounding box is provided.
elif "bbox" in item:

  # Extract the bounding box.
  bbox = item["bbox"]

  # Create a rectangle from the bounding box coordinates.
  rect = pymupdf.Rect(bbox["x0"], bbox["y0"], bbox["x1"], bbox["y1"])

  # Add a redaction annotation.
  page.add_redact_annot(rect, fill=(0, 0, 0))
```

Then finally, the page is finalized:

```
page.apply_redactions()
```

Redaction Mapping Format

The ***input*** redaction mapping is structured like this:

```
{
  "pages": [
    {
      "page": 1,
      "sensitive": [
        {
          "bbox": { "x0": 100, "y0": 150, "x1": 200, "y1": 165 },
          "entity_type": "EMAIL"
        },
        {
          "boxes": [
            { "x0": 100, "y0": 150, "x1": 120, "y1": 165 },
            { "x0": 125, "y0": 150, "x1": 140, "y1": 165 }
          ],
          "entity_type": "NAME"
        }
      ]
    }
  ]
}
```

This allows both **single-box** and **multi-box** redaction per page.

Post-Redaction Sanitization

After redactions are applied, the _sanitize_document() method:

- Clears metadata: self.doc.set_metadata({})
- Deletes non-redaction annotations
- Scrubs hidden content with self.doc.scrub()

This ensures that **no sensitive data remains in hidden layers** or metadata.

Summary

Feature	**Description**
Input Types	Path, file-like, bytes, PyMuPDF Document

Feature	**Description**
Output Types	File or in-memory (bytes)
Redaction Mechanism	Rectangle annotation + page.apply_redactions()
Supports	Text and optional image redaction
Mapping Structure	Page-wise, supports bbox and multiple boxes per entity
Thread-Safe	Yes, with TimeoutLock
Metadata Cleanup	Yes
Hidden Content Removal	Yes

🔍 Entity Detection System

The entity detection system is designed to find sensitive information (such as names, emails, IDs) in text extracted from PDF pages. Each detection engine implements the shared asynchronous method:

```
async def detect_sensitive_data_async(extracted_data: Dict[str, Any], requested_entities: Optional[List[str]]) -> Tuple[List[Dict[str, Any]], Dict[str, Any]]
```

Every engine:

1. Receives structured PDF text as input (extracted_data with page content and bounding boxes).
2. Reconstructs text from each page.
3. Detects entities using its internal logic (AI, rule-based, or transformer).
4. Returns a list of detected entities and a redaction mapping based on bounding boxes.

Shared Input Format

Each engine processes input in this format:

```
{
 "pages": [
  {
   "page": 1,
   "words": [
    { "text": "John", "x0": 100, "y0": 200, "x1": 150, "y1": 210 },
    ...
   ]
  }
 ]
}
```

Before detecting anything, most engines **rebuild the full page text and bounding box mapping** using this utility:

```
full_text, text_mapping = TextUtils.reconstruct_text_and_mapping(words)
```

This gives:

- full_text: the complete text of the page
- text_mapping: mapping from character offsets to bounding boxes

Detection Engines Explained

1. PresidioEntityDetector
Engine Type: Rule-based (Microsoft Presidio)

Detection Flow:

- For each page:
    1. Rebuilds text using TextUtils.reconstruct_text_and_mapping
    2. Validates requested entities
    3. Analyzes text using Presidio:

       ```
       # Run the analysis using Presidio's analyzer.
        result = self.analyzer.analyze(text=text, language=language, entities=entities)
       ```
    4. Returns detected entities and bounding boxes

Output Format:
Each detected entity includes:

```
{
 "text": "example@example.com",
 "entity_type": "EMAIL_ADDRESS",
 "page": 1,
 "score": 0.99,
 "bbox": { ... }
}
```

2. GeminiEntityDetector
Engine Type: AI-based (Gemini LLM)

Detection Flow:

- For each page:
    1. Rebuilds text using TextUtils.reconstruct_text_and_mapping
    2. Constructs a prompt using Gemini Helper.
    3. Sends prompt to Gemini API by using Gemini Helper.

response = await self.send_request(prompt, requested_entities)

4. Parses the response with by using Gemini Helper.:

   # Parse the response into JSON format if a response was received.

   result = self.parse_response(response) if response else None

5. Enriches entities with original text spans and remaps using character positions

Notes:

- Token remapping is performed by _maybe_add_original_text()
- Handles multi-line or split words by combining spans

3. GLiNEREntityDetector / HIDEMEEntityDetector

Files: gliner.py, glinerbase.py, hideme.py
Engine Type: Local Transformer Model

Detection Flow:

- Uses a preloaded model (GLiNER format)
- For each page:
  1. Reconstructs full text using TextUtils.reconstruct_text_and_mapping
  2. Splits page into text paragraphs
  3. Predicts entities via:

     batch_results = self._process_paragraph_batch(...)

  4. Deduplicates and filters false positives

Special Notes:

- HIDEME uses the same base logic but with a specialized model
- Supports filter_pronouns, dedup_spans, and caching

4. HybridEntityDetector

File: hybrid.py
Engine Type: Composite (multi-engine)

Detection Flow:

- Accepts a configuration like:
- { "use_presidio": True, "use_gliner": True, "use_gemini": False, "use_hideme": False }
- For each enabled engine:

1. Loads the engine using initialization_service.get_presidio_detector()
2. Runs detection concurrently:

   parallel_results = await self._run_all_detectors_in_parallel(...)
3. Merges and deduplicates results from all engines: _finalize_detection(....)

Benefits:

- Robust fallback across engines
- Merges bbox + entity_type matches to avoid duplication

Output Format (All Engines)

All detectors return a tuple:

(List[Dict], Dict)

1. **Detected Entities**:

```
[
 {
   "text": "John",
   "entity_type": "PERSON",
   "page": 1,
   "score": 0.92,
   "bbox": { "x0": ..., "y0": ..., "x1": ..., "y1": ... }
 },
 ...
]
```

2. **Redaction Mapping**:

```
{
 "pages": [
  {
    "page": 1,
    "sensitive": [
     { "bbox": { ... }, "entity_type": "PERSON" },
     ...
```

```
      ]
    }
  ]
}
```

This redaction mapping can be passed directly to the PDFRedactionService to apply visual redactions.

Summary

Detector	**Type**	**Input Text Built With**	**Detection Mechanism**	**Mapping Method**
Presidio	Rule-based	TextUtils.reconstruct_text_and_mapping	NLP rules via Presidio	Character offsets → bbox
Gemini	AI (Cloud)	Same	Prompt + Gemini API	Remap via _maybe_add_original_text
GLiNER/HIDEME	Transformer	Same	Token span prediction	****
Hybrid	Composite	Each sub-engine handles its own	Aggregates results	Deduplicates merged mappings