**1. Entry Point: run\_sign\_range\_inclusion**

def run\_sign\_range\_inclusion(self, target\_class: dict):

"""

This function will:

1. Find consult class from the doctor ranges

2. Merge output in the pred final column

Return the dataframe with appended consult predictions in pred final column

"""

try:

doctor\_range\_dict, page\_indexed\_num\_desig\_dict = (

RangeExtractor.\_doctor\_range\_find(self.df.copy())

)

if "consult" in target\_class:

self.df = self.init\_consult\_range\_extractor.merge\_with\_consult\_index(

doctor\_range\_dict, page\_indexed\_num\_desig\_dict

)

except Exception as e:

LOG.error("Range inclusion stucked in some error due to {}".format(e))

return self.df

**How it works:**

1. Makes a copy of the dataframe to avoid modifying the original
2. Calls \_doctor\_range\_find to get two dictionaries:
   * doctor\_range\_dict: Maps files to page ranges associated with doctors
   * page\_indexed\_num\_desig\_dict: Maps files to lists of designation counts per page
3. If "consult" is in the target classes, calls merge\_with\_consult\_index to update the predictions
4. Returns the modified dataframe

**2. Doctor Range Finding: \_doctor\_range\_find**

@staticmethod

def \_doctor\_range\_find(df):

"""

This function will:

1. Find designation in the MR text lines

2. Find doctor's presence in the designation lines

3. Find sign patterns in MR text lines

4. Find doctor's presence in the signed lines

Return doctor range dictionary and predicted class range dictionary

"""

init\_designation\_finder = DesignationFinder(df)

df = init\_designation\_finder.designation\_handler()

df["num\_of\_desig"] = df["designations"].apply(lambda x: len(x))

init\_desig\_count\_generator = DesigCountGenerator(df)

page\_indexed\_num\_desig\_dict = init\_desig\_count\_generator.find\_page\_range()

df["dr\_names"] = df["designation\_lines"].apply(

lambda x: RangeRegexHandler.find\_doctors(x)

)

df["electronic\_sign"] = df["text\_raw"].apply(

lambda x: RangeRegexHandler.find\_electronically\_signed\_by(x)

)

df["electronic\_sign\_dr\_names"] = df["electronic\_sign"].apply(

lambda x: RangeRegexHandler.find\_doctors(x)

)

init\_range\_generation = RangeGenerator(df)

doctor\_range\_dict = init\_range\_generation.find\_doctors\_range()

return doctor\_range\_dict, page\_indexed\_num\_desig\_dict

**How it works step-by-step:**

**2.1 Finding Designations:**

init\_designation\_finder = DesignationFinder(df)

df = init\_designation\_finder.designation\_handler()

* Creates a DesignationFinder instance
* Calls designation\_handler(), which:
  + Applies \_find\_designation() to each row's text\_raw column
  + \_find\_designation() uses regex pattern SIGN\_DR\_DESIG to find medical designations
  + Returns both the designations and the lines they appear in
  + Filters rows to keep only those with designations
  + Adds two columns: designations and designation\_lines

**2.2 Counting Designations:**

df["num\_of\_desig"] = df["designations"].apply(lambda x: len(x))

init\_desig\_count\_generator = DesigCountGenerator(df)

page\_indexed\_num\_desig\_dict = init\_desig\_count\_generator.find\_page\_range()

* Counts the number of designations in each row
* Creates a DesigCountGenerator instance
* Calls find\_page\_range(), which:
  + Gets the max page number for each file using \_max\_page\_file\_dict()
  + Creates a dictionary mapping each file to a list of designation counts per page
  + For each file, the list index is the page number, and the value is the designation count

**2.3 Finding Doctor Names:**

df["dr\_names"] = df["designation\_lines"].apply(

lambda x: RangeRegexHandler.find\_doctors(x)

)

* Applies find\_doctors() to the designation lines
* find\_doctors() uses the DOCTOR\_PATTERN regex to find doctor names
* Returns a list of tuples: (line\_number, [doctor\_names])

**2.4 Finding Electronic Signatures:**

df["electronic\_sign"] = df["text\_raw"].apply(

lambda x: RangeRegexHandler.find\_electronically\_signed\_by(x)

)

* Applies find\_electronically\_signed\_by() to the raw text
* Splits the text by newlines
* Uses ES\_PATTERN regex to find electronic signature patterns
* Captures up to 2 lines after the match
* Returns a list of tuples: (line\_number, text\_with\_signature)

**2.5 Finding Doctor Names in Electronic Signatures:**

df["electronic\_sign\_dr\_names"] = df["electronic\_sign"].apply(

lambda x: RangeRegexHandler.find\_doctors(x)

)

* Applies find\_doctors() to the electronic signature text
* Finds doctor names in the electronic signature lines

**2.6 Generating Doctor Ranges:**

init\_range\_generation = RangeGenerator(df)

doctor\_range\_dict = init\_range\_generation.find\_doctors\_range()

* Creates a RangeGenerator instance
* Calls find\_doctors\_range(), which:
  + Processes each file separately
  + For each file, calls \_electronic\_sign\_processing() which:
    - Filters out pages with no electronic signatures or doctor names
    - For each electronic signature:
      * Finds the nearest doctor name before the signature
      * Determines which pages are part of this doctor's range
      * Handles complex cases like multiple signatures
    - Creates a dictionary mapping start pages to end pages for each doctor

**3. Merging with Consult Index: merge\_with\_consult\_index**

def merge\_with\_consult\_index(self, doctor\_ranges\_dict, page\_indexed\_num\_desig\_dict):

# Step 1: Find existing consult pages

consult\_df = self.df[

self.df["pred\_final"].apply(lambda x: "consult" in str(x))

].reset\_index(drop=True)

# Step 2: Group consult pages by file

page\_dict\_set = {}

for ind, row in consult\_df.iterrows():

if row["file\_name"] not in page\_dict\_set:

page\_dict\_set[row["file\_name"]] = [row["page"]]

else:

page\_dict\_set[row["file\_name"]].append(row["page"])

# Step 3: Find potential start pages

potential\_start = {}

for file in page\_dict\_set.keys():

page\_list = page\_dict\_set[file]

for page in page\_list:

if page - 1 not in page\_list:

if file not in potential\_start:

potential\_start[file] = [page]

else:

potential\_start[file].append(page)

# Step 4: Match potential starts with doctor ranges

ps\_doctor\_range = {}

for file in potential\_start.keys():

ps\_page\_list = potential\_start[file]

if file in doctor\_ranges\_dict:

page\_range\_dict = doctor\_ranges\_dict[file]

for page in page\_range\_dict:

if page in ps\_page\_list:

if file not in ps\_doctor\_range:

ps\_doctor\_range[file] = {page: page\_range\_dict[page][0]}

else:

ps\_doctor\_range[file][page] = page\_range\_dict[page][0]

# Step 5: Validate and filter ranges

ps\_doctor\_range = self.update\_range(ps\_doctor\_range)

ps\_doctor\_range = ConsultRangeExtractor.\_remove\_labs\_orders\_like\_classes(

ps\_doctor\_range, page\_indexed\_num\_desig\_dict

)

# Step 6: Create a dictionary of pages to add "consult" to

consult\_added\_page\_dict = {}

for file in ps\_doctor\_range.keys():

con\_page\_dict = ps\_doctor\_range[file]

for page\_st in con\_page\_dict.keys():

page\_end = con\_page\_dict[page\_st]

if isinstance(page\_st, str):

page\_st = eval(page\_st)

if isinstance(page\_end, str):

page\_end = eval(page\_end)

if file not in consult\_added\_page\_dict:

consult\_added\_page\_dict[file] = list(range(page\_st, page\_end + 1))

else:

consult\_added\_page\_dict[file].extend(

list(range(page\_st, page\_end + 1))

)

# Step 7: Update predictions

self.df["pred\_final"] = self.df.apply(

lambda x: ConsultRangeExtractor.\_append\_consult\_class(

x["file\_name"], x["page"], x["pred\_final"], consult\_added\_page\_dict

),

axis=1,

)

return self.df

**How it works step-by-step:**

**3.1 Find Existing Consult Pages:**

consult\_df = self.df[

self.df["pred\_final"].apply(lambda x: "consult" in str(x))

].reset\_index(drop=True)

* Filters the dataframe to keep only rows where "consult" is already in the predictions

**3.2 Group Consult Pages by File:**

page\_dict\_set = {}

for ind, row in consult\_df.iterrows():

if row["file\_name"] not in page\_dict\_set:

page\_dict\_set[row["file\_name"]] = [row["page"]]

else:

page\_dict\_set[row["file\_name"]].append(row["page"])

* Creates a dictionary mapping each file to a list of pages already classified as "consult"

**3.3 Find Potential Start Pages:**

potential\_start = {}

for file in page\_dict\_set.keys():

page\_list = page\_dict\_set[file]

for page in page\_list:

if page - 1 not in page\_list:

if file not in potential\_start:

potential\_start[file] = [page]

else:

potential\_start[file].append(page)

* For each file, finds pages that are the start of a "consult" section
* A start page is one where the previous page is not a "consult"

**3.4 Match Potential Starts with Doctor Ranges:**

ps\_doctor\_range = {}

for file in potential\_start.keys():

ps\_page\_list = potential\_start[file]

if file in doctor\_ranges\_dict:

page\_range\_dict = doctor\_ranges\_dict[file]

for page in page\_range\_dict:

if page in ps\_page\_list:

if file not in ps\_doctor\_range:

ps\_doctor\_range[file] = {page: page\_range\_dict[page][0]}

else:

ps\_doctor\_range[file][page] = page\_range\_dict[page][0]

* Matches potential start pages with doctor signature ranges
* If a doctor signature range starts on a potential consult start page, it's included in ps\_doctor\_range

**3.5 Validate and Filter Ranges:**

ps\_doctor\_range = self.update\_range(ps\_doctor\_range)

ps\_doctor\_range = ConsultRangeExtractor.\_remove\_labs\_orders\_like\_classes(

ps\_doctor\_range, page\_indexed\_num\_desig\_dict

)

* First, update\_range():
  + Creates a predication dictionary using \_create\_pred\_final\_dict()
  + For each range, checks if it's valid using \_valid\_range()
  + A range is valid if no other class has > 75% presence
  + Returns only the valid ranges
* Then, \_remove\_labs\_orders\_like\_classes():
  + For each range, calculates the number of designations
  + If there are too many designations (> 4 per page on average), it's likely not a consult
  + Returns only ranges with an appropriate designation density

**3.6 Create a Dictionary of Pages to Add "consult" to:**

consult\_added\_page\_dict = {}

for file in ps\_doctor\_range.keys():

con\_page\_dict = ps\_doctor\_range[file]

for page\_st in con\_page\_dict.keys():

page\_end = con\_page\_dict[page\_st]

if isinstance(page\_st, str):

page\_st = eval(page\_st)

if isinstance(page\_end, str):

page\_end = eval(page\_end)

if file not in consult\_added\_page\_dict:

consult\_added\_page\_dict[file] = list(range(page\_st, page\_end + 1))

else:

consult\_added\_page\_dict[file].extend(

list(range(page\_st, page\_end + 1))

)

* Converts the doctor ranges into a flat list of pages for each file
* Handles cases where the page numbers are stored as strings
* Creates a dictionary mapping each file to a list of pages to add "consult" to

**3.7 Update Predictions:**

self.df["pred\_final"] = self.df.apply(

lambda x: ConsultRangeExtractor.\_append\_consult\_class(

x["file\_name"], x["page"], x["pred\_final"], consult\_added\_page\_dict

),

axis=1,

)

* Applies \_append\_consult\_class() to each row
* \_append\_consult\_class():
  + Checks if the file and page are in the consult\_added\_page\_dict
  + If yes and "consult" isn't already in the predictions, adds it
  + Returns the updated prediction list

**4. Supporting Methods: Finding and Matching Patterns**

**4.1 Doctor Name Finding:**

@staticmethod

def find\_doctors(text\_list):

doctor\_list = []

for text in text\_list:

pattern = re.compile(DOCTOR\_PATTERN, re.VERBOSE)

matches = pattern.findall(text[1])

if matches != []:

doctor\_list.append((text[0], matches))

return doctor\_list

* Takes a list of (line\_number, text) tuples
* Uses DOCTOR\_PATTERN regex to find doctor names
* Returns a list of (line\_number, [doctor\_names]) tuples

**4.2 Electronic Signature Finding:**

@staticmethod

def find\_electronically\_signed\_by(text\_list):

text\_list = text\_list.split("\n")

signed = []

for ind, text in enumerate(text\_list):

matches = re.finditer(ES\_PATTERN, text)

match\_list = [match.group(0) for match in matches]

if match\_list != []:

match = match\_list[0]

next\_2\_line\_text = ""

for line\_text in text\_list[ind : ind + 2]:

next\_2\_line\_text += " " + line\_text

signed.append((ind, next\_2\_line\_text))

return signed

* Takes a text string, splits it by newlines
* Uses ES\_PATTERN regex to find electronic signature phrases
* When a match is found, captures that line and up to 2 lines after it
* Returns a list of (line\_number, text) tuples

**4.3 Fuzzy Doctor Name Matching:**

@staticmethod

def \_check\_fuzzy\_doctor(dr, dr\_list):

dr = dr.translate(str.maketrans("", "", string.punctuation))

for doctor in dr\_list:

doctor = doctor.translate(str.maketrans("", "", string.punctuation))

if fuzz.token\_set\_ratio(dr, doctor) == 100:

return True

return False

* Removes punctuation from both doctor names
* Uses fuzz.token\_set\_ratio for fuzzy matching
* Returns True if there's a 100% token set match

**4.4 Doctor Range Processing:**

@staticmethod

def \_electronic\_sign\_processing(df\_1):

# [Code omitted for brevity]

# For each page with electronic signatures

for page in electronically\_signed\_dict.keys():

edr\_list = electronically\_signed\_dict[page]

# For each electronic signature

for eline, dr in edr\_list:

# Get the doctor name

edoctor\_name = dr[0]

if edoctor\_name not in indexed\_doctors:

indexed\_doctors[edoctor\_name] = {}

# Process different range scenarios

# [Complex logic for determining page ranges]

# Clean and format the results

# [Code for cleaning and formatting]

return page\_start\_dict

* This is the most complex method, determining which pages belong to which doctors
* Handles scenarios like:
  + Doctor name appears before electronic signature on the same page
  + Doctor name appears on a previous page before the signature
  + Doctor name appears on multiple pages

**Key Logic Patterns and Algorithms**

1. **Pattern Matching**:
   * Uses regex patterns (likely complex ones) to identify:
     + Doctor names with titles
     + Electronic signatures
     + Medical designations
2. **Fuzzy Matching**:
   * Uses token set ratio for fuzzy matching of doctor names
   * Handles variations in punctuation and spacing
3. **Range Detection**:
   * Uses doctor names and signatures to determine page ranges
   * Handles complex cases like:
     + Multiple doctors in the same document
     + Multiple signatures for the same doctor
     + Signatures and doctor names on different pages
4. **Filtering and Validation**:
   * Uses designation density to filter out lab reports
   * Uses existing class predictions to validate ranges
   * Ensures consult classifications don't override strong existing classifications
5. **Page Continuity Logic**:
   * Identifies start and end pages of consults
   * Ensures consults are continuous page ranges
   * Handles potential breaks in continuity

Does this detailed, code-level breakdown help you understand the logic? Let me know if you'd like me to elaborate on any specific part or if you have questions about how certain components work together.