User Guide

This user guide attempts to provide an overview of the file system and the code used to implement the **frontend** (User Interface) and the **backend** (The query system)

File structure

Query_annotation_tool (main folder)
tkinter_app (sub folder)
frontend
- main.py
- dummy.py
- import_instruct.txt
- annotation_instruct.txt
- search_instruct.txt

backend
- myMdl
- QueryParser.py
- QueryProcessor.py
- Boolean.py
- Field.py
- CaptureGroups.py
- Sequential.py

FRONTEND:
Contains files and code used to design the frontend

MAIN.PY
Contains code used to design the User Interface

DUMMY.PY
Contains code to link frontend with backend

IMPORT_INSTRUCT.TXT
Contains text instructing how files should be imported

ANNOTATION_INSTRUCT.TXT
Contains text instructing how to use the annotation tool

SEARCH_INSTRUCT.TXT
Contains text instructing how to use the search tool

BACKEND
Contains files and code used to perform queries

MYMDL
Contains the saved NER model.

Front end (User Interface)

The UI has been divided into 3 parts

Import files Tab

- This tab provides 2 options, namely to import a file (for annotation purposes) or a folder (for querying purposes).
- The imported file should be a text file (.txt extension) divided into sections with suitable headings. For Example, '{Abstract: The paper aims to study…, Introduction: We have performed experiments to….,}'. The annotation tool will search the specified file only.
- The imported folder should consist of text files in the format mentioned above. The search tool will search all the files in the folder.

Search Tool Tab

It enables the user to fire queries in specific formats. The query system performs a search on all the files in the imported directory and returns the results with the matches highlighted.

Queries can be written in the following formats:

BOOLEAN

It consists of a word(s) and their different combinations. For example, The user can search for

A single word:

substrate (searches for the word 'substrate' in the text)

Multiple words:

stainless steel (searches for the words 'stainless' or 'steel' occurring separately or together in any order from the text)

Multiple word combinations:

stainless|carbon steel (here it searches for the combinations 'stainless steel' and 'carbon steel')

FIELD

In addition to words, one can also search for a particular lemma, any predefined entity from molecules, actions, processes, quantities, and conditions or words having a specific POS (Part of speech) tag. The user can fire **single**, **multiple**, and **multiple word combinations** as in the case of Boolean queries. For Example, the user can search for

Single, Multiple words, and Multiple word combos:

word=substrate (searches for the word 'substrate' in the text)

word=stainless steel (searches for the words 'stainless' or 'steel' or both from the text occurring in any order)

word= stainless|word=carbon word=steel (here it searches for the combinations 'stainless steel' and 'carbon steel')

lemma=coating (searches for the words whose lemma is 'coating' like 'coatings' or 'coated')

tag=NN (searches for the words whose POS tag is 'NN')

Using|by entity=Process (searches for the combinations 'Using <process name>' or 'by <process name>'

Multiple pairs:
entity=Molecules entity=Process (searches for a sentence with the occurrence of either one of the entities or both)

CAPTURE GROUPS
These enable us to capture any field query in a variable. For example,
Ingredient: entity=Molecules process_name:entity=process (here any occurrence of a molecule name and a process name will be captured by the variable 'Ingredient')
Substrate:word=stainless|word=carbon word=steel (here the combination 'stainless steel' and 'carbon steel' will be captured by the variable 'Substrate')

SEQUENTIAL
In the above-mentioned queries, the order of occurrence of words wasn't considered, but in sequential queries, one can search for Boolean/Field/Capture groups in a particular order. For example,
stainless, steel (Here only those sentences will be returned who have the word 'stainless' followed by 'steel', so the order will be strictly followed)

GENERAL INSTRUCTIONS:
- To query write your query in the query field. Make sure to not add any redundant spaces and adhere to the formats described above.
- If the query contains capture groups, results will be exported in a CSV format with the column name as the variable name used in capturing entities.

For example, *substrate1:word=stainless steel substrate2:word=carbon steel*
The exported file will look like :

substrate1	substrate2
stainless steel	carbon steel
stainless steel	carbon steel
stainless steel	

For other types of queries like *word=stainless steel*
The exported file will look like:
stainless steel
stainless steel
stainless steel

Annotation Tool Tab:
This tab enables the user to fire certain queries and annotate the results obtained from the imported file.

GENERAL INSTRUCTIONS
- The format to write queries is: *<Entity name>: any Field query* (essentially a capture group with only one variable)
- The Entity column specifies the predefined entity names that the user can capture via queries.
- The letters beside each entity name specify the key to press to associate the selection to that entity.
- Only predefined entity names can be used in place of <Entity name> while firing a query

Frontend code

The front-end code is in the **main.py** and the **dummy.py** file.

MAIN.PY

truncate()
Helper function used by search tool and annotation tool classes.

class TabControl
tab_switch()
It is responsible for switching tabs that are namely, Import files, Search Tool, and Annotation Tool.

class AnnotationTool:
Implements the annotation part.
Handles exporting of annotated results.

For each entity the following variables have been defined:
Tag: To assign a particular color while highlighting entities.
Key: The key to press on the keyboard to highlight an entity
Text: Name of entity displayed
Var: To capture the entity number if the corresponding checkbox is checked during relational tagging.

ExportAction()
If the query contains a single entity then the exported file is automatically exported as <entity_name>.csv. Else, a name has to be provided.

Query()
The query function obtains the results and displays them in the proper format in the results field. It also stores the results to be exported later.

OnKeyPressA-L()
It implements the annotation part. This function contains two parts,
If tagging_mode is set to entity tagging (2) then annotated entities are added to the dictionary 'ent_dict'.
If tagging_mode is set to relational tagging. Annotated pairs are added to the dictionary ‘mul_ents’.

OnKeyPressX()
It implements the removal of highlighted entities. Again depending on the mode, it deletes the selected entry from the corresponding dictionary.

enable_annotation()
It checks which entities have been ticked for relational tagging and initializes ‘mul_ents’ with the selected ones.

clear_results_field()
Clears the results field and reinitializes the dictionaries.

update()
Creates the sections menu, after a file has been imported.

change_dropdown()
It stores the selected section in the ‘section’ variable.

Class ImportTab:
UploadActionFile()
Opens a dialog box and stores the input file path in the IP_DATASET variable.

UploadActionFolder()
It opens a dialog box and stores the input folder path in the IP_DATASET variable.

Class SearchTool:
Contains the following variables:
Section: To store the name of the section
Results: Captures 'non-capture group' results
Results_dict: Captures 'capture group' results
Type: Stores the type of query, capture/non-capture group.

update()
It creates a section menu after a folder has been imported.

Query()
It displays the results in the results field, highlighting the captured words.
Depending on the query fired if it's a non-capture group query it appends the results to a list called ‘results’. Else, it creates a dictionary called ‘results_dict’ where the keys are the variable names used in the query.

ExportAction()
It writes the results to the file selected by the user. It also resets the type.

clear_results_field()
Clears the results field and reinitializes the result dictionary/list. It also resets the type.

Class App
It displays the main window.

DUMMY.PY

Global variables:
ent_dict: For tagging mode equal to ‘entity tagging’ it stores the annotated results and the results returned from queries.

mul_ents: For tagging mode as ‘relational entity tagging’, it stores the annotated results along with the results returned from queries.

get_query()
Parameters: query, input filepath, section name
Returns: It returns results obtained from the backend.

getOptions()
Parameters: path to file
Returns: section names existing in the imported file.

getInstruct()
Parameters: instruction file's name
Returns: reads the file and returns text to display in the instruction fields of import/search tool/ annotation tabs.

clear_dicts()
Reinitializes both the dictionaries in the event of pressing the 'clear' button.

create_mulents()
Parameters: names of entities checked for relational tagging.
Returns: It initializes the mul_ents dictionary with the passed entity names as the keys.

create_muldict()
Adds annotations to mul_ents during relational entity tagging
Parameters: entity name, start index, end index, highlighted word(s)
Returns: If the entry is not already present, it appends a list consisting of start index, end index, and the highlighted word(s) to the mul_ents dictionary, i.e it stores the annotation performed by the user.

create_dict()
Parameters: entity name, start index, end index, highlighted word(s), an 'add' parameter that distinguishes if its text from results, or newly annotated text.
Returns: if the entry is not already present, then it appends a list of start index, end index, highlight word(s) to the ent_dict dictionary, i.e it stores the annotation performed by the user.

adding_again()
Parameters: highlighted text, start index, end index
Returns: Returns True if the entry already exists, else False.

export_as_search()
Parameters: list/dictionary of search results from the Search Tool
Returns: writes the results to a CSV file

export_as()
Parameters: tagging mode (entity tagging or relational entity tagging) and the query
Returns: If the query has a single entity name then the filename is <entity name>.csv else a filename has to be provided by the user. Entries are exported if the dictionaries (mul_ents or ent_dict) are not empty.

write_to_file()

Parameters: filename, results to write
Returns: writes results to the CSV file in a proper format

flatten()
Helper function to write_to_file()

BACKEND:

QueryParser.py
The main file that checks what kind of query (Boolean, Field, Capture groups, Sequential) it is and returns the appropriate results.

QueryProcessor.py
Has functions that aid in query processing.

Boolean.py/Field.py/CaptureGroups.py/Sequential.py
It has code to execute Boolean/Field/CaptureGroups/Sequential type of queries. More details about the types and the code are in the subsequent sections

QueryParser.py

Variables defined:
Query: stores the query entered by the user
ip_filename: stores the name of the file imported
Section: stores the section name selected by the user
Db: It stores the text from where the search is to be performed

ParseQuery()
Parses query according to its format and calls the corresponding module like Boolean, Field, etc. It also returns the results obtained which are received by the get_query function in the dummy.py file.

QueryProcesor:

Word()
Parameters: Takes a sentence and the word to find
Returns: All of the word's occurrences i.e its start and end indices in the sentence.

Lemma()
Parameters: Takes a sentence and the lemma word to find
Returns: Searches for the word in the sentence whose lemma is the same as the one specified by the user and returns its start and end index.

Entity()
Parameters: Takes a sentence and the entity name to search for.
Returns: Returns start and end indices of the words which are tagged as that entity name by the trained NER model.

PosTag()
Parameters: Takes a sentence and the POS tag
Returns: Returns start and end indices of the words which are tagged as that POS tag by NLTK's POS tagger

Combinations()
Parameters: query of the form *a|b c|d*
Returns: It finds the combinations to be searched for like, ac, ad, bc, and bd.

RemoveBrackets()
Parameters: query
Returns: removes brackets '()' from the query.

EvaluateExp()
Parameters: field query of the form '*word/lemma/tag/entity=xyz*' and a sentence
Returns: Searches for the word/lemma/tag/entity in the sentence and returns a list of the word's start index and end index

get_db()
Parameters: Input file path and the section name.
Returns: Parses and returns the particular section of the file in the database.

BOOLEAN

Variables:
Query: Captures query of type Boolean
Db: The text to search from is stored here.
Mode: Whether the results are to be returned to a capture group or sequential query.

SingleWord()
Parameter: query
Processes single word queries like 'steel',' substrate', etc

MultiWord()
Parameter: query
Processes multiword queries like 'stainless steel'

MultiWordCombos()
Parameter: query
Processes queries of the format 'carbon|stainless steel'. It computes the combinations that are 'carbon steel' and 'stainless steel' and finds there occurrences in the text

CheckQuery()
It Checks what type of Boolean query it is and calls the suitable function.

FIELD

Variables:
Query: Captures query of type Field
Db: The text to search from is stored here.
Mode: Whether the results are to be returned to a capture group or sequential query.

MultiWordCombos()
Parameter: query
Processes queries of the format 'Using|by entity=Process'. It computes the combinations that are 'Using <Process name>' and 'by <Process name>' and finds their occurrences in the text.

Multipair()
Parameter: query
Processes queries of the format 'entity=Molecules entity=Processes'. It finds occurrences of anyone or both of the entity names.

CheckQuery()
It Checks what type of Field query it is and calls the suitable function.

CAPTUREGROUPS
Variables:
Query: Captures query of type Field
Db: The text to search from is stored here.
Mode: Whether the results are to be returned to a sequential query.

MultiPair()
Parameter:query
Processes query of the format 'Substrate1:word=stainless steel Substrate2:word=carbon steel'. Also, it finds occurrences of any one or both of the variables.

MultiWordCombos()
*Parameter:*query
Processes query of the format '*Coating:word=coating Substrate:word=stainless|word=carbon|steel*'. Finds combinations if required and searches for them, like here combinations to find for variable 'Substrate' are 'stainless steel' and 'carbon steel'. Also, it finds occurrences of any one or both of the variables.

MultiWordOptional()
*Parameter:*query
Processes query of the format '*Property: anti-corrosive|anti-corrosion (? Coating:word=coating)*'. Finds combinations (if required) and searches for them. Like here combinations to find for variable 'Property' are 'anticorrosive coating' and 'anticorrosion coating'. Here, searching for the word 'coating' is optional. Also it, finds occurrences of anyone or both of the variables.

CheckQuery()
It Checks the type of Capture Group query and calls a suitable function.

SEQUENTIAL

Query: captures query of type Boolean
Db: the text to search from is stored here.

ParseQuery()
Parameter: query
It processes query of the format ‘stainless, steel’. Here while searching the order is strictly followed and only occurrences where ‘stainless’ and ‘steel’ come together in the same order are returned. Instead of ‘stainless’ or ‘steel’, it can be any Field, Boolean, or Capture group type query. The query is first split on each ‘,’ and every part is evaluated separately. The sentences containing occurrences of all parts and in the right order are returned.

CheckQuery()
It calls the ParseQuery function.