IPOP Cabinet

None

None

None

Table of contents

1. User Guide

**

**# data-utils

rename to something... cabinet

all sub-folders will be suffixed with "_drawer"

TODO: switch to mkdocs, boo sphinx

:)

metamap TOS: https://lhncbc.nlm.nih.gov/ii/tools/MetaMap/run-locally/Ts_and_Cs.html

does metamap have a "validated/authorized" endpoint?

windows install: https://lhncbc.nlm.nih.gov/ii/tools/MetaMap/Docs/README_win32.html

Times for metamap vs scispacy...

img

img

These are "straight up" and do not involve parallelization, in that regard:

- metmap benefits about 4x improvement from IO thread-based multiprocessing
- scispacy can utilize batch processing link and can benefit even further from CPU (cpus go zoom) parallelization in with batch processing and limiting pipeline models
- HOWEVER
- a fastapi endpoint (based on internet search) can easily serve about 200req/s which is faster than scispacy model of about 100it/s
- so ok to use endpoint
- we can further unlock this using websockets
- ner endpoint that does the SNOMED traversal
- something like... `post_ner(text: str, terminal_tree_node: str -> cui) ?

future work will turn this into a rust/python package using pyo3 and maturin... this isn't necessary for current system calls and may benefit network calls but will mostly be required for parsing the snomed source files when we support providing your own instead of just (implying we keep current functionality as well) the id-based maps

this will also be supported by rust libraries such as mmi-parser to parse out mmi output from the rest api

2. API Reference

2.1 Getting Started

This is where you can find source code documentation, examples, and more technical details.

Our cabinet of tools.

hlights

- Local MetaMap operations
- SciSpacy NER via API
- SNOMED tree traversal
- UMLS CUI to SNOMED CUI
- Common data normalization tasks

In general, we try to expose the high-level functionality of these tools at the top level of their corresponding "drawers" (e.g. `cabinet.umls_drawer`).

This way you can import a drawer and use its functionality specifically.

For example:

```
from cabinet import umls_drawer
umls_drawer.post_ner_single("I have a headache.")
```

If you want more granular control/exposure, check out the underscore methods inside the drawers although this is not recommended practice.

2.2 UMLS Drawer

This drawer is for UMLS related activities.

The `scispacy_ner` module gives you access to the scispacy biomedical NER model via our API.

The `metamap_ner` module interacts with the MetaMap NLP tool to extract structured information from biomedical text and requires you to have MetaMap installed locally.

The `knowledge_base` module allows you to interact with the UMLS knowledge base data at a high level and mostly focuses on SNOMED CT concepts. Further work on this module may take advantage of the *entire* UMLS, but require a locally downloaded copy due to licensing restrictions.

In general, we recommend using the `scispacy_ner` module for NER tasks and the `knowledge_base` module for knowledge base related tasks unless you specifically need the power of MetaMap.

The `post_ner` methods exposed here utilize the API to perform NER on your text.

This module contains functions for interacting with the scispacy NER model via our API.

The private functions utilize async/await syntax and are used by the public functions which are synchronous. The public functions are the ones that you should use in your code unless you are confident that you know what you are doing.

The core type of this module is `NEROutput` which is a pydantic model that represents the output from the scispacy NER model. All public functions return either an instance of this class or an iterator of instances of this class attached to an index (tuple[int, NEROutput]) for the index of the text that was submitted... this helps with link to original data.

`web_socket_ner`, specifically, returns an iterator and thus needs to be consumed to be used:

```
>>> from cabinet.umls_drawer.scispacy import web_socket_ner
>>> for text_index, ner_output in web_socket_ner(texts=["cocaine", "heroin", "cociane"]):
...     print(text_index, ner_output)
0 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cocaine"}' score=1.0
1 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"heroin"}' score=1.0
2 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cociane"}' score=1.0
```

2.2.1 `NEROutput`

Bases: `BaseModel`

Output from the (scispacy) NER model.

This class is keyword only so you must pass in the arguments as: `cui="C0004096", concept_name="Acetaminophen", ...`

Args/Attributes: cui (str): The UMLS CUI. concept_name (str): The UMLS concept name. concept_definition (str): The UMLS concept definition. entity (str): The entity that matched a UMLS concept from the source text. score (float): The score of the match.

Examples:

An example of manually creating this class:

```
>>> from cabinet.umls_drawer import NEROutput
>>> NEROutput(
...     cui="C0004096",
...     concept_name="Acetaminophen",
...     concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of rheumatoid
arthritis and osteoarthritis.",
...     entity="acetaminophen",
...     score=0.96,
... )
```

However, much more likely is that you get this class as a return type from one of the various functions in this module that make calls to our API.

```
>>> from cabinet.umls_drawer import post_ner_single
>>> post_ner_single(text="cocaine")
[
    0 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cocaine"}' score=1.0
]
```

Source code in `src/cabinet/umls_drawer/scispacy_ner.py`

~~~python
class NEROutput(BaseModel):
    """Output from the ([scispacy](https://github.com/allenai/scispacy/tree/4f9ba0931d216ddfb9a8f01334d76cfb662738ae)) NER model.

    This class is keyword only so you must pass in the arguments as: `cui="C0004096", concept_name="Acetaminophen", ...`

    Args/Attributes:
        cui (str): The UMLS CUI.
        concept_name (str): The UMLS concept name.
        concept_definition (str): The UMLS concept definition.
        entity (str): The entity that matched a UMLS concept from the source text.
        score (float): The score of the match.

    Examples:
        An example of manually creating this class:
        ```python
        >>> from cabinet.umls_drawer import NEROutput
        >>> NEROutput(
        ...     cui="C0004096",
        ...     concept_name="Acetaminophen",
        ...     concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of 
rheumatoid arthritis and osteoarthritis.",
        ...     entity="acetaminophen",
        ...     score=0.96,
        ... )
        ~~~

        However, much more likely is that you get this class as a return type from one of the various functions in this module that
        make calls to our API.
        ```python
        >>> from cabinet.umls_drawer import post_ner_single
        >>> post_ner_single(text="cocaine")
        [
            0 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cocaine"}' score=1.0
        ]
        ```

    """

    cui: str
    """The UMLS CUI."""
    concept_name: str
    """The UMLS concept name."""
    concept_definition: str
    """The UMLS concept definition."""
    entity: str
    """The entity that matched a UMLS concept from the source text."""
    score: float
    """The score of the match."""
```

`concept_definition: str` `class-attribute`

The UMLS concept definition.

`concept_name: str` `class-attribute`

The UMLS concept name.

`cui: str` `class-attribute`

The UMLS CUI.

`entity: str` `class-attribute`

The entity that matched a UMLS concept from the source text.

`score: float` `class-attribute`

The score of the match.

2.2.2 `post_ner_many(texts, with_progress=True)`

Submit multiple text blobs to the scispacy NER model and return the results.
```

**Parameters:**

| Name | Type | Description | Default |
|---|---|---|---|
| `texts` | `list[str]` | The texts to submit to the NER model. | *required* |
| `with_progress` | `bool` | Whether or not to show a progress bar. Defaults to True. | `True` |

**Returns:**

| Type | Description |
|---|---|
| `list[tuple[int, NEROutput]]` | Iterator[tuple[int, NEROutput]]: The results from the NER model. |

**Raises:**

| Type | Description |
|---|---|
| `Exception` | If the response status is not 200. |

**Example**

~~~python

from cabinet.umls_drawer import post_ner_many post_ner_many(texts=["acetaminophen", "ibuprofen"]) [ (0, NEROutput( cui="C0004096", concept_name="Acetaminophen", concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of rheumatoid arthritis and osteoarthritis.", entity="acetaminophen", score=0.96, )), (1, NEROutput( cui="C0004096", concept_name="Ibuprofen", concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of rheumatoid arthritis and osteoarthritis.", entity="ibuprofen", score=0.96, )) ]

**Source code in `src/cabinet/umls_drawer/scispacy_ner.py`**

```python
@validate_arguments
def post_ner_many(
    texts: list[str],
    with_progress: bool = True,
) -> list[tuple[int, NEROutput]]:
    """Submit multiple text blobs to the scispacy NER model and return the results.

    Args:
        texts (list[str]): The texts to submit to the NER model.
        with_progress (bool, optional): Whether or not to show a progress bar. Defaults to True.

    Returns:
        Iterator[tuple[int, NEROutput]]: The results from the NER model.

    Raises:
        Exception: If the response status is not 200.

    Example:
        ```python
        >>> from cabinet.umls_drawer import post_ner_many
        >>> post_ner_many(texts=["acetaminophen", "ibuprofen"])
        [
            (0, NEROutput(
                cui="C0004096",
                concept_name="Acetaminophen",
                concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of
rheumatoid arthritis and osteoarthritis.",
                entity="acetaminophen",
                score=0.96,
            )),
            (1, NEROutput(
                cui="C0004096",
                concept_name="Ibuprofen",
                concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of
rheumatoid arthritis and osteoarthritis.",
                entity="ibuprofen",
                score=0.96,
            ))
        ]
    """
    return asyncio.run(_post_ner_many(texts, with_progress=with_progress))
~~~

### 2.2.3 `post_ner_single(text)`

Submit a single text blob to the scispacy NER model and return the results.

**Parameters:**

| Name | Type | Description | Default |
|---|---|---|---|
| `text` | `str` | The text to submit to the NER model. | *required* |

**Returns:**

| Type | Description |
|---|---|
| `list[NEROutput]` | list[NEROutput]: The results from the NER model. |

**Raises:**

| Type | Description |
|---|---|
| `Exception` | If the response status is not 200. |

**Example**

```python

from cabinet.umls_drawer import post_ner_single post_ner_single(text="acetaminophen") [ NEROutput( cui="C0004096", concept_name="Acetaminophen", concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of rheumatoid arthritis and osteoarthritis.", entity="acetaminophen", score=0.96, ) ]

**Source code in `src/cabinet/umls_drawer/scispacy_ner.py`**

```
@validate_arguments
def post_ner_single(text: str) -> list[NEROutput]:
    """Submit a single text blob to the scispacy NER model and return the results.

    Args:
        text (str): The text to submit to the NER model.

    Returns:
        list[NEROutput]: The results from the NER model.

    Raises:
        Exception: If the response status is not 200.

    Example:
        ```python
        >>> from cabinet.umls_drawer import post_ner_single
        >>> post_ner_single(text="acetaminophen")
        [
            NEROutput(
                cui="C0004096",
                concept_name="Acetaminophen",
                concept_definition="A nonsteroidal anti-inflammatory drug that is used as an analgesic and antipyretic. It is also used in the treatment of 
rheumatoid arthritis and osteoarthritis.",
                entity="acetaminophen",
                score=0.96,
            )
        ]
    """
    return asyncio.run(_post_nlp_single(text))
```

2.2.4 `websocket_ner(texts, with_progress=True)` `async`

Connect to the scispacy NER model websocket and submit texts.

IMPORTANT: This function requires using the `async for` syntax and thus may not work in all scenarios or environments. It exists for **very** large datasets where the overhead of the HTTP request/response cycle is too much.

Parameters:

Name	Type	Description	Default
`texts`	`list[str]`	The texts to submit to the NER model.	*required*
`with_progress`	`bool`	Whether or not to show a progress bar. Defaults to True.	`True`

Yields:

Type	Description
`AsyncIterator[tuple[int, NEROutput]]`	AsyncIterator[tuple[int, NEROutput]]: The results from the NER model.

Raises:

Type	Description
`Exception`	If the response status is not 200.

Example

~~~python

from cabinet.umls_drawer import websocket_ner async for i, result in websocket_ner(texts=["acetaminophen", "ibuprofen"]): ... print(i, result) 0 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cocaine"}' score=1.0 1 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"heroin"}' score=1.0 2 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cociane"}' score=1.0

**Source code in `src/cabinet/umls_drawer/scispacy_ner.py`**

```python
@validate_arguments
async def websocket_ner(
    texts: list[str], with_progress: bool = True
) -> AsyncIterator[tuple[int, NEROutput]]:
    """Connect to the scispacy NER model websocket and submit texts.

    *IMPORTANT*: This function requires using the `async for` syntax and thus may not work in all scenarios or environments.
    It exists for **very** large datasets where the overhead of the HTTP request/response cycle is too much.

    Args:
        texts (list[str]): The texts to submit to the NER model.
        with_progress (bool, optional): Whether or not to show a progress bar. Defaults to True.

    Yields:
        AsyncIterator[tuple[int, NEROutput]]: The results from the NER model.

    Raises:
        Exception: If the response status is not 200.

    Example:
        ```python
        >>> from cabinet.umls_drawer import websocket_ner
        >>> async for i, result in websocket_ner(texts=["acetaminophen", "ibuprofen"]):
        ...     print(i, result)
        0 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cocaine"}' score=1.0
        1 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"heroin"}' score=1.0
        2 cui='12' concept_name='test' concept_definition='test22' entity='{"text":"cociane"}' score=1.0
    """
    async with aiohttp.ClientSession(_WS_URL) as session:
        async with session.ws_connect("/models/ner/ws") as ws:
            match with_progress:
                case True:
                    for i, text in tqdm_asyncio(enumerate(texts)):
                        await ws.send_bytes(orjson.dumps({"text": text}))
                        raw_data = await ws.receive_bytes()
                        response_data = orjson.loads(raw_data)
                        data = NEROutput(**response_data)
                        yield (i, data)
                case False:
                    for i, text in enumerate(texts):
                        await ws.send_bytes(orjson.dumps({"text": text}))
                        raw_data = await ws.receive_bytes()
                        response_data = orjson.loads(raw_data)
                        data = NEROutput(**response_data)
                        yield (i, data)
~~~

This module is for working with MetaMap.

MetaMap is a tool for extracting structured information from biomedical text provided by the National Library of Medicine (NLM). MetaMap is a text processing engine is a natural language processing (NLP) system that uses a set of rules and heuristics to identify and extract concepts from unstructured text.

You can download MetaMap by purchasing a NLM License (or accessing via your institution) and downloading the binary from here.

For more information, see the MetaMap documentation.

2.2.5 `MMOutputType`

Bases: `enum.Enum`

Enum for MetaMap output types.

Source code in `src/cabinet/umls_drawer/metamap_ner.py`

```python
class MMOutputType(enum.Enum):
    """Enum for MetaMap output types."""

    MMI = "mmi"
    """Fielded MMI output format, see [here](https://lhncbc.nlm.nih.gov/ii/tools/MetaMap/Docs/MMI_Output.pdf) for more information."""
    JSON = "json"
    """Json output, see [here](https://lhncbc.nlm.nih.gov/ii/tools/MetaMap/Docs/JSON.pdf) for more information."""
```

`JSON = 'json'` `class-attribute`

Json output, see here for more information.

`MMI = 'mmi'` `class-attribute`

Fielded MMI output format, see here for more information.

2.2.6 `MetaMap`

Bases: `BaseModel`

Class for running MetaMap on a text string.

Parameters:

Name	Type	Description	Default
`metamap_location`	`str`	The path to the MetaMap installation. This should be the path to the `public_mm` directory.	*required*

Example

```python
from cabinet.umls_drawer import MetaMap
mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
```

Source code in `src/cabinet/umls_drawer/metamap_ner.py`

~~~python
class MetaMap(BaseModel):
    """Class for running MetaMap on a text string.

    Args:
        metamap_location (str): The path to the MetaMap installation. This should be the path to the `public_mm` directory.

    Example:
        ```python
        from cabinet.umls_drawer import MetaMap
        mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
        ~~~
    """

    metamap_location: Path
    _initialized: bool = PrivateAttr(default=False)

    @validator("metamap_location")
    def _check_metamap_location(cls, v: str) -> Path:
        """Check the MetaMap location."""
        path = Path(v).expanduser()
        if path.exists() is False:
            raise FileNotFoundError(
                "`metamap_location` does not exist. Please check the path."
)
        elif path.is_dir() is False:
            raise NotADirectoryError(
                "`metamap_location` is not a directory. Please check the path. We are expecting the path to the `public_mm` directory."
)
        return path

    def initialize(self) -> None:
        """Initialize MetaMap.

        This function must be run to start the MetaMap servers.
        It will check if the servers are already running and if not, it will start them.
        """
        # use generic "metamap" so supports any version
        # but make check here that not older than 2016
        # if older than 2016v2, raise error
        # check initialization cache-file- to see if already initialized
        if _check_metamap_servers_status() is True:
            c.print(
                "[green bold]INFO:[/] [green]MetaMap servers are already running.[/]"
)
            self._initialized = True
            return None
        else:
            c.print(
                "[yellow bold]WARNING:[/] [yellow]MetaMap servers are not running. Starting servers now.[/]"
)
            skrmed_server = self.metamap_location / "bin" / "skrmedpostctl"
            wsd_server = self.metamap_location / "bin" / "wsdserverctl"
            os.system(skrmed_server.as_posix() + " start")
            os.system(wsd_server.as_posix() + " start")
            with c.status(
                "Waiting for servers...", spinner="dots", spinner_style="yellow"
            ):
                time.sleep(60)
            if _check_metamap_servers_status() is True:
                c.print(
                    "[green bold]INFO:[/] [green]MetaMap servers are now running.[/]"
)
                self._initialized = True
                return None
            else:
                c.print(
                    "[red bold]ERROR:[/] [red]MetaMap servers failed to start.[/] Please check the MetaMap installation and try again."
)
                return None

    @validate_arguments
    def run(
        self, text: str, output_type: Literal["mmi", "json"] = "mmi"
) -> list[str] | str | None:
        """Run MetaMap on a text string.

        This will return None if no results were found, otherwise the return type will
        match the output_type argument. Future work will include returning a dataclass
        for each result, but for now:
            MMOutputType.JSON -> str (the json data itself)
            MMOutputType.MMI -> list[str] (each line of the MMI output)

        Args:
            text (str): The text to run MetaMap on.
            output_type (Literal["mmi", "json"], optional): The output type. Defaults to "mmi".

        Returns:
            list[str] | str | None: The output of MetaMap. The type of the output depends on the `output_type` argument.

        Example:
            ```python
            >>> from cabinet.umls_drawer import MetaMap, MMOutputType
            >>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
            >>> mm.initialize()
            >>> results = mm.run(text="I have a headache.", output_type=MMOutputType.MMI)
            >>> print(results)
            ```
        """
        if self._initialized is False:
            c.print(
```
```

~~~
                "[red bold]ERROR:[/] [red]MetaMap is not initialized.[/] Please try running `initialize()` first."
            )
            sys.exit(1)

        input_command = Popen(["echo", text], stdout=PIPE)
        match MMOutputType(output_type):
            case MMOutputType.MMI:
                mm_command = Popen(
                    # metamap, silent, MMI, word sense disambiguation, negation auto on for MMI
                    ["metamap", "--silent", "-N", "-y"],
                    stdin=input_command.stdout,
                    stdout=PIPE,
                )
                output = _run_process_command(mm_command)
                # skip the first line which is command
                return output.splitlines()[1:]
            case MMOutputType.JSON:
                mm_command = Popen(
                    # metamap, silent, JSON (no format), word sense disambiguation, negation
                    ["metamap", "--silent", "--JSONn", "-y", "--negex"],
                    stdin=input_command.stdout,
                    stdout=PIPE,
                )
                output = _run_process_command(mm_command)
                return output
        return None

    @validate_arguments
    def run_many(
        self, texts: list[str], output_type: Literal["mmi", "json"] = "mmi"
    ) -> Iterator[list[str] | str | None]:
        """Runs MetaMap on multiple strings.

        Calls thread_map from tqdm.contrib.concurrent to run MetaMap on multiple strings.

        Returns an iterator that must be consumed.

        Args:
            texts (list[str]): The texts to run MetaMap on.
            output_type (Literal["mmi", "json"], optional): The output type. Defaults to "mmi".

        Returns:
            Iterator[list[str] | str | None]: An iterator that must be consumed. The type of the output depends on the `output_type` argument.

        Example:
            ```python
            >>> from cabinet.umls_drawer import MetaMap
            >>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
            >>> mm.initialize()
            >>> results = mm.run_many(texts=["I have a headache.", "I have a fever."])
            >>> for result in results:
                    print(result)
            ```
        """
        # local import to avoid exposing function and since only used here for now
        from tqdm.contrib.concurrent import thread_map

        # allow default selection of max-workers
        return thread_map(self.run, texts)
~~~

`initialize()`

Initialize MetaMap.

This function must be run to start the MetaMap servers. It will check if the servers are already running and if not, it will start them.

Source code in `src/cabinet/umls_drawer/metamap_ner.py`

```python
def initialize(self) -> None:
    """Initialize MetaMap.

    This function must be run to start the MetaMap servers.
    It will check if the servers are already running and if not, it will start them.
    """
    # use generic "metamap" so supports any version
    # but make check here that not older than 2016
    # if older than 2016v2, raise error
    # check initialization cache-file- to see if already initialized
    if _check_metamap_servers_status() is True:
        c.print(
            "[green bold]INFO:[/] [green]MetaMap servers are already running.[/]"
        )
        self._initialized = True
        return None
    else:
        c.print(
            "[yellow bold]WARNING:[/] [yellow]MetaMap servers are not running. Starting servers now.[/]"
        )
        skrmed_server = self.metamap_location / "bin" / "skrmedpostctl"
        wsd_server = self.metamap_location / "bin" / "wsdserverctl"
        os.system(skrmed_server.as_posix() + " start")
        os.system(wsd_server.as_posix() + " start")
        with c.status(
            "Waiting for servers...", spinner="dots", spinner_style="yellow"
        ):
            time.sleep(60)
        if _check_metamap_servers_status() is True:
            c.print(
                "[green bold]INFO:[/] [green]MetaMap servers are now running.[/]"
            )
            self._initialized = True
            return None
        else:
            c.print(
                "[red bold]ERROR:[/] [red]MetaMap servers failed to start.[/] Please check the MetaMap installation and try again."
            )
            return None
```

`run(text, output_type='mmi')`

Run MetaMap on a text string.

This will return None if no results were found, otherwise the return type will match the output_type argument. Future work will include returning a dataclass for each result, but for now: MMOutputType.JSON -> str (the json data itself) MMOutputType.MMI -> list[str] (each line of the MMI output)

Parameters:

Name	Type	Description	Default
`text`	`str`	The text to run MetaMap on.	*required*
`output_type`	`Literal['mmi', 'json']`	The output type. Defaults to "mmi".	`'mmi'`

Returns:

Type	Description
`list[str] | str | None`	list[str] | str | None: The output of MetaMap. The type of the output depends on the `output_type` argument.

Example

```
>>> from cabinet.umls_drawer import MetaMap, MMOutputType
>>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
>>> mm.initialize()
>>> results = mm.run(text="I have a headache.", output_type=MMOutputType.MMI)
>>> print(results)
```

Source code in `src/cabinet/umls_drawer/metamap_ner.py`

~~~python
@validate_arguments
def run(
    self, text: str, output_type: Literal["mmi", "json"] = "mmi"
) -> list[str] | str | None:
    """Run MetaMap on a text string.

    This will return None if no results were found, otherwise the return type will
    match the output_type argument. Future work will include returning a dataclass
    for each result, but for now:
        MMOutputType.JSON -> str  (the json data itself)
        MMOutputType.MMI -> list[str] (each line of the MMI output)

    Args:
        text (str): The text to run MetaMap on.
        output_type (Literal["mmi", "json"], optional): The output type. Defaults to "mmi".

    Returns:
        list[str] | str | None: The output of MetaMap. The type of the output depends on the `output_type` argument.

    Example:
        ```python
        >>> from cabinet.umls_drawer import MetaMap, MMOutputType
        >>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
        >>> mm.initialize()
        >>> results = mm.run(text="I have a headache.", output_type=MMOutputType.MMI)
        >>> print(results)
        ~~~
    """
    if self._initialized is False:
        c.print(
            "[red bold]ERROR:[/] [red]MetaMap is not initialized.[/] Please try running `initialize()` first."
)
        sys.exit(1)

    input_command = Popen(["echo", text], stdout=PIPE)
    match MMOutputType(output_type):
        case MMOutputType.MMI:
            mm_command = Popen(
                # metamap, silent, MMI, word sense disambiguation, negation auto on for MMI
                ["metamap", "--silent", "-N", "-y"],
                stdin=input_command.stdout,
                stdout=PIPE,
)
            output = _run_process_command(mm_command)
            # skip the first line which is command
            return output.splitlines()[1:]
        case MMOutputType.JSON:
            mm_command = Popen(
                # metamap, silent, JSON (no format), word sense disambiguation, negation
                ["metamap", "--silent", "--JSONn", "-y", "--negex"],
                stdin=input_command.stdout,
                stdout=PIPE,
)
            output = _run_process_command(mm_command)
            return output
    return None
```

`run_many(texts, output_type='mmi')`

Runs MetaMap on multiple strings.

Calls thread_map from tqdm.contrib.concurrent to run MetaMap on multiple strings.

Returns an iterator that must be consumed.

Parameters:

Name	Type	Description	Default
`texts`	`list[str]`	The texts to run MetaMap on.	*required*
`output_type`	`Literal['mmi', 'json']`	The output type. Defaults to "mmi".	`'mmi'`
```

**Returns:**

| Type | Description |
|---|---|
| `Iterator[list[str] \| str \| None]` | Iterator[list[str] \| str \| None]: An iterator that must be consumed. The type of the output depends on the `output_type` argument. |

**Example**

```
>>> from cabinet.umls_drawer import MetaMap
>>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
>>> mm.initialize()
>>> results = mm.run_many(texts=["I have a headache.", "I have a fever."])
>>> for result in results:
    print(result)
```

Source code in `src/cabinet/umls_drawer/metamap_ner.py`

~~~
@validate_arguments
def run_many(
    self, texts: list[str], output_type: Literal["mmi", "json"] = "mmi"
) -> Iterator[list[str] | str | None]:
    """Runs MetaMap on multiple strings.

    Calls thread_map from tqdm.contrib.concurrent to run MetaMap on multiple strings.

    Returns an iterator that must be consumed.

    Args:
        texts (list[str]): The texts to run MetaMap on.
        output_type (Literal["mmi", "json"], optional): The output type. Defaults to "mmi".

    Returns:
        Iterator[list[str] | str | None]: An iterator that must be consumed. The type of the output depends on the `output_type` argument.

    Example:
        ```python
        >>> from cabinet.umls_drawer import MetaMap
        >>> mm = MetaMap(metamap_location="/Users/username/metamap/public_mm")
        >>> mm.initialize()
        >>> results = mm.run_many(texts=["I have a headache.", "I have a fever."])
        >>> for result in results:
            print(result)
        ```
    """
    # local import to avoid exposing function and since only used here for now
    from tqdm.contrib.concurrent import thread_map

    # allow default selection of max-workers
    return thread_map(self.run, texts)
~~~

2.3 Cleaning Drawer

This drawer exists for cleaning functions.

It's a bit of a catch-all for functions that don't fit in any other drawer.

This module contains code for typical data normalization tasks.

We generally enforce 'type-saftey' by using pydantic's validate_arguments decorator and other pydantic types.

2.3.1 `categorize_age(age)`

Categorize an age into a string.

Parameters:

Name	Type	Description	Default
`age`	`PositiveInt`	Age to categorize. Must be a positive integer.	*required*

Returns:

Name	Type	Description
`str`	`str`	Categorized age.

Examples:

```
>>> categorize_age(10)
'<18'
>>> categorize_age(18)
'18-25'
>>> # a general example using pandas
>>> df['age'].apply(categorize_age)
pandas.Series(['<18', '18-25', '26-35', '36-45', '46-55', '56-65', '65+'])
```

Source code in `src/cabinet/cleaning_drawer/normalize.py`

```python
@validate_arguments
def categorize_age(age: PositiveInt) -> str:
    """Categorize an age into a string.

    Args:
        age (PositiveInt): Age to categorize. Must be a positive integer.

    Returns:
        str: Categorized age.

    Examples:
        >>> categorize_age(10)
        '<18'
        >>> categorize_age(18)
        '18-25'
        >>> # a general example using pandas
        >>> df['age'].apply(categorize_age)
        pandas.Series(['<18', '18-25', '26-35', '36-45', '46-55', '56-65', '65+'])
    """
    if age < 18:
        return "<18"
    elif age <= 25:
        return "18-25"
    elif age <= 35:
        return "26-35"
    elif age <= 45:
        return "36-45"
    elif age <= 55:
        return "46-55"
    elif age <= 65:
        return "56-65"
    else:
        return ">65"
```