My Document

Author Name

July 27, 2024

1 Module `metagpt`

1.1 Sub-modules

- metagpt.distorters
- metagpt.evaluators
- metagpt.experiments
- metagpt.predictors
- metagpt.utils

2 Namespace `metagpt.distorters`

2.1 Sub-modules

- metagpt.distorters.distorter_template

3 Module `metagpt.distorters.distorter_template`

This module contains the DistorterTemplate class, which is responsible for distorting well-formed OME XML into a modified key-value representation. The distortion process can include converting XML to key-value pairs, shuffling the order of entries, and renaming keys to similar words.

3.1 Classes

3.1.1 Class `DistorterTemplate`

```
class DistorterTemplate
```

A class for distorting OME XML into modified key-value representations.

The distorter takes well-formed OME XML as input and returns a "distorted" key-value version of it. Distortion can include: - OME XML to key-value conversion - Shuffling of the order of entries - Renaming keys to similar words

Methods

Method `distort`

```
def distort(
    self,
    ome_xml: str,
    out_path: str,
    should_pred: str = 'maybe'
) -> Optional[Dict[str, Any]]
```

Distort the OME XML.

Args ——= **`ome_xml`** : str : The input OME XML string.

`out_path : str` The path where the distorted data will be saved.

should_pred : str Whether to predict new data or use existing data. Options are “yes”, “no”, or “maybe”.

Returns ——= Optional[Dict[str, Any]] : The distorted metadata, or None if no data is available.

Method `extract_unique_keys`

```
def extract_unique_keys(
    self,
    metadata: Dict[str, Any]
) -> List[str]
```

Extract all unique key names from a dictionary, including nested structures, without full paths or indices.
Args ——= **metadata** : Dict[str, Any] : The dictionary containing metadata.
Returns ——= List[str] : A list of unique key names.

Method `gen_mapping`

```
def gen_mapping(
    self,
    dict_meta: Dict[str, Any]
) -> Dict[str, str]
```

Rename the keys in the OME XML to similar words using a GPT model.
Args ——= **dict_meta** : Dict[str, Any] : The input dictionary.
Returns ——= Dict[str, str] : A dictionary mapping original keys to new keys.

Method `isolate_keys`

```
def isolate_keys(
    self,
    dict_meta: Dict[str, Any]
) -> Dict[str, None]
```

Isolate the keys in the OME XML.
Args ——= **dict_meta** : Dict[str, Any] : The input dictionary.
Returns ——= Dict[str, None] : A dictionary with the same keys as the input, but all values set to None.

Method `load_fake_data`

```
def load_fake_data(
    self,
    path: str
) -> Optional[Dict[str, Any]]
```

Load the fake data from a file.
Args ——= **path** : str : The file path from which to load the data.
Returns ——= Optional[Dict[str, Any]] : The loaded data, or None if the file doesn’t exist.

Method `modify_metadata_structure`

```
def modify_metadata_structure(
    self,
    metadata: Dict[str, Any],
    operations: Optional[List[<built-in function callable>]] = None,
    probability: float = 0.3
) -> Dict[str, Any]
```

Modify the structure of a metadata dictionary systematically and randomly.
Args ——= **metadata** : Dict[str, Any] : The original metadata dictionary.

`operations : Optional[List[callable]]` List of operations to perform. If None, all operations are used.
`probability : float` Probability of applying an operation to each element (0.0 to 1.0).

Returns ——= Dict[str, Any] : A new dictionary with modified structure.

Method `pred`

```
def pred(
    self,
    ome_xml: str,
    out_path: str
) -> Dict[str, Any]
```

Predict the distorted data.
Args ——= `ome_xml` : str : The input OME XML string.

`out_path : str` The path where the distorted data will be saved.

Returns ——= Dict[str, Any] : The distorted metadata.

Method `rename_metadata_keys`

```
def rename_metadata_keys(
    self,
    metadata: Dict[str, Any],
    key_mapping: Dict[str, str]
) -> Dict[str, Any]
```

Rename keys in a metadata dictionary based on a provided mapping.
Args ——= `metadata` : Dict[str, Any] : The original metadata dictionary.

`key_mapping : Dict[str, str]` A dictionary mapping original key names to new key names.

Returns ——= Dict[str, Any] : A new dictionary with renamed keys.

Method `save_fake_data`

```
def save_fake_data(
    self,
    fake_data: Dict[str, Any],
    path: str
) -> None
```

Save the fake data to a file.
Args ——= `fake_data` : Dict[str, Any] : The data to be saved.

`path : str` The file path where the data will be saved.

Method `shuffle_order`

```
def shuffle_order(
    self,
    dict_meta: Dict[str, Any]
) -> Dict[str, Any]
```

Shuffle the order of the keys in the OME XML.
Args ——= `dict_meta` : Dict[str, Any] : The input dictionary.
Returns ——= Dict[str, Any] : A new dictionary with shuffled keys.

Method `xml_to_key_value`

```
def xml_to_key_value(
    self,
    ome_xml: str
) -> Dict[str, Any]
```

Convert the OME XML to key-value pairs.
Args ——= **`ome_xml`** : str : The input OME XML string.
Returns ——= Dict[str, Any] : A dictionary representation of the XML.
Raises ——= Exception : If parsing fails.

4 Namespace `metagpt.evaluators`

4.1 Sub-modules

- metagpt.evaluators.evaluator_template

5 Module `metagpt.evaluators.evaluator_template`

This module contains the EvaluatorTemplate class, which is responsible for evaluating the performance of OME XML generation models by calculating the edit distance between the ground truth and the prediction.

The class provides various methods for data analysis and visualization, including edit distance calculations, path analysis, and performance comparisons across different methods and image formats.

5.1 Classes

5.1.1 Class `EvaluatorTemplate`

```
class EvaluatorTemplate(
    schema: Optional[str] = None,
    dataset: Optional[metagpt.utils.DataClasses.Dataset] = None,
    out_path: Optional[str] = None
)
```

This class evaluates the performance of an OME XML generation model by calculating the edit distance between the ground truth and the prediction.

Reference: `https://github.com/timtadh/zhang-shasha`

Initialize the EvaluatorTemplate.

Args ——= **`schema`** : Optional[str] : The schema to use for evaluation.

`dataset` : Optional[Dataset] The dataset to evaluate.
`out_path` : Optional[str] The output path for saving results.

Methods

Method `attempts_paths_plt`

```
def attempts_paths_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot number of attempts against number of paths.

Method `format_counts_plt`

```
def format_counts_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot counts of samples by image format.

Method `format_method_plt`

```
def format_method_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot edit distance by method and image format.

Method `generate_results_report`

```
def generate_results_report(
    self,
    figure_paths: List[str],
    context: str
) -> Optional[str]
```

Generate a results report based on the provided figures and context.
Args ——= **`figure_paths`** : List[str] : Paths to the figure images.

`context : str` Context information for the report.

Returns ——= Optional[str] : The generated report, or None if an error occurred.

Method `get_graph`

```
def get_graph(
    self,
    xml_root: ome_types._autogenerated.ome_2016_06.ome.OME,
    root: Optional[zss.simple_tree.Node] = None
) -> zss.simple_tree.Node
```

Get the graph representation of an OME XML tree as a zss Node.
Args ——= **`xml_root`** : OME : The root of the XML tree.

`root : Optional[Node]` The root node of the graph (used for recursion).

Returns ——= Node : The root node of the graph representation.

Method `json_to_pygram`

```
def json_to_pygram(
    self,
    json_data: Dict[str, Any]
) -> Any
```

Convert a JSON structure to a pygram tree.
Args ——= **`json_data`** : Dict[str, Any] : The JSON data to convert.
Returns ——= Any : The root node of the pygram tree.

Method `method_attempts_plt`

```
def method_attempts_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot number of attempts by method.

Method `method_cost_plt`

```
def method_cost_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot method cost.

Method `method_edit_distance_no_annot_plt`

```
def method_edit_distance_no_annot_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot method edit distance without annotations.

Method `method_edit_distance_only_annot_plt`

```
def method_edit_distance_only_annot_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot method edit distance for annotations only.

Method `method_edit_distance_plt`

```
def method_edit_distance_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot method edit distance.

Method `method_time_plt`

```
def method_time_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot method prediction time.

Method `n_paths_cost_plt`

```
def n_paths_cost_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot number of paths against cost.

Method `n_paths_method_plt`

```
def n_paths_method_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot number of paths per method.

Method `n_paths_time_plt`

```
def n_paths_time_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot number of paths against prediction time.

Method `path_df`

```
def path_df(
    self
) -> pandas.core.frame.DataFrame
```

Create a DataFrame with paths as Index and samples as Columns.
Returns —= pd.DataFrame : DataFrame with path information.

Method `path_difference`

```
def path_difference(
    self,
    xml_a: ome_types._autogenerated.ome_2016_06.ome.OME,
    xml_b: ome_types._autogenerated.ome_2016_06.ome.OME
) -> int
```

Calculate the length of the difference between the path sets in two XML trees.
Args —= **`xml_a`** : OME : The first XML tree.

`xml_b` : OME The second XML tree.

Returns —= int : The length of the difference between the path sets.

Method `paths_annotation_stacked_plt`

```
def paths_annotation_stacked_plt(
    self,
    df_sample: pandas.core.frame.DataFrame
)
```

Plot stacked bar chart of paths and annotations.

Method `plot_price_per_token`

```
def plot_price_per_token(
    self
)
```

Plot price per token for different models.

Method `pygram_edit_distance`

```
def pygram_edit_distance(
    self,
    xml_a: ome_types._autogenerated.ome_2016_06.ome.OME,
    xml_b: ome_types._autogenerated.ome_2016_06.ome.OME
) -> float
```

Calculate the edit distance between two XML trees using pygram.
Args ——= **`xml_a`** : OME : The first XML tree.

`xml_b` : OME The second XML tree.

Returns ——= float : The edit distance between the two trees.

Method `report`

```
def report(
    self
)
```

Generate and write an evaluation report to a file.

Method `sample_df`

```
def sample_df(
    self,
    df_paths: pandas.core.frame.DataFrame
) -> pandas.core.frame.DataFrame
```

Create a DataFrame with samples as Index and properties as Columns.
Args ——= **`df_paths`** : pd.DataFrame : DataFrame containing path information.
Returns ——= pd.DataFrame : DataFrame with sample properties.

Method `zss_edit_distance`

```
def zss_edit_distance(
    self,
    xml_a: ome_types._autogenerated.ome_2016_06.ome.OME,
    xml_b: ome_types._autogenerated.ome_2016_06.ome.OME
) -> int
```

Calculate the Zhang-Shasha edit distance between two XML trees.
Args ——= **`xml_a`** : OME : The first XML tree.

`xml_b` : OME The second XML tree.

Returns ——= int : The edit distance between the two trees.

6 Namespace `metagpt.experiments`

6.1 Sub-modules

- metagpt.experiments.experiment_template

7 Module `metagpt.experiments.experiment_template`

This module contains the ExperimentTemplate class, which defines an experiment object that can be used to run experiments. The experiment defines the dataset, predictors, evaluators, and other parameters necessary for running experiments on OME XML metadata.

7.1 Classes

7.1.1 Class `ExperimentTemplate`

```
class ExperimentTemplate
```

The ExperimentTemplate class defines an experiment object that can be used to run experiments. It encapsulates the dataset, predictors, evaluators, and other experiment parameters.

Initialize the ExperimentTemplate with default values.

Methods

Method `run`

```
def run(
    self
) -> None
```

Run the experiment. This method processes each image in the data_paths, generates metadata, and runs predictors and evaluators.

8 Module `metagpt.predictors`

8.1 Sub-modules

- metagpt.predictors.predictor_distorter
- metagpt.predictors.predictor_network
- metagpt.predictors.predictor_network_annotator
- metagpt.predictors.predictor_seperator
- metagpt.predictors.predictor_simple
- metagpt.predictors.predictor_simple_annotator
- metagpt.predictors.predictor_state
- metagpt.predictors.predictor_state_tree
- metagpt.predictors.predictor_template

9 Module `metagpt.predictors.predictor_distorter`

This module contains the PredictorDistorter class, which is responsible for inventing new metadata syntax for microscopy images based on existing metadata.

9.1 Classes

9.1.1 Class `PredictorDistorter`

```
class PredictorDistorter(
    raw_meta: str
)
```

A predictor class for inventing new metadata syntax for microscopy images.

This class takes existing metadata and translates it into a new syntax, maintaining the original structure and values but changing the keys.

Initialize the PredictorDistorter.

Args ——= **`raw_meta`** : str : The raw metadata to be translated.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `out_new_meta` Helper class to define the output structure of the assistant.

Methods

Method `init_assistant`

```
def init_assistant(
    self
) -> None
```

Initialize the OpenAI assistant.

Method `init_run`

```
def init_run(
    self
) -> None
```

Initialize and monitor the run of the assistant.

Method `predict`

```
def predict(
    self
) -> Optional[Dict[str, Any]]
```

Predict the new metadata syntax based on the raw metadata.
Returns ——= Optional[Dict[str, Any]] : The predicted new metadata syntax, or None if prediction fails.

10 Module `metagpt.predictors.predictor_network`

This module contains the PredictorNetwork class, which uses a network of predictors to process, annotate, and merge metadata for microscopy images.

10.1 Classes

10.1.1 Class `PredictorNetwork`

```
class PredictorNetwork(
    raw_meta: str
)
```

A predictor class that uses a network of predictors to process and annotate metadata.
This predictor approach uses three assistants: 1. A separator to split the raw metadata into already contained and new metadata. 2. An annotator to predict structured annotations from the new metadata. 3. A simple predictor to process the remaining metadata.
Initialize the PredictorNetwork.
Args ——= **`raw_meta`** : str : The raw metadata to be processed and annotated.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[str], float, int]
```

Predict structured annotations based on the raw metadata.

This method uses three predictors in sequence: 1. PredictorSeperator to split the metadata. 2. PredictorSimpleAnnotation to generate annotations. 3. PredictorSimple to process the remaining metadata.

Returns ——= Tuple[Optional[str], float, int] :
- The merged XML annotation (or None if prediction fails) - The total cost of the prediction - The total number of attempts made

11 Module `metagpt.predictors.predictor_network_annotator`

This module contains the PredictorNetworkAnnotation class, which uses a network of predictors to process and annotate metadata for microscopy images.

11.1 Classes

11.1.1 Class `PredictorNetworkAnnotation`

```
class PredictorNetworkAnnotation(
    raw_meta: str
)
```

A predictor class that uses two assistants to process and annotate metadata.

This predictor approach uses two assistants: 1. A separator to split the raw metadata into already contained and new metadata. 2. An annotator to predict structured annotations from the new metadata.

Initialize the PredictorNetworkAnnotation.

Args ——= **`raw_meta`** : str : The raw metadata to be processed and annotated.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[Any], float, float]
```

Predict structured annotations based on the raw metadata.

This method uses two predictors in sequence: 1. PredictorSeperator to split the metadata. 2. PredictorSimpleAnnotation to generate annotations.

Returns ——= Tuple[Optional[Any], float, float] :
- The predicted annotations (or None if prediction fails) - The total cost of the prediction - The total number of attempts made

12 Module `metagpt.predictors.predictor_seperator`

This module contains the PredictorSeperator class, which is responsible for separating raw metadata into structured annotations and OME properties.

12.1 Classes

12.1.1 Class `PredictorSeperator`

```
class PredictorSeperator(
    raw_meta: str
)
```

A predictor class that separates raw metadata into structured annotations and OME properties using OpenAI's language model and vector embeddings.

Initialize the PredictorSeperator.

Args ——= **`raw_meta`** : str : The raw metadata to be processed.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `SepOutputTool` This tool automatically formats and structures the metadata in the appropriate way.

Methods

Method `init_assistant`

```
def init_assistant(
    self
) -> None
```

Initialize the OpenAI assistant.

Method `init_run`

```
def init_run(
    self
) -> None
```

Initialize and monitor the run of the assistant.

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[Tuple[Dict[str, str], Dict[str, str]]], float, int]
```

Predict the separation of raw metadata into structured annotations and OME properties.

Returns ——= Tuple[Optional[Tuple[Dict[str, str], Dict[str, str]]], float, int]: - A tuple containing two dictionaries (annotation_properties, ome_properties), or None if prediction fails - The cost of the prediction - The number of attempts made

13 Module `metagpt.predictors.predictor_simple`

This module contains the PredictorSimple class, which is responsible for predicting well-formed OME XML from raw metadata using OpenAI's language model.

13.1 Classes

13.1.1 Class `PredictorSimple`

```
class PredictorSimple(
    raw_meta: str
)
```

A predictor class that generates well-formed OME XML from raw metadata using OpenAI's language model and vector embeddings.

Initialize the PredictorSimple.

Args ——= **`raw_meta`** : str : The raw metadata to be processed.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `OMEXMLResponse` The response containing the well-formed OME XML.

Methods

Method `init_assistant`

```
def init_assistant(
    self
) -> None
```

Initialize the OpenAI assistant.

Method `init_run`

```
def init_run(
    self
) -> None
```

Initialize and monitor the run of the assistant.

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[str], float, int]
```

Predict well-formed OME XML based on the raw metadata.

Returns ——= Tuple[Optional[str], float, int]: - The predicted OME XML as a string, or None if prediction fails - The cost of the prediction - The number of attempts made

14 Module `metagpt.predictors.predictor_simple_annotator`

This module contains the PredictorSimpleAnnotation class, which is responsible for predicting structured annotations for the OME model from raw metadata.

14.1 Classes

14.1.1 Class `PredictorSimpleAnnotation`

```
class PredictorSimpleAnnotation(
    raw_meta: str
)
```

A predictor class that generates structured annotations for the OME model from raw metadata using OpenAI's language model.

Initialize the PredictorSimpleAnnotation.

Args ——= **`raw_meta`** : str : The raw metadata to be processed.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `XMLAnnotationFunction` The function call to hand in the structured annotations to the OME XML.

Methods

Method `init_assistant`

```
def init_assistant(
    self
) -> None
```

Initialize the OpenAI assistant.

Method `init_run`

```
def init_run(
    self
) -> None
```

Initialize and monitor the run of the assistant.

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[Dict[str, Any]], float, int]
```

Predict structured annotations based on the raw metadata.

Returns ——= Tuple[Optional[Dict[str, Any]], float, int]: - The predicted annotations as a dictionary, or None if prediction fails - The cost of the prediction - The number of attempts made

15 Module `metagpt.predictors.predictor_state`

This module contains the PredictorState class, which is responsible for predicting and updating OME metadata using JSON patches and OpenAI's language model.

15.1 Classes

15.1.1 Class `AddReplaceTestOperation`

```
class AddReplaceTestOperation(
    **data: Any
)
```

Model for Add, Replace, and Test operations in JSON Patch.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['add', 'replace', 'test']`

Variable `path` Type: `str`

Variable `value` Type: `Any`

15.1.2 Class `JsonPatch`

```
class JsonPatch(
    **data: Any
)
```

Model for a complete JSON Patch.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `root` Type: `List[Union[metagpt.predictors.predictor_state.AddReplaceTestOperation, metagpt`

15.1.3 Class `MoveCopyOperation`

```
class MoveCopyOperation(
    **data: Any
)
```

Model for Move and Copy operations in JSON Patch.
Create a new model by parsing and validating input data from keyword arguments.
Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.
self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `from_` Type: `str`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['move', 'copy']`

Variable `path` Type: `str`

15.1.4 Class `PredictorState`

```
class PredictorState(
    raw_meta: str,
    state: pydantic.main.BaseModel = None
)
```

A predictor class that generates and applies JSON patches to update OME metadata using OpenAI's language model.
Initialize the PredictorState.
Args ——= **`raw_meta`** : str : The raw metadata to process.

`state` : BaseModel, optional The initial state. Defaults to OME().

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `init_assistant`

```
def init_assistant(
    self
) -> None
```

Initialize the OpenAI assistant.

Method `init_run`

```
def init_run(
    self
) -> None
```

Initialize and monitor the run of the assistant.

Method `predict`

```
def predict(
    self,
    indent: Optional[int] = 0
) -> tuple[typing.Optional[str], float, int]
```

Predict OME metadata and apply JSON patches to update the state.
Args —= **`indent`** : Optional[int] : Indentation for logging. Defaults to 0.
Returns —= tuple[Optional[str], float, int] : The updated OME XML, cost, and number of attempts.

15.1.5 Class `RemoveOperation`

```
class RemoveOperation(
    **data: Any
)
```

Model for Remove operation in JSON Patch.
Create a new model by parsing and validating input data from keyword arguments.
Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.
self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['remove']`

Variable `path` Type: `str`

15.1.6 Class `update_json_state`

```
class update_json_state(
    **data: Any
)
```

Model for updating the state of the predictor from a list of JSON patches.
Create a new model by parsing and validating input data from keyword arguments.
Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.
self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `json_patches` Type: `Optional[List[metagpt.predictors.predictor_state.JsonPatch]]`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

16 Module `metagpt.predictors.predictor_state_tree`

This module contains the PredictorStateTree class and related components for predicting OME metadata using a tree-based approach with OpenAI's language model.

16.1 Functions

16.1.1 Function `create_instance`

```
def create_instance(
    instance: Type[pydantic.main.BaseModel],
    obj_dict: Dict[str, Any]
) -> Optional[pydantic.main.BaseModel]
```

Create an instance of a Pydantic model, filling it with child objects.
Args ——= **`instance`** : Type[BaseModel] : The Pydantic model class to instantiate.

`obj_dict` : Dict[str, Any] Dictionary of child objects to include.

Returns ——= Optional[BaseModel] : The instantiated model, or None if instantiation fails.

16.2 Classes

16.2.1 Class `PredictorStateTree`

```
class PredictorStateTree(
    raw_meta: str,
    model: Type[pydantic.main.BaseModel] = None
)
```

A predictor class that uses a tree-based approach to predict OME metadata.
Initialize the PredictorStateTree.
Args ——= **`raw_meta`** : str : The raw metadata to process.

`model` : Type[BaseModel], optional The root model to use. Defaults to OME.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `build_tree`

```
def build_tree(
    self,
    root_model: Type[pydantic.main.BaseModel]
) -> metagpt.predictors.predictor_state_tree.TreeNode
```

Build the complete dependency tree starting from the root model.
Args —= **`root_model`** : Type[BaseModel] : The root model to start building the tree from.
Returns —= TreeNode : The root node of the built tree.

Method `collect_dependencies`

```
def collect_dependencies(
    self,
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    collected: Dict[str, Type[pydantic.main.BaseModel]]
) -> None
```

Collect all dependent models for a given model.
Args —= **`model`** : Type[BaseModel] : The model to collect dependencies for.

`known_models` **: Dict[str, Type[BaseModel]]** Dictionary of known models.
`collected` **: Dict[str, Type[BaseModel]]** Dictionary to store collected models.

Method `create_dependency_tree`

```
def create_dependency_tree(
    self,
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    visited: Set[str]
) -> metagpt.predictors.predictor_state_tree.TreeNode
```

Create a dependency tree for a given model.
Args —= **`model`** : Type[BaseModel] : The model to create a tree for.

`known_models` **: Dict[str, Type[BaseModel]]** Dictionary of known models.
`visited` **: Set[str]** Set of visited model names.

Returns —= TreeNode : The root node of the created tree.

Method `predict`

```
def predict(
    self
) -> Tuple[Optional[pydantic.main.BaseModel], Optional[float], Optional[int]]
```

Predict the OME metadata using the dependency tree.
Returns —= Tuple[Optional[BaseModel], Optional[float], Optional[int]]: The predicted metadata, cost (None for this implementation), and attempts (None for this implementation).

Method `print_tree`

```
def print_tree(
    self,
    node: Optional[metagpt.predictors.predictor_state_tree.TreeNode] = None,
    indent: str = ''
) -> None
```

Print the structure of the dependency tree.
Args —= **`node`** : Optional[TreeNode] : The node to start printing from. If None, starts from the root.

`indent` **: str** The current indentation string.

16.2.2 Class `TreeNode`

```
class TreeNode(
    model: Type[pydantic.main.BaseModel]
)
```

Represents a node in the dependency tree for OME metadata prediction.

Methods

Method `add_child`

```
def add_child(
    self,
    child: TreeNode
) -> None
```

Add a child node to this node.

Method `instantiate_model`

```
def instantiate_model(
    self,
    child_objects: Dict[str, Any]
) -> pydantic.main.BaseModel
```

Instantiate the model for this node, including child objects.
Args —— = **`child_objects`** : Dict[str, Any] : Dictionary of child objects to include.
Returns —— = BaseModel : The instantiated model, or a MaybeModel if instantiation fails.

Method `predict_meta`

```
def predict_meta(
    self,
    raw_meta: str,
    indent: int = 0
) -> Tuple[Optional[pydantic.main.BaseModel], float, int]
```

Predict metadata for this node and its children.
Args —— = **`raw_meta`** : str : The raw metadata to process.

`indent : int` The indentation level for printing (used for debugging).

Returns —— = Tuple[Optional[BaseModel], float, int] : The predicted state, total cost, and total attempts.

Method `required_fields`

```
def required_fields(
    self,
    model: Type[pydantic.main.BaseModel],
    recursive: bool = False
) -> collections.abc.Iterator[str]
```

Get all required fields of a Pydantic model, optionally including nested models.
Args —— = **`model`** : Type[BaseModel] : The Pydantic model to inspect.

`recursive : bool` Whether to include fields from nested models.

Yields —— = str : Names of required fields.

17 Module `metagpt.predictors.predictor_template`

This module contains the PredictorTemplate class, which serves as a base class for creating predictors that utilize OpenAI's API to generate OME XML from raw metadata.

17.1 Classes

17.1.1 Class `PredictorTemplate`

```
class PredictorTemplate
```

A template for creating a new predictor. A predictor utilizes one or several assistants to predict the OME XML from the raw metadata.

Descendants

- metagpt.predictors.predictor_distorter.PredictorDistorter
- metagpt.predictors.predictor_network.PredictorNetwork
- metagpt.predictors.predictor_network_annotator.PredictorNetworkAnnotation
- metagpt.predictors.predictor_seperator.PredictorSeperator
- metagpt.predictors.predictor_simple.PredictorSimple
- metagpt.predictors.predictor_simple_annotator.PredictorSimpleAnnotation
- metagpt.predictors.predictor_state.PredictorState
- metagpt.predictors.predictor_state_tree.PredictorStateTree

Methods

Method `add_attempts`

```
def add_attempts(
    self,
    i: float = 1
) -> None
```

Add an attempt to the attempt counter. Normalized by the number of assistants.
Args ——= **`i`** : float : The number of attempts to add. Defaults to 1.

Method `clean_assistants`

```
def clean_assistants(
    self
) -> None
```

Clean up the assistants, threads, and vector stores.

Method `export_ome_xml`

```
def export_ome_xml(
    self
) -> None
```

Export the OME XML response to a file.

Method `generate_message`

```
def generate_message(
    self,
    msg: Optional[str] = None
) -> Any
```

Generate a new message in the thread.
Args ——= **`msg`** : Optional[str] : The message content. If None, uses self.message.
Returns ——= Any : The created message object.

Method `get_cost`

```
def get_cost(
    self
) -> Optional[float]
```

Calculate the cost of the prediction based on token usage.
Returns —= Optional[float] : The calculated cost, or None if there was an error.

Method `get_response`

```
def get_response(
    self
) -> None
```

Predict the OME XML from the raw metadata and handle the response.

Method `init_thread`

```
def init_thread(
    self
) -> None
```

Initialize a new thread with the initial prompt and message.

Method `init_vector_store`

```
def init_vector_store(
    self
) -> None
```

Initialize the vector store and upload file batches.

Method `predict`

```
def predict(
    self
) -> Dict[str, Any]
```

Predict the OME XML from the raw metadata.
Returns —= Dict[str, Any] : The predicted OME XML and related information.
Raises —= NotImplementedError : This method should be implemented by subclasses.

Method `read_ome_as_string`

```
def read_ome_as_string(
    self,
    path: str
) -> str
```

Read the OME XML as a string from a file.
Args —= **`path`** : str : The path to the OME XML file.
Returns —= str : The contents of the OME XML file as a string.

Method `read_ome_as_xml`

```
def read_ome_as_xml(
    self,
    path: str
) -> str
```

Read the OME XML file and return the root element as a string.
Args —= **`path`** : str : The path to the OME XML file.
Returns —= str : The root element of the OME XML as a string.

Method `read_raw_metadata`

```
def read_raw_metadata(
    self
) -> str
```

Read the raw metadata from the file.
Returns ——= str : The contents of the raw metadata file.
Raises ——= FileNotFoundError : If the path_to_raw_metadata is not set or the file doesn't exist.

Method `subdivide_raw_metadata`

```
def subdivide_raw_metadata(
    self
) -> None
```

Subdivide the raw metadata into appropriate chunks.

Method `validate`

```
def validate(
    self,
    ome_xml: str
) -> Optional[Exception]
```

Validate the OME XML against the OME XSD.
Args ——= **`ome_xml`** : str : The OME XML string to validate.
Returns ——= Optional[Exception] : The exception if validation fails, None otherwise.

18 Module `metagpt.utils`

18.1 Sub-modules

- metagpt.utils.BioformatsReader
- metagpt.utils.DataClasses
- metagpt.utils.utils

19 Module `metagpt.utils.BioformatsReader`

This module implements functions to read proprietary images and return their metadata in OME-XML format and as raw metadata key-value pairs using Bio-Formats.

19.1 Functions

19.1.1 Function `get_omexml_metadata`

```
def get_omexml_metadata(
    path: Optional[str] = None,
    url: Optional[str] = None
) -> str
```

Read the OME metadata from a file using Bio-Formats.
Args ——= **`path`** : Optional[str] : Path to the file. Defaults to None.

`url` : **Optional**[**str**] URL of the file. Defaults to None.

Returns ——= str : The metadata as XML.
Raises ——= ValueError : If neither path nor url is provided.

19.1.2 Function `get_raw_metadata`

```
def get_raw_metadata(
    path: str
) -> Dict[str, str]
```

Read the raw metadata from a file using Bio-Formats.
Args —= **`path`** : str : Path to the file.
Returns —= Dict[str, str] : The metadata as a dictionary.

19.1.3 Function `raw_to_tree`

```
def raw_to_tree(
    raw_metadata: Dict[str, str]
) -> Dict[str, Union[str, Dict]]
```

Convert the raw metadata to a tree structure by separating the key on the "|" character.
Args —= **`raw_metadata`** : Dict[str, str] : The raw metadata dictionary.
Returns —= Dict[str, Union[str, Dict]] : The metadata in a tree structure.

20 Module `metagpt.utils.DataClasses`

Data classes for the metagpt package.

20.1 Classes

20.1.1 Class `Dataset`

```
class Dataset(
    name: str = None,
    samples: dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None)] = Fiel
    cost: Optional[float] = 0,
    time: Optional[float] = 0
)
```

Dataset(name: str = None, samples: dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None)] = FieldInfo(annotation=NoneType, required=False, default_factory=dict), cost: Optional[float] = 0, time: Optional[float] = 0)

Class variables

Variable `cost` Type: `Optional[float]`

Variable `name` Type: `str`

Variable `samples` Type: `dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None]`

Variable `time` Type: `Optional[float]`

Methods

Method `add_sample`

```
def add_sample(
    self,
    sample: metagpt.utils.DataClasses.Sample
)
```

20.1.2 Class `Sample`

```
class Sample(
    format: str,
    attempts: float,
    index: int,
    file_name: str,
    name: str = None,
    metadata_str: str = None,
    method: str = None,
    metadata_xml: ome_types._autogenerated.ome_2016_06.ome.OME = FieldInfo(annotation=NoneType,
    cost: Optional[float] = None,
    paths: Optional[list[str]] = None,
    time: Optional[float] = None,
    gpt_model: Optional[str] = None
)
```

Sample(format: str, attempts: float, index: int, file_name: str, name: str = None, metadata_str: str = None, method: str = None, metadata_xml: ome_types._autogenerated.ome_2016_06.ome.OME = FieldInfo(annotation=NoneType, required=False, default_factory=OME, description='The metadata as an OME object'), cost: Optional[float] = None, paths: Optional[list[str]] = None, time: Optional[float] = None, gpt_model: Optional[str] = None)

Class variables

Variable `attempts` Type: `float`

Variable `cost` Type: `Optional[float]`

Variable `file_name` Type: `str`

Variable `format` Type: `str`

Variable `gpt_model` Type: `Optional[str]`

Variable `index` Type: `int`

Variable `metadata_str` Type: `str`

Variable `metadata_xml` Type: `ome_types._autogenerated.ome_2016_06.ome.OME`

Variable `method` Type: `str`

Variable `name` Type: `str`

Variable `paths` Type: `Optional[list[str]]`

Variable `time` Type: `Optional[float]`

21 Module `metagpt.utils.utils`

This module contains various utility functions and classes for handling OME XML data, JSON operations, and other helper functions used in the MetaGPT project.

21.1 Functions

21.1.1 Function `browse_schema`

```
def browse_schema(
    cls: Type[pydantic.main.BaseModel],
    additional_ignored_keywords: List[str] = [],
    max_depth: int = inf
) -> Dict[str, Any]
```

Browse a schema as jsonschema, with depth control.
Args —= **`cls`** : Type[BaseModel] : The Pydantic model to convert to a schema.

`additional_ignored_keywords` : List[str], optional Additional keywords to ignore in the schema. Defaults to [].

`max_depth` : int, optional Maximum depth of nesting to include in the schema. Defaults to infinity.

Returns —= Dict[str, Any] : A dictionary in the format of OpenAI's schema as jsonschema.

21.1.2 Function `camel_to_snake`

```
def camel_to_snake(
    name: str
) -> str
```

Convert a CamelCase string to snake_case.
Args —= **`name`** : str : The CamelCase string to convert.
Returns —= str : The converted snake_case string.

21.1.3 Function `custom_apply`

```
def custom_apply(
    patch: jsonpatch.JsonPatch,
    data: Dict[str, Any]
) -> Dict[str, Any]
```

Apply the JSON Patch, automatically creating missing nodes.
Args —= **`patch`** : jsonpatch.JsonPatch : The JSON Patch to apply.

`data` : Dict[str, Any] The data to apply the patch to.

Returns —= Dict[str, Any] : The updated data after applying the patch.

21.1.4 Function `dict_to_xml_annotation`

```
def dict_to_xml_annotation(
    value: Dict[str, Any]
) -> ome_types._autogenerated.ome_2016_06.xml_annotation.XMLAnnotation
```

Convert a dictionary to an XMLAnnotation object, handling nested dictionaries.
Args —= **`value`** : Dict[str, Any] : The dictionary to be converted to an XMLAnnotation object. It requires the key 'annotations' which is a dictionary of key-value pairs.
Returns —= XMLAnnotation : The resulting XMLAnnotation object.

21.1.5 Function `ensure_path_exists`

```
def ensure_path_exists(
    data: Dict[str, Any],
    path: str
) -> None
```

Ensure that the path exists in the data structure, creating empty lists or dicts as needed.
Args —= **`data`** : Dict[str, Any] : The data structure to modify.

`path` : str The path to ensure exists.

21.1.6 Function `flatten`

```
def flatten(
    container: Union[List, Tuple, Set]
) -> Generator[Any, None, None]
```

Flatten a nested container (list, tuple, or set).
Args —= **`container`** : Union[List, Tuple, Set] : The nested container to flatten.
Yields —= Any : Each non-container element in the flattened structure.

21.1.7 Function `from_dict`

```
def from_dict(
    ome_dict: Dict[str, Any],
    state: Optional[ome_types._autogenerated.ome_2016_06.ome.OME] = None
) -> ome_types._autogenerated.ome_2016_06.ome.OME
```

Convert a dictionary to an OME object.
Args —= **`ome_dict`** : Dict[str, Any] : The dictionary to convert.

`state` **: Optional[OME]** The initial OME state to update.

Returns —= OME : The resulting OME object.

21.1.8 Function `generate_paths`

```
def generate_paths(
    json_data: Union[Dict[str, Any], List[Any]],
    current_path: str = '',
    paths: List[str] = None
) -> List[str]
```

Generate all possible paths from a nested JSON structure.
Args —= **`json_data`** : Union[Dict[str, Any], List[Any]] : The nested JSON structure to traverse.

`current_path` **: str, optional** The current path being built. Defaults to "".
`paths` **: List[str], optional** The list to store all generated paths. Defaults to None.

Returns —= List[str] : A list of strings, where each string represents a path in the format "path/to/element = value".

21.1.9 Function `get_json`

```
def get_json(
    xml_root: xml.etree.ElementTree.Element,
    paths: Dict[str, Any] = {}
) -> Dict[str, Any]
```

Convert an XML tree to a JSON-like dictionary structure.
Args —= **`xml_root`** : ET.Element : The root element of the XML tree.

`paths` **: Dict[str, Any], optional** A dictionary to store the converted structure. Defaults to {}.

Returns —= Dict[str, Any] : The JSON-like dictionary representation of the XML tree.

21.1.10 Function `load_output`

```
def load_output(
    path: str
) -> Tuple[Optional[str], Optional[float], Optional[float], Optional[float]]
```

Load output from a file.
Args —= **`path`** : str : The file path to load from.
Returns —= Tuple[Optional[str], Optional[float], Optional[float], Optional[float]] :
The loaded output, cost, attempts, and prediction time.

21.1.11 Function `make_prediction`

```
def make_prediction(
    predictor: Any,
    in_data: Any,
    dataset: metagpt.utils.DataClasses.Dataset,
    file_name: str,
    index: int,
    should_predict: str = 'maybe',
    start_point: Optional[str] = None,
    data_format: Optional[str] = None,
    model: Optional[str] = None,
    out_path: Optional[str] = None
) -> None
```

Make a prediction using the specified predictor and add the result to the dataset.
Args —= **`predictor`** : Any : The predictor object.

`in_data` : Any Input data for the prediction.
`dataset` : Dataset The dataset to add the prediction to.
`file_name` : str Name of the file being processed.
`index` : int Index of the prediction.
`should_predict` : str, optional Whether to predict. Defaults to "maybe".
`start_point` : Optional[str], optional Starting point for the prediction. Defaults to None.
`data_format` : Optional[str], optional Data format. Defaults to None.
`model` : Optional[str], optional Model to use for prediction. Defaults to None.
`out_path` : Optional[str], optional Output path. Defaults to None.

21.1.12 Function `merge_xml_annotation`

```
def merge_xml_annotation(
    annot: Dict[str, Any],
    ome: Optional[str] = None
) -> Optional[str]
```

Merge the annotation section with the OME XML.
Args —= **`ome`** : Optional[str] : The OME XML string.

`annot` : Dict[str, Any] The annotation dictionary.

Returns —= Optional[str] : The merged XML string, or None if inputs are invalid.

21.1.13 Function `num_tokens_from_string`

```
def num_tokens_from_string(
    string: str,
    encoding_name: str = 'cl100k_base'
) -> int
```

Returns the number of tokens in a text string.
Args —= **`string`** : str : The input string to tokenize.

`encoding_name` : str, optional The name of the tokenizer encoding to use. Defaults to "cl100k_base".

Returns —= int : The number of tokens in the input string.

21.1.14 Function `read_ome_xml`

```
def read_ome_xml(
    path: str
) -> xml.etree.ElementTree.Element
```

Read an OME XML file and return the root element.
Args —= **`path`** : str : The path to the OME XML file.
Returns —= ET.Element : The root element of the XML tree.

21.1.15 Function `render_cell_output`

```
def render_cell_output(
    output_path: str
) -> None
```

Load the captured output from a file and render it.
Args —= **`output_path`** : str : Path to the output file where the cell output is saved.

21.1.16 Function `safe_float`

```
def safe_float(
    value: Any
) -> Optional[float]
```

Safely convert a value to float, returning None if conversion is not possible.
Args —= **`value`** : Any : The value to convert to float.
Returns —= Optional[float] : The float value if conversion is successful, None otherwise.

21.1.17 Function `save_and_stream_output`

```
def save_and_stream_output(
    output_path: str = 'out/jupyter_cell_outputs/cell_output_2024-07-27T19:37:02.981914_.json'
)
```

Context manager to capture the output of a code block, save it to a file, and print it to the console in real-time.
Args —= **`output_path`** : str : Path to the output file where the cell output will be saved.

21.1.18 Function `save_output`

```
def save_output(
    output: str,
    cost: float,
    attempts: float,
    pred_time: float,
    path: str
) -> bool
```

Save output to a file.
Args —= **`output`** : str : The output to save.

`cost` : **float** The cost of the prediction.
`attempts` : **float** The number of attempts made.
`pred_time` : **float** The prediction time.
`path` : **str** The file path to save to.

Returns —= bool : True if save was successful, False otherwise.

21.1.19 Function `update_state`

```
def update_state(
    current_state: ome_types._autogenerated.ome_2016_06.ome.OME,
    proposed_change: List[Dict[str, Any]]
) -> ome_types._autogenerated.ome_2016_06.ome.OME
```

Update the OME state based on proposed changes using JSONPatch, automatically creating missing nodes.
Args —= **`current_state`** : OME : The current OME state.

`proposed_change` : **List[Dict[str, Any]]** The change proposed as a JSON Patch document.

Returns —= OME : The updated OME state.
Raises —= ValueError : If the patch is invalid or cannot be applied, or if the resulting document is not a valid OME model.

21.2 Classes

21.2.1 Class `Tee`

```
class Tee(
    *streams
)
```

A class to duplicate output to multiple streams.

Methods

Method `flush`

```
def flush(
    self
)
```

Method `write`

```
def write(
    self,
    data
)
```

Generated by *pdoc* 0.11.1 (`https://pdoc3.github.io`).