Operators

Operators in DocETL are designed for semantically processing unstructured data. They form the building blocks of data processing pipelines, allowing you to transform, analyze, and manipulate datasets efficiently.

Overview

- Datasets contain items, where a item is an object in the JSON list, with fields and values. An item here could be simple text chunk or a document reference.
- DocETL provides several operators, each tailored for specific unstructured data processing tasks.
- By default, operations are parallelized on your data using multithreading for improved performance.

Caching in DocETL

DocETL employs caching for all LLM calls and partially-optimized plans. The cache is stored in the `.cache/docetl/general` and `.cache/docetl/llm` directories within your home directory. This caching mechanism helps to improve performance and reduce redundant API calls when running similar operations or reprocessing data.

Common Attributes

All operators share some common attributes:

- `name`: A unique identifier for the operator.
- `type`: Specifies the type of operation (e.g., "map", "reduce", "filter").

LLM-based operators have additional attributes:

- `prompt`: A Jinja2 template that defines the instruction for the language model.
- `output`: Specifies the schema for the output from the LLM call.
- `model` (optional): Allows specifying a different model from the pipeline default.
- `litellm_completion_kwargs` (optional): Additional parameters to pass to LiteLLM completion calls.

DocETL uses LiteLLM to execute all LLM calls, providing support for 100+ LLM providers including OpenAI, Anthropic, Azure, and more. You can pass any LiteLLM completion arguments using the `litellm_completion_kwargs` field.

Example:

```yaml
- name: extract_insights
  type: map
  model: gpt-4o-mini
  litellm_completion_kwargs:
    max_tokens: 500          # limit response length
    temperature: 0.7         # control randomness
    top_p: 0.9              # nucleus sampling parameter
  prompt: |
    Analyze the following user interaction log:
    {{ input.log }}

    Extract 2-3 main insights from this log, each being 1-2 words, to help
inform future product development. Consider any difficulties or pain points
the user may have had. Also provide 1-2 supporting actions for each insight.
    Return the results as a list of dictionaries, each containing 'insight'
and 'supporting_actions' keys.
  output:
    schema:
      insights: "list[{insight: string, supporting_actions: list[string]}]"
```

Input and Output

Prompts can reference any fields in the data, including:

- Original fields from the input data.
- Fields synthesized by previous operations in the pipeline.

For map operations, you can only reference `input`, but in reduce operations, you can reference `inputs` (since it's a list of inputs).

Example:

```yaml
prompt: |
  Summarize the user behavior insights for the country: {{ inputs[0].country
}}

  Insights and supporting actions:
  {% for item in inputs %}
  - Insight: {{ item.insight }}
  Supporting actions:
  {% for action in item.supporting_actions %}
  - {{ action }}
  {% endfor %}
  {% endfor %}
```

What happens if the input is too long?

When the input data exceeds the token limit of the LLM, DocETL automatically truncates tokens from the middle of the data to make it fit in the prompt. This approach preserves the beginning and end of the input, which often contain crucial context.

A warning is displayed whenever truncation occurs, alerting you to potential loss of information:

```
WARNING: Input exceeded token limit. Truncated 500 tokens from the middle of
the input.
```

If you frequently encounter this warning, consider using DocETL's optimizer or breaking down your input yourself into smaller chunks to handle large inputs more effectively.

Output Schema

The `output` attribute defines the structure of the LLM's response. It supports various data types (see schemas for more details):

- `string` (or `str`, `text`, `varchar`): For text data
- `integer` (or `int`): For whole numbers
- `number` (or `float`, `decimal`): For decimal numbers
- `boolean` (or `bool`): For true/false values
- `list`: For arrays or sequences of items
- objects: Using notation `{field: type}`
- `enum`: For a set of possible values

Example:

```yaml
output:
  schema:
    insights: "list[{insight: string, supporting_actions: string}]"
    detailed_summary: string
```

Keep Output Types Simple

It's recommended to keep output types as simple as possible. Complex nested structures may be difficult for the LLM to consistently produce, potentially leading to parsing errors. The structured output feature works best with straightforward schemas. If you need complex data structures, consider breaking them down into multiple simpler operations.

For example, instead of:

```yaml
output:
  schema:
    insights: "list[{insight: string, supporting_actions: list[{action: string, 
priority: integer}]}]"
```

Consider:

```yaml
output:
  schema:
    insights: "list[{insight: string, supporting_actions: string}]"
```

And then use a separate operation to further process the supporting actions if needed.

Read more about schemas in the schemas section.

Validation

Validation is a first-class citizen in DocETL, ensuring the quality and correctness of processed data.

Basic Validation

LLM-based operators can include a `validate` field, which accepts a list of Python statements:

```yaml
validate:
  - len(output["insights"]) >= 2
  - all(len(insight["supporting_actions"]) >= 1 for insight in 
output["insights"])
```

Access variables using dictionary syntax: `output["field"]`. Note that you can't access `input` docs in validation, but the output docs should have all the fields from the input docs (for non-reduce operations), since fields pass through unchanged.

The `num_retries_on_validate_failure` attribute specifies how many times to retry the LLM if any validation statements fail.

Advanced Validation: Gleaning

Gleaning is an advanced validation technique that uses LLM-based validators to refine outputs iteratively.

To enable gleaning, specify:

- `validation_prompt`: Instructions for the LLM to evaluate and improve the output.
- `num_rounds`: The maximum number of refinement iterations.
- `model` (optional): The model to use for the LLM executing the validation prompt. Defaults to the model specified for this operation. **Note that if the validator LLM determines the output needs to be improved, the final output will be generated by the model specified for this operation.**
- `if` (optional): A Python boolean expression (evaluated with `safe_eval`) that refers to **fields in the current `output`**. If the expression evaluates to `False`, DocETL skips gleaning entirely.

Example:

```yaml
gleaning:
  num_rounds: 1
  validation_prompt: |
    Evaluate the extraction for completeness and relevance:
    1. Are all key user behaviors and pain points from the log addressed in
the insights?
    2. Are the supporting actions practical and relevant to the insights?
    3. Is there any important information missing or any irrelevant
information included?
```

This approach allows for *context-aware* validation and refinement of LLM outputs. Note that it is expensive, since it at least doubles the number of LLM calls required for each operator.

Example map operation (with a different model for the validation prompt):

```yaml
- name: extract_insights
  type: map
  model: gpt-4o
  prompt: |
    From the user log below, list 2-3 concise insights (1-2 words each) and 1-
2 supporting actions per insight.
    Return as a list of dictionaries with 'insight' and 'supporting_actions'.
    Log: {{ input.log }}
  output:
    schema:
      insights_summary: "string"
  gleaning:
    if: "len(output['insights_summary']) < 10"  # Only refine if summary is
too short
```

```yaml
    num_rounds: 2 # Will refine up to 2 times if needed
    model: gpt-4o-mini
    validation_prompt: |
      There should be at least 2 insights, and each insight should have at least 1 supporting action.
```

Choosing a Different Model for Validation

You may want to use a different model for the validation prompt. For example, you can use a more powerful (and expensive) model for generating outputs, but a cheaper model for validation—especially if the validation only checks a single aspect. This approach helps reduce costs while still ensuring quality, since the final output is always produced by the more capable model.

Conditional Gleaning

You can also use the `if` field to conditionally skip gleaning. For example, if you only want to glean if the output is too short, you can use:

```yaml
gleaning:
  if: "len(output['insights_summary']) < 10"
  num_rounds: 2
```

If the `if` field evaluates to `False`, DocETL skips gleaning entirely. Or, if the `if` field does not exist, DocETL will always glean.

How Gleaning Works

Gleaning is an iterative process that refines LLM outputs using context-aware validation. Here's how it works:

1. **Initial Operation**: The LLM generates an initial output based on the original operation prompt.
2. **Validation**: The validation prompt is appended to the chat thread, along with the original operation prompt and output. This is submitted to the LLM. *Note that the validation prompt doesn't need any variables, since it's appended to the chat thread.*
3. **Assessment**: The LLM responds with an assessment of the output according to the validation prompt. The model used for this step is specified by the `model` field in the `gleaning` dictionary field, or defaults to the model specified for that operation.
4. **Decision**: The system interprets the assessment:
    - If there's no error or room for improvement, the current output is returned.

    - If improvements are suggested, the process continues.
5. **Refinement**: If improvements are needed:
    - A new prompt is created, including the original operation prompt, the original output, and the validator feedback.
    - This is submitted to the LLM to generate an improved output.
6. **Iteration**: Steps 2-5 are repeated until either:
    - The validator has no more feedback (i.e., the evaluation passes), or
    - The number of iterations exceeds `num_rounds`.
7. **Final Output**: The last refined output is returned.

Note that gleaning can significantly increase the number of LLM calls for each operator, potentially doubling it at minimum. While this increases cost and latency, it can lead to higher quality outputs for complex tasks.