Extending KomMKonLLM

This document offers pointers for users who want to create interfaces to additional LLMs or covering array generators.

If you have developed a new interface and think it would be useful for others, please do not hesitate to create a pull request!

Code Structure

This section gives a high-level overview o the code and directory structure of this project.

A Docker Compose file contained in the root folder ties together the infrastructure and offers configuring options. In most cases, you do not need to modify any further code to adjust the behavior of our tool.

Besides this file, the following sub-projects are contained in the folder:

- `meta-runner` is the primary controller involved in coordinating the testing process. It contains the overall testing logic as well as methods to generate synonyms for sentences, construct covering arrays, translate them to concrete LLM queries, and submit them to the model interface.
- `pyann-model-executor` contains model interface implementations. It acts as a glue layer between the runner and the actual LLMs, offering a unified interface.

- `ResultStore` implements an interface to the underlying database and additionally contains a JupyterLab installation used for the analysis of results.
- `t5-flask-app` is a wrapper around a T5 LLM using pretraining by HuggingFace. It is mostly obsolete, but may be interesting to developers who wish to quickly set up an interface for a model offered by HuggingFace.
- `data` is initially empty and later used to store the Postgres DBMS data files, i.e. the actual database.
- `example` contains example test sentences, generated synonyms and associated covering arrays as well as the LLMs output. It primarily serves as supporting documentation to ease the understanding of our approach.

All of our code is in Python (we use 3.12 and 3.13 internally, but any version above 3.8 will likely work). The subfolders listed above commonly contain a `requirements.txt` file that lists dependencies as well as a `Dockerfile` (for those subprojects that are built as custom Docker images).

LLM interfaces

In most circumstances, you do not need to create your own LLM interface. We offer an adapter for Ollama, which is a wrapper around dozens of models; see the **Configuration** section in the main README. If you wish to define your own interface nonetheless, please follow the instructions below.

There are three steps involved in creating an adapter for a new LLM:

1. Create a subclass of `ModelExecutor` that allows you to query and update your LLM.
2. Update the `models` variable in `pyann-model-executor/main.py`.
3. If required, update `docker-compose.yaml` to start your LLM in a Docker container.

LLM interfaces are implemented in `pyann-model-executor/models.py`. The superclass for such interfaces is `ModelExecutor`; to add your own LLM interface, create a subclass that inherits from this class and implement the `query()`, `set_settings()` and `get_settings()` methods.

A very simple example is provided using our wrapper for T5, which you can copy and modify for your purposes:

```
class T5Executor(ModelExecutor):
    def __init__(self):
        """Set model parameters and default values."""
        self.settings = {
            'early_stopping': True,
            'max_length': 50,
            'num_beam': 2
        }
        self.endpoint = getenv('MODEL_IP', default='t5:5069')

    def query(self, prompt: str) -> str:
        """Query the model with the given string."""
        params = self.settings.copy()
        params['prompt'] = prompt
        return requests.get(f'http://{self.endpoint}/query', params=params)

    def set_settings(self, settings: dict) -> dict:
        """Update the model settings."""
        self.settings = settings
        return self.settings

    def get_settings(self) -> dict:
        """Retrieve the current model settings."""
        return self.settings
```

Next, you will need to update the `models` variable in `pyann-model-executor/main.py`. The keys in this dictionary are used when parsing the `MODEL_UNDER_TEST` environment variable.

Finally, update `docker-compose.yaml`; if required, you can add your LLM in its own container here. You most likely want to update the `MODEL_UNDER_TEST` option of `meta_runner` too to test your implementation right away.

CA generators

Interfaces to CA generators are implemented in `meta-runner/src/payload_generator/ca_generator.py`. Create a subclass of CaGenerator and implement at least its `__init__()` and `generate()` method, optionally overriding the `read()` method to modify how CA file contents are retrieved.

A basic skeleton for creating your own CA generator wrapper looks like this:

```python
class MyGenExecutor(CaGenerator):
    def __init__(self, gen_path: Path):
        self.gen_path = gen_path

    def generate(self, synonyms: list[list[str]], strength: int) -> tuple[P
        """Generate a CA and return its location and number of rows."""
        cardinalities = CaGenerator.cardinalities(synonyms, as_str=True)
        path = CaGenerator.ca_filename(synonyms, strength)
        if not (path.is_file() and path.stat().st_size > len(synonyms)):
            # Only generate the CA if it does not exist yet
            # TODO Execute your CA generator here
            return path, num_rows
        else:
            # Otherwise just open the file and count the lines
            return path, len([self.read(synonyms, strength)])
```

See the comments in `ca_generator.py` for further hints and code to copy and adjust.

Do not forget to add your generator in `CaGenerator::get_generator()` when you are done!