Advanced Human Language Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Outline

Task 2.3 - DDI using large language models

Assignment

Write a python program that parses the given XML file and recognizes and classifies sentences stating drug-drug interactions. The program must use a LLM-based approach.

```
$ python3 ./nn-DDI.py devel.xml result.out
DDI-DrugBank.d398.s0|DDI-DrugBank.d398.s0.e0|DDI-DrugBank.d398.s0.e1|effect
DDI-DrugBank.d398.s0|DDI-DrugBank.d398.s0.e0|DDI-DrugBank.d398.s0.e2|effect
DDI-DrugBank.d211.s2|DDI-DrugBank.d211.s2.e0|DDI-DrugBank.d211.s2.e5|mechanism
...
```

Outline

General Structure

We will use the same representation than in the NN approach:

Identify the target pair of drugs in a sentence with labels `DRUG1` and `DRUG2`, and ask the model to decide whether the sentence states an interaction between them.

Original sentence:

> *Exposure to oral ketamine is unaffected by itraconazole compounds but greatly increased by ticlopidine.*

Generated examples (potential interactions)

Pair	Masked sentence
e0-e1	Exposure to oral DRUG1 is unaffected by DRUG2 but greatly increased by DRUG_OTHER.
e0-e2	Exposure to oral DRUG1 is unaffected by DRUG_OTHER but greatly increased by DRUG2.
e1-e2	Exposure to oral DRUG_OTHER is unaffected by DRUG1 but greatly increased by DRUG2.

General Structure

- We will need to convert the dataset to/from the pseudo-XML representation.
- We will need to instruct the LLM with a suitable prompt describing the task, followed by each target sentence to annotate.
- We will be using two different strategies:
  - Few-shot learning: Add some solved examples in the prompt, so the model understands better what is expected.
  - Fine-tuning: Tune the LLM weights using several (dozens, hundreds, ...) solved examples, so it learns the task and does not need the few-show examples each time.

Few-shot learning

- The LLM is asked to perform an specific task in a prompt, which contains:
  - The task description. It must be precise and well explained.
  - Some pairs of (input, expected output) examples.
  - The input for the example to solve
- Adjustable elements:
  - The **prompt**: How the target task is described. Includes specific orders to produce output in certain ways/formats, and NOT to produce undesired output.
  - The **number** of few-shot examples.
  - The **variety** of used few-shot examples (e.g. making sure all interaction types appear in some example).

Fine-Tuning

- The LLM is presented a collection of prompts, and the model weights are iteratively updated. Each prompt includes:
  - The task description.
  - One pair of (input, expected output) example.
- Adjustable elements:
  - The **prompt**: How required task is described. Specific orders to produce output in certain ways, and NOT to produce undesired output. Not as crucial as in few-shot
  - The **amount** of fine tuning examples
  - The **distribution** of used examples (e.g. making sure all interaction types are represented enough, maybe balancing too frequent or too scarce classes).

Outline

Outline

Few shot - Annotator

```
# load training data (FS examples)
trainfile = os.path.join(paths.DATA,traindata+".xml")
fs_examples = Examples(trainfile, "DDI").select_examples(num_few_shot,
                                                         balanced=True)
# load prompts, create few-shot prompt
prompts = Prompts(promptfile, fs_examples)

# load test data
testfile = os.path.join(paths.DATA,testdata+".xml")
test = Examples(testfile, "DDI")

# load model and tokenizer
if ollama:
   engine = Inference(model, ollama=True)
else :
   MODEL_PATH = f"/scratch/nas/1/PDI/mml0/models/{model}"
   engine = Inference(MODEL_PATH, quantized=quantized)

# annotate each example in testdata
annotated = []
for i,ex in enumerate(test.select_examples()):
    # create prompt for this example, adding it to FS prompt
    messages = prompts.prepare_messages(ex['input'])
    # call model to generate response
    gen_text = engine.generate(messages)
    # store responses
    ex['predicted'] = gen_text
    ex['evaluator'] = test.eval_format(ex,gen_text)
    annotated.append(ex)

# print results
for e in annotated: print(e["evaluator"])
```

Outline

Fine tuning - Trainer

```python
# load prompts
prompts = Prompts(promptfile)

# load model and tokenizer
MODEL_PATH = f"/scratch/nas/1/PDI/mml0/models/{model}"
engine = FineTuning(MODEL_PATH, quantized=quantized)

# load and tokenize datasets
trainfile = os.path.join(paths.DATA,traindata+".xml")
train_examples = Examples(trainfile, "DDI").select_examples(5000,
                                                             balanced=True)
train_dataset = engine.tokenize_dataset(train_examples, prompts)

valfile = os.path.join(paths.DATA,valdata+".xml")
val_examples = Examples(valfile, "").select_examples(500,
                                                      balanced=True)
val_dataset = engine.tokenize_dataset(val_examples, prompts)

# Fine-tune the model and save results
engine.train(train_dataset,
             val_dataset,
             outputdir)
```

Fine tuning - Annotator

```python
# load prompts
prompts = Prompts(promptfile)

# load test/devel dataset
testfile = os.path.join(paths.DATA,testdata+".xml")
test = Examples(testfile, "DDI")

# load model and tokenizer
MODEL_PATH = f"/scratch/nas/1/PDI/mml0/models/{model}"
engine = Inference(MODEL_PATH, quantized=quantized, peft=weightdir)

# analyze each example
annotated = []
for i,ex in enumerate(test.select_examples()):
    # prepare sequence of messages for this example
    messages = prompts.prepare_messages(ex['input'])
    # call model to generate response
    gen_text = engine.generate(messages)
    # extract json from response
    ex["predicted"] = gen_text
    ex['evaluator'] = test.eval_format(ex,gen_text)
    annotated.append(ex)

# save output
for e in annotated:  print(e["evaluator"])
```

Outline

Auxiliary classes - Examples

```python
class Examples:
   ## constructor: parses data XML file, and converts sentences into
   ## the format expected by the LLM for the given task (NER/DDI).
   def __init__(self, xmlfile, task)

   ## get 'numFS' examples (or all dataset if ommited)
   ## balance classes if requested
   def select_examples(self, numFS=-1, balanced=False)

   ## convert LLM output into the format expected by the
   ## evaluator for the task this instance is loaded for
   def eval_format(self, ex, text)
```

- Class `Examples` takes care of loading the dataset XML files and converting the examples into the appropriate representation for the LLM.
- Class `Examples` also converts LLM responses into the output format expected by the evaluator.
- For DDI, `balanced` parameter helps selecting training examples.
- Method `select_examples` can be replaced with a custom selection strategy.

Auxiliary classes - `Prompts`

```python
class Prompts:
    # Constructor: loads prompts from json file (sysprompt and usrprompt)
    # If a list of fs_examples is provided, they are added to the prompt
    def __init__(self, promptfile, fs_examples=[])

    # Build prompt for a particular example, concantenating the
    # base prompt (sysprompt + usrprompt + fs_xamples --if any)
    # with the given user request 'question', plus the expected
    # answer --if provided
    def prepare_messages(self, question, answer="")
```

- The constructor stores a prompt that contains the system prompt and the user prompt. For few-shot, it also contains the provided solved examples.
- Method `prepare_messages` is used in few-shot to add a request (question) to the prompt (which already has some solved examples), without the corresponding answer.
- Method `prepare_messages` is used in fine-tuning training to add a complete solved example (request plus expected answer). In this scenario, the base prompt does not contain few-shot examples.

Auxiliary classes - `model.Inference`

```python
class Inference:
    # load given model in inference mode
    def __init__(self, model_path, quantized=False, peft=None, ollama=False)

    # Generate completion for given messages
    def generate(self, messages)

```

- Constructor loads requested model (either using ollama or HF transformers) in inference mode. The model can be loaded in full, or quantized to 4 bits. If `peft` is given it is expected to be the path to fine-tuned LoRa adapters for the given model, which are also loaded.
- Method `generate` uses the loaded LLM to generate a completion for given prompt (typically, the answer to the last request in it).

Auxiliary classes - `model.FineTuning`

```
class Inference:
    # load given model in LoRa fine tuning mode.
    def __init__(self, model_path, quantized=False)

    # Tokenizes given dataset with the model appropriate tokenizer.
    # Returns a "datasets.Dataset" object suitable for HF trainer
    def tokenize_dataset(self, dataset, prompts)

    # Fine tunes model with given train and validation data.
    # Saves tuned weights in outputdir
    def train(self, train_dataset, val_dataset, outputdir)

```

- The constructor loads requested model (quantized if required) and adds LoRa adapters to be tuned.
- Method `tokenize_datasets` uses model tokenizer to preprocess the training/validation data and convert them into batched `datasets.Dataset`, suitable for the trainer.
- Method `train` performs the fine-tuning using training data and computes loss on validation data at each epoch.
- Method `train` can be adjusted to change learning parameters if needed.

Outline

Build a good LLM-based DDI detection

Strategy: Experiment with different few-shot configurations:

- Prompts:
  - Explain better/more clearly the task.
  - If the model produces wrong output, insist on what is the expected output or its right format.
  - If the model misses some expected output, insist on its importance.
  - If the model produces unrequested output, explicitly request it not to do so.
- Few-shot examples
  - Experiment with different number of few-shot examples.
  - Experiment with better example selection methods (even hand-picking most informative examples from training set). Modify `select_examples` method to achieve this.
- Model
  - Use different models (llama, qwen, gemma, mistral...).
  - Experiment with/without quantization

Build a good LLM-based DDI detection

Strategy: Experiment with different fine-tuning configurations:

- Prompts:
  - Improve prompts as in few-shot (although it has much less effect here).
- Training examples
  - Experiment with different number of training examples.
  - Experiment with better example selection methods. Modify `select_examples` method to achieve this.
- Model
  - Use different models (llama, qwen, gemma, mistral...) and sizes.
  - Experiment with/without quantization
- Learning hyperparameters
  - Learning rate
  - Epochs
  - batch size (or gradient accumulation steps)

Build a good LLM-based DDI detection

Few-shot caveats:

- Model context size: Too many few-shot examples might fill the model context window and make it forget the instructions.
- Available GPU RAM: Too many few-shot examples, even if they don't fill the context window, may not fit into GPU RAM.
- Token offset errors: The model might slightly alter the text, which will result in wrong offsets when converting the XML tags to the format expected by the evaluator.

Fine-tuning caveats:

- Available GPU RAM: Fine tuning a model takes about 3 times the RAM used in inference. Bigger GPUs are required. Smaller batch sizes may help. Quantization too.
- Available time: Fine tuning takes a lot of time, since the dataset needs to be run through the model, loses computed, weights updated. For each example, several epochs.

Outline

Exercise Goals

What you should do:

- Experiment with few-shot and fine-tuning variations.
- Experiment with different prompts and few-shot examples.
- Experiment with different prompts and training examples.
- Experiment with different learning hyperparameters.
- Keep track of tried variants and parameter combinations.

What you should **NOT** do:

- Get insights about errors from `devel` dataset. You have `train` for that. `devel` is only used to evaluate the performance of a given configuration.
- Select architectures or hyperparameters based on system performance on `test` dataset. You have `devel` for that. `test` is only used to evaluate model generalization ability once the best configuration has been chosen.

Deliverables

At the end of the DDI task, you will need to deliver a single report on the work carried out on DDI-ML, DDI-NN, and DDI-LLM systems

So, during the development and experimentation on DDI-LLM:

- keep track of tried/discarded architectures/hyperparameters
- keep track of tried/discarded input information.
- Record obtained results in the different experiments, and compile the information you'll later need to elaborate the report.

Annex: Resources to use LLMs. Ollama

ollama can be used for few-shot experiments.

- Ollama runs LLMs with or without GPU (much slower without)
- Install ollama: `https://ollama.com/download`
- Install ollama python API: `pip install ollama`
- Download desired ollama models

  ```
  ollama pull llama3.2:3b
  ollama pull qwen3:8b
  ollama pull ministral3:8b
  ```

  See `https://ollama.com/library` for a list
- Use provided `fewshot.py` with `-ollama` option.

Annex: Resources to use LLMs. Huggingface

Huggingface Transformers can be used for fine-tuning experiments.

- If you have a GPU (with at least 8-10GB), you can install `requirements.txt` in a python virtual environment, and run the programs out-of-the box.
- Use HF model id (e.g. `meta-llama/Llama-3.2-3B-Instruct`) as `MODEL_PATH` when loading the model.
- If you don't have a GPU, you can use `google colab`, or UPC boada cluster, where `llama3.2-3B` and `qwen3-3B` are already installed.