Alpaca Code Analysis

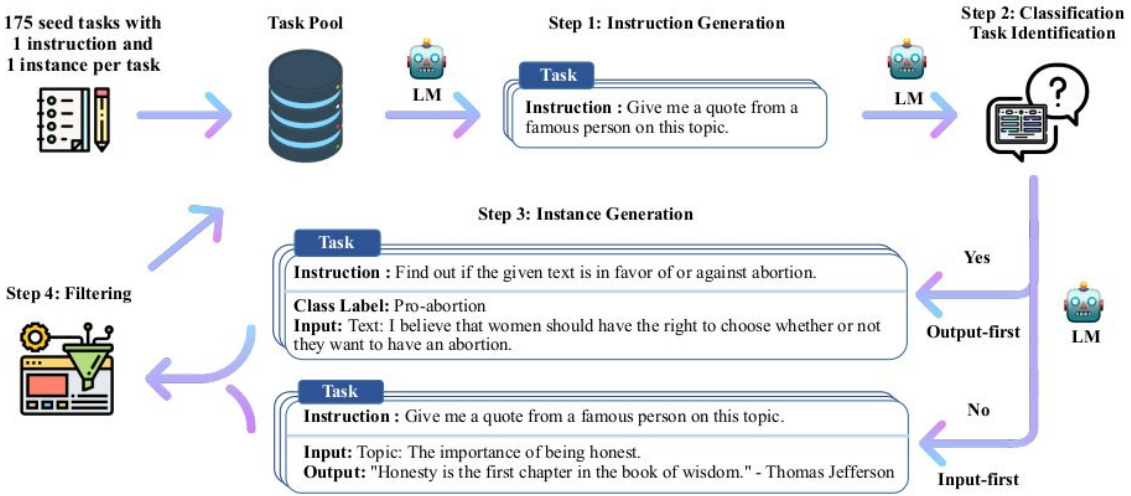
Data Networks Lab
Khen Bo Kan

Instruction Generation

Preparation for instruction task generation:

- Seed tasks - human-written instruction tasks (175 - Alpaca, 20 - codeAlpaca)
- "Prompt.txt" - Instruction auto-generation requirements for chatGPT

  https://github.com/sahil280114/codealpaca/blob/master/prompt.txt



A high-level overview of SELF-INSTRUCT

Instruction Generation

Process:

1. Load and read 'seed_task.json'
2. Format the seed tasks to fit the desired 'instruction, input, output' format - 'seed_instruction_data'
3. Tokenize 'seed_instruction_data'
4. Sample the **seed instructions** and generate the prompt

```python
def encode_prompt(prompt_instructions):
    """Encode multiple prompt instructions into a single string."""
    prompt = open("./prompt.txt").read() + "\n"

    for idx, task_dict in enumerate(prompt_instructions):
        (instruction, input, output) = task_dict["instruction"], task_dict["input"], task_dict["output"]
        instruction = re.sub(r"\s+", " ", instruction).strip().rstrip(":")
        input = "<noinput>" if input.lower() == "" else input
        prompt += f"###\n"
        prompt += f"{idx + 1}. Instruction: {instruction}\n"
        prompt += f"{idx + 1}. Input:\n{input}\n"
        prompt += f"{idx + 1}. Output:\n{output}\n"
    prompt += f"###\n"
    prompt += f"{idx + 2}. Instruction:"
    return prompt
```

Func to create the prompt for chatGPT

```python
def generate_instruction_following_data(
    output_dir="./",
    seed_tasks_path="./seed_tasks.jsonl",
    num_instructions_to_generate=20000,
    model_name="text-davinci-003",
    num_prompt_instructions=3,
    request_batch_size=5,
    temperature=1.0,
    top_p=1.0,
    num_cpus=16,
):
    seed_tasks = [json.loads(l) for l in open(seed_tasks_path, "r")]
    seed_instruction_data = [
        {"instruction": t["instruction"], "input": t["instances"][0]["input"], "output": t["instances"][0]["output"]}
        for t in seed_tasks
    ]
    print(f"Loaded {len(seed_instruction_data)} human-written seed instructions")
```

Step 1 and 2

```python
    all_instructions = [d["instruction"] for d in seed_instruction_data] + [
        d["instruction"] for d in machine_instruction_data
    ]
    all_instruction_tokens = [scorer._tokenizer.tokenize(inst) for inst in all_instructions]

    while len(machine_instruction_data) < num_instructions_to_generate:
        request_idx += 1

        batch_inputs = []
        for _ in range(request_batch_size):
            # only sampling from the seed tasks
            prompt_instructions = random.sample(seed_instruction_data, num_prompt_instructions)
            prompt = encode_prompt(prompt_instructions)
            batch_inputs.append(prompt)
        decoding_args = utils.OpenAIDecodingArguments(
            temperature=temperature,
            n=1,
            max_tokens=3072,  # hard-code to maximize the length. the requests will be automatically adjusted
            top_p=top_p,
            stop=["\n20", "20.", "20."],
        )
```

Step 3 and 4

Instruction Generation

5. Input the generated prompt (batch_inputs) into chatGPT to create instructions

6. Process the GPT response: cleaning and filtering (length and key words, punctuation, etc.)

7. Compute similarity for further filtering. Only responses that have the similarity score below 70% are kept.

Result:

Increase of instruction tasks from 20 to 20K (less than 200$)

```
decoding_args = utils.OpenAIDecodingArguments(
    temperature=temperature,
    n=1,
    max_tokens=3072,  # hard-code to maximize the length. the requests will be automatically adjusted
    top_p=top_p,
    stop=["\n20", "20.", "20."],
)
request_start = time.time()
print("Calling openai...")
results = utils.openai_completion(
    prompts=batch_inputs,
    model_name=model_name,
    batch_size=request_batch_size,
    decoding_args=decoding_args,
    logit_bias={"50256": -100},  # prevent the <|endoftext|> token from being generated
)
```

Step 5

```
instruction_data = []
for result in results:
    new_instructions = post_process_gpt3_response(num_prompt_instructions, result)
    instruction_data += new_instructions

total = len(instruction_data)
keep = 0
for instruction_data_entry in instruction_data:
    # computing similarity with the pre-tokenzied instructions
    new_instruction_tokens = scorer._tokenizer.tokenize(instruction_data_entry["instruction"])
```

Step 6 and 7

Fine-tuning

- **SupervisedDataset**:

Loading and preprocessing the data, tokenizing the input, and storing the input IDs and labels for each example.

```
IGNORE_INDEX = -100
DEFAULT_PAD_TOKEN = "[PAD]"
DEFAULT_EOS_TOKEN = "</s>"
DEFAULT_BOS_TOKEN = "</s>"
DEFAULT_UNK_TOKEN = "</s>"
PROMPT_DICT = {
    "prompt_input": (
        "Below is an instruction that describes a task, paired with an input that provides further context. "
        "Write a response that appropriately completes the request.\n\n"
        "### Instruction:\n{instruction}\n\n### Input:\n{input}\n\n### Response:"
    ),
    "prompt_no_input": (
        "Below is an instruction that describes a task. "
        "Write a response that appropriately completes the request.\n\n"
        "### Instruction:\n{instruction}\n\n### Response:"
    ),
}
```

- **DataCollatorForSupervised Dataset:**

Retrieving the input IDs and labels from the batch of examples; adds padding so that the input and labels have the same length.

```
data_module = make_supervised_data_module(tokenizer=tokenizer, data_args=data_args)
trainer = Trainer(model=model, tokenizer=tokenizer, args=training_args, **data_module)
trainer.train()
trainer.save_model(training_args.output_dir)
```

```
def make_supervised_data_module(tokenizer: transformers.PreTrainedTokenizer, data_args) -> Dict:
    """Make dataset and collator for supervised fine-tuning."""
    train_dataset = SupervisedDataset(tokenizer=tokenizer, data_path=data_args.data_path)
    data_collator = DataCollatorForSupervisedDataset(tokenizer=tokenizer)
    return dict(train_dataset=train_dataset, eval_dataset=None, data_collator=data_collator)
```