Fine-Tuning and Retrieval-Augmentation: Teaching LLMs new tricks

Self Learning Tutorial

Terence Lim
EID: tl27939

Outline

- Introduction
  - Open-source LLMs: Gemma and Llama
- Task 1: Medical natural language inference (NLI) task
  - MedNLI benchmark dataset
  - Fine-tuning with consumer-grade hardware using quantization and low-rank adaptation
  - Evaluation of fine-tuned model predictions
- Task 2: Medical question-answering (QA) task
  - PubMedQA benchmark dataset
  - Retrieval-augmented generation with embedding models and vector database
  - Evaluation of document-retrieval and model-prediction accuracy
- Appendix
  - Fine-tuning transformer models such as the electra-discriminator encoder
  - References

Overview

Instruction-tuned LLM's have achieved remarkable natural language processing capabilities with promising applications in healthcare, for example, as AI medical assistants to interact “knowledgeably” or “informatively” with patients or clinical staff, hence allowing scarce, skilled, human labor to be deployed elsewhere.

Open-source LLMs, such as Meta's LLama-2 (July 2023) and Google's Gemma (February 2024) may be **trained and run locally on consumer hardware**. While these may not be as powerful as the closed-source behemoths such as GPT4, they can be more suitable for use in the healthcare domain due to concerns about data privacy and security, especially if they can be customized for the targeted use-case.

Out-of-the-box, open-source LLM's may not have been instructed for the desired downstream tasks. The technique of **fine-tuning** LLM's can be employed to learn such new tasks. LLMs have also been observed to "hallucinate" if pre-trained with stale or irrelevant data, but fine-tuning with large, new datasets can be computationally expensive: enter **retrieval-augmented generation (RAG)** as a feasible solution.

This tutorial demonstrates how to (1) fine-tune the latest open-source LLM's, using mere consumer-grade systems, and (2) implement RAG for medical NLP tasks.

Open-source LLM’s

- Following the impact the closed-source GPT4 from OpenAI, many smaller, open-source, pretrained LLM’s released. Although less powerful (but improving), these have advantages, that are particularly important for healthcare applications:
  - Transparency and flexibility, with full in-house control over sensitive information hence reducing the risk of data leak or unauthorized access
  - Cost savings because no licensing fees are involved
  - Benefit from community contributions and experimentations
- One well-performing open source LLM with a license that allows commercial use is LlaMa-2, released in July 2023 by Meta, with pre-trained and chat-tuned models of 7 to 70 billion parameters.
- In February 2024, Google unveiled open-source Gemma models designed for smaller tasks such as simple chatbots or summarizations. The 7 billion parameter version is claimed to surpass significantly larger models on key benchmarks and can run on a developer’s laptop, while the 2 billion parameter version can run directly on mobile devices.
- Huggingface provides a repository for open-source transformer and large-language models, and a standard interface for fine-tuning and other usages.
  - Community contributions such as quantization and low-rank adaptation “fit” models in consumer hardware
  - the Appendix section overview the standard HF functions and parameters for performing fine-tuning

Llama model

LLaMA-2 comes in 3 different sizes (7B, 13B, and 70B parameters) in both chat and base versions.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	Model architecture:	Data collection for helpfulness and safety:
13B	Pretraining Tokens: 2 Trillion	Supervised fine-tuning: Over 100,000
70B	Context Length: 4096	Human Preferences: Over 1,000,000

Image credits: Meta

Llama-2 is a “gated model”, to access on Hugging Face:

- Begin by filling an access form on Meta’s website. You should receive an approval email fairly quickly
- Hugging Face’s model card shows another license to be accepted: the email used on Meta’s access form must be the same as that used in your Hugging Face account.
- To download the models from Hugging Face we must authenticate ourselves. Head to Hugging Face Settings > Access Tokens > New token, then create a new and copy this access token to place into your code later. You can also save the token on your machine using the `huggingface-cli` login terminal login command.

Gemma model

- The models come in two sizes: 2B and 7B parameters, with each size offering both a base (pretrained) and an instruction-tuned variant
  - 2B: Trained on 2T tokens. With a memory footprint of approximately 1.5GB, it's well-suited for tasks like text classification and straightforward question answering.
  - 7B: Trained on 6T tokens. Suitable for consumer-grade GPUs and TPUs. 5GB memory requirement enables tackling more intricate tasks such as summarization and code generation.
- Training data primarily consisting of English content sourced from Web Docs, mathematical texts, and code.
- Impressive context length of 8,192 tokens.
- Commercial use allowed

Parameters	2B	7B
*d*_model	2048	3072
Layers	18	28
Feedforward hidden dims	32768	49152
Num heads	8	16
Num KV heads	1	16
Head size	256	256
Vocab size	256128	256128

Table 1 | Key model parameters.

User:	`<start_of_turn>user`
	`Knock knock.<end_of_turn>`
	`<start_of_turn>model`

Model:	`Who's there?<end_of_turn>`

User:	`<start_of_turn>user`
	`Gemma.<end_of_turn>`
	`<start_of_turn>model`

Model:	`Gemma who?<end_of_turn>`

Table 4 | Example dialogue with user and model control tokens.

Task 1. Medical NLI

NLI in the medical domain

- Natural language inference (NLI) is the task of determining whether a given hypothesis can be inferred from a given premise. This seemingly simple, but challenging problem, formerly known as recognizing textual entailment (RTE) (Dagan et al., 2006) has long been a popular task among NLP researchers.
- Romanov and Shivade (2018) explored the problem of NLI in the clinical domain, and introduced MedNLI - a new, publicly available, expert annotated dataset.
  - As the source of premise sentences, they used the MIMIC-III v1.3 (Johnson et al., 2016) database. With de-identified records of 38,597 patients, it is the largest repository of publicly available clinical data, and contains 2,078,705 clinical notes written by healthcare professionals in English.
  - The hypothesis sentences were generated by clinicians. They were asked to write three sentences (hypotheses): 1) A clearly true statement, 2) A clearly false statement, and 3) A statement that might be true or false. This procedure produces three training pairs of sentences for each initial premise with three different labels: entailment, contradiction, and ne

#	Premise	Hypothesis	Label
1	ALT , AST , and lactate were elevated as noted above	patient has abnormal lfts	entailment
2	Chest x-ray showed mild congestive heart failure	The patient complains of cough	neutral
3	During hospitalization , patient became progressively more dyspnic requiring BiPAP and then a NRB	The patient is on room air	contradiction
4	She was not able to speak , but appeared to comprehend well	Patient had aphasia	entailment
5	T1DM : x 7yrs , h/o DKA x 6 attributed to poor medication compliance , last A1c [** 3-23 **] : 13.3 % 2	The patient maintains strict glucose control	contradiction
6	Had an ultimately negative esophagogastroduodenoscopy and colonoscopy	Patient has no pain	neutral
7	Aorta is mildly tortuous and calcified .	the aorta is normal	contradiction

Table 1: Examples from the development set of MedNLI

MedNLI dataset

- Clone https://github.com/jgc128/mednli
- Data files are jsonl-formatted, containing
- Training, development and test pairs of premises and hypotheses (with gold labels)

Dataset size	
Training pairs	11232
Development pairs	1395
Test pairs	1422
Average sentence length in tokens	
Premise	20.0
Hypothesis	5.8
Maximum sentence length in tokens	
Premise	202
Hypothesis	20

Table 2: Key statistics of the dataset

Get mednli dataset

```
# helper to read jsonl
import json
def read_jsonl(filename, max_samples=None):
    """helper to read jsonl files as pandas dataframe"""
    lines = []
    with open(filename) as f:
        lines = f.read().splitlines()
    line_dicts = [json.loads(line) for line in lines]
    max_samples = max_samples or len(line_dicts)
    return pd.DataFrame(line_dicts).iloc[:max_samples]
```

```
# read in train, dev and test sets
X_train = read_jsonl('mednli/mli_train_v1.jsonl')
y_train = X_train['gold_label']
X_dev = read_jsonl('mednli/mli_dev_v1.jsonl')
y_dev = X_dev['gold_label']
X_test = read_jsonl('mednli/mli_test_v1.jsonl')
y_test = X_test['gold_label']
```

```
# Show premise, hypothesis and label for dev examples
X_dev.iloc[:6][['sentence1', 'sentence2', 'gold_label']]
```

	sentence1	sentence2	gold_label
0	No history of blood clots or DVTs, has never h...	Patient has angina	entailment
1	No history of blood clots or DVTs, has never h...	Patient has had multiple PEs	contradiction
2	No history of blood clots or DVTs, has never h...	Patient has CAD	neutral
3	Over the past week PTA he has been more somnol...	He has been less alert over the past week	entailment
4	Over the past week PTA he has been more somnol...	Over the past week he has been alert and orie...	contradiction
5	Over the past week PTA he has been more somnol...	He is disorientated and complains of weakness	neutral

Fine-tune LLMs for Medical NLI

Initialize model and parameters

Pre-trained model may be loaded from HF hub by name, or from training “checkpoints” in local folders. Set TrainingArguments to be used by the Trainer class to configure for fine-tuning downstream tasks (see Appendix for descriptions of other parameters):

- optim — The optimizer to use: adamw_hf, adamw_torch, adamw_torch_fused, adamw_apex_fused, adamw_anyprecision or adafactor.
- lr_scheduler_type — The scheduler type to use.
- learning_rate — The initial learning rate.
- weight_decay — The weight decay to apply (if not zero) to all layers except all bias and LayerNorm weights.
- fp16 — Whether to use fp16 16-bit (mixed) precision training instead of 32-bit training.
- prediction_loss_only — When performing evaluation and generating predictions, only returns the loss.

```
# pre-trained LLM to use
model_id = "meta-llama/Llama-2-7b-chat-hf"
model_id = "meta-llama/Llama-2-13b-chat-hf"
model_id = "google/gemma-2b-it"
model_id = "google/gemma-7b-it"
output_dir = os.path.join('models', model_id)
```

```
# select to load model from HF hub, or previously checkpoint-
from_checkpoint = model_id
#from_checkpoint = output_dir
#from_checkpoint = os.path.join(output_dir, 'checkpoint-474')
```

```
# whether to train and/or evaluate
do_train = True
do_eval = True
```

```
# define arguments for trainer
training_args = TrainingArguments(
    output_dir=output_dir,
    optim="paged_adamw_32bit",
    lr_scheduler_type="cosine",
    learning_rate=2e-4,
    weight_decay=0.001,
    bf16=False,
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    group_by_length=False,
    gradient_accumulation_steps=8,
    fp16=True,   # use mixed precision floats
```

Load and quantize the model

If a model is “too big” for the computer system available, then standard Hugging Face pipelines need to be reconfigured to be able to load, train, or run inference:

Quantization is a technique to represent the weights and activations with low-precision data types like 8-bit integer (int8) instead of the usual 32-bit floating point (float32). Reducing the computational and memory costs means operations like matrix multiplication can be performed much faster and even allows to run models on embedded devices.

- The bitsandbytes library is a lightweight Python wrapper around CUDA custom functions, in particular 8-bit optimizers, matrix multiplication (LLM.int8()), and 8 & 4-bit quantization functions. A quantization_config is included in AutoModel for loading pre-trained models.

Get quantized pre-trained model

```
# Load model from HF hub or local folder
compute_dtype = getattr(torch, "float16")
if do_train:   # Load and quantize pre-trained model to start fine-tune
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=False,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=compute_dtype
    )
    model = AutoModelForCausalLM.from_pretrained(
        from_checkpoint,
        device_map=device,  # "auto" may be slower because offload to cpu
        quantization_config=bnb_config
    )
else:    # Load previously checkpoint-saved model to continue fine-tune or evaluate
    model = AutoPeftModelForCausalLM.from_pretrained(
        from_checkpoint,
        torch_dtype=compute_dtype,
        return_dict=False,
        low_cpu_mem_usage=True,
        device_map=device,  # "auto" may be slower because offload to cpu
    )

# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Fix missing pad_token if error
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token})
    model.resize_token_embeddings(len(tokenizer))
model.config.use_cache = False
model.config.pretraining_tp = 1
```

Training with QLoRA

PEFT (Parameter-Efficient Fine-Tuning) methods enable efficient adaptation of large pretrained models to downstream applications by only fine-tuning a small number of (extra) model parameters instead of all the model's parameters.

- LoRA (Low-Rank Adaptation) works by attaching extra trainable parameters into a model and decomposing a large weight matrix into two smaller, low-rank matrices (called update matrices). These new matrices can be trained to adapt to the new data while keeping the overall number of changes low.
- QLoRA by Tim Dettmers et al (2023) combines quantization of a model to 4-bits and inserting a small set of trainable low-rank adaptation (LoRA) weights to enable efficient training.

A model is prepared for training with LoRA by wrapping the base model with a PEFT configuration in a SFTTrainer (Supervised Fine-tuning Trainer) class:

- r — attention dimension (the “rank”)
- lora_alpha — scaling factor for the weight matrices, a higher alpha assigns more weight to the LoRA activations
- lora_dropout — The dropout probability for Lora layers.

```
# Set config for PEFT
peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)
```

```
# Set config for SFT Trainer
trainer = SFTTrainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=dev_data,
    peft_config=peft_config,
    dataset_text_field="prompt",
    tokenizer=tokenizer,
    args=training_args,
    packing=False,
    max_seq_length=1024,
)
```

```
lap: 100%|██████████| 11232/11232 [00:00<00:00
lap: 100%|██████████| 1395/1395 [00:00<00:00,
```

```
if do_train:
    for _ in range(10):    # reclaim memory b
        with torch.no_grad():
            torch.cuda.empty_cache()
    gc.collect()
    training_stats = trainer.train()
```

Generate model predictions

- Create prompt with instruction, context (premise) and query (hypothesis):

```
# format premise and hypothesis as chat prompt
def as_test_prompt(ex):
    """prompt for response to test example"""
    return f"""
            Use the following context to determine if the factuality of the
            statement enclosed in square brackets at the end is entailment,
            neutral, or contradiction, and return the answer in 1 word as
            "entailment" or "neutral" or "negative":

            {ex['sentence1']}

            [{ex['sentence2']}]

            Answer:
            """.strip()

# format premise and hypothesis with gold label as training example
def as_prompt(ex):
    """training example"""
    return f"{as_test_prompt(ex)}  {ex['gold_label']}".strip()
```

```
# reformat all examples as prompts
X_train = pd.DataFrame(X_train.apply(as_prompt, axis=1), columns=["prompt"])
train_data = Dataset.from_pandas(X_train)
X_dev = pd.DataFrame(X_dev.apply(as_prompt, axis=1), columns=["prompt"])
dev_data = Dataset.from_pandas(X_dev)
X_test = pd.DataFrame(X_test.apply(as_test_prompt, axis=1), columns=["prompt"])
test_data = Dataset.from_pandas(X_test)
```

- Generate response from model:

```
def predict(X_test, model, tokenizer):
    """Generate model predictions on test set"""
    y_pred = []
    for i in tqdm(range(len(X_test))):
        prompt = X_test.iloc[i]["prompt"]
        input_ids = tokenizer(prompt, return_tensors="pt").to("cuda")
        outputs = model.generate(**input_ids,
                                 max_new_tokens=4,  # 8
                                 do_sample=False,   # True
                                 temperature=0.00,  # 0.01
        )
        result = tokenizer.decode(outputs[0][len(input_ids[0]):])
        answer = result.lower() # result.split("=")[-1].lower()
        if "entailment" in answer:
            y_pred.append("entailment")
        elif "contradiction" in answer:
            y_pred.append("contradiction")
        else:
            y_pred.append("neutral")
    return y_pred
```

Responses before fine-tuning

Before the pre-trained models were fine-tuned, they could not respond correctly to the downstream NLI task:

- Given the same prompt instruction, they did not produce an answer as required.
- Overall accuracy was no better than random responses.

```
# Evaluate pre-trained that has *NOT* been tuned for downstream NLI task
y_pred = predict(X_test, model, tokenizer)
evaluate(y_test, y_pred, verbose=True)

prompt = X_test.iloc[42]["prompt"]
print('Prompt:', prompt)

input_ids = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = model.generate(**input_ids,
                         max_new_tokens=4,
                         do_sample=False,
                         temperature=0.00
                         )
result = tokenizer.decode(outputs[0][len(input_ids[0]):])
answer = result.lower()
print('Answer:', answer)
```

```
Accuracy: 0.333

Classification Report:
              precision    recall  f1-score   support

contradiction       0.00      0.00      0.00       474
   entailment       0.00      0.00      0.00       474
      neutral       0.33      1.00      0.50       474

     accuracy                           0.33      1422
    macro avg       0.11      0.33      0.17      1422
 weighted avg       0.11      0.33      0.17      1422


Confusion Matrix:
[[  0   0 474]
 [  0   0 474]
 [  0   0 474]]
Prompt: Use the following context to determine if the factuality of the
            statement enclosed in square brackets at the end is entailment,
            neutral, or contradiction, and return the answer in 1 word as
            "entailment" or "neutral" or "negative":

            He could think of what he wanted to say but was having trouble
getting the words out.

            [ The patient is having trouble speaking. ]

            Answer:
Answer:

the factu
```

Evaluation of fine-tuned models

After fine-tuning, the models are evaluated based on accuracy of test set predictions:

- gemma-7b performed best (accuracy = 0.878), slightly ahead of runner-up Llama-2-13b (0.870)
- both pairs of open-source LLMs, gemma (2024) and Llama-2 (2023), outperformed an early encoder transformers-model: electra-base-discriminator (2020) – see Appendix

gemma-7b-it

Accuracy: 0.878

Classification Report:

	precision	recall	f1-score	support
contradiction	0.92	0.94	0.93	474
entailment	0.86	0.87	0.86	474
neutral	0.85	0.82	0.84	474

gemma-2b-it

Accuracy: 0.817

Classification Report:

	precision	recall	f1-score	support
contradiction	0.87	0.88	0.87	474
entailment	0.81	0.78	0.79	474
neutral	0.77	0.80	0.79	474

Llama-2-13b-chat

Accuracy: 0.870

Classification Report:

	precision	recall	f1-score	support
contradiction	0.92	0.94	0.93	474
entailment	0.86	0.85	0.85	474
neutral	0.83	0.82	0.83	474

Llama-2-7b-chat

Accuracy: 0.861

Classification Report:

	precision	recall	f1-score	support
contradiction	0.91	0.91	0.91	474
entailment	0.83	0.86	0.85	474
neutral	0.84	0.80	0.82	474

electra-base-discriminator

Accuracy: 0.809

Classification Report:

	precision	recall	f1-score	support
contradiction	0.87	0.89	0.88	462
entailed	0.78	0.77	0.78	481
neutral	0.77	0.76	0.77	479

Task 2: Medical QA

QA in the medical domain

The question answering (QA) NLP task challenges a model to respond to an input question. Closed-domain systems answer the questions from a specific context, while open-domain systems are based on broad unrestricted knowledge.
Jin et al (2019) introduced PubMedQA, a biomedical question answering (QA) dataset collected from PubMed abstracts. The task of PubMedQA is to answer research questions with yes/no/maybe using the corresponding abstracts. PubMedQA has 1000 expert-annotated, 61,200 unlabeled and 211,300 artificially generated QA instances.

- Two annotators (both qualified M.D.candidates) labeled 1000 instances from pre-PQA-U with yes/no/maybe to build PQA-L

PubMedQA is split into three subsets: labeled, unlabeled and artificially generated. They are denoted as PQA-L(abeled), PQA-U(nlabeled) and PQA-A(rtificial), respectively. We show the architecture of PubMedQA dataset in Fig. 2.

Statistic	PQA-L	PQA-U	PQA-A
Number of QA pairs	1.0k	61.2k	211.3k
Prop. of yes (%)	55.2	–	92.8
Prop. of no (%)	33.8	–	7.2
Prop. of maybe (%)	11.0	–	0.0
Avg. question length	14.4	15.0	16.3
Avg. context length	238.9	237.3	238.0
Avg. long answer length	43.2	45.9	41.0

PubMedQA dataset

- Clone data from repo at https://pubmedqa.github.io
- Read from json files: each instance is composed of (1) a question which is either an existing research article title or derived from one, (2) a context which is the corresponding abstract without its conclusion, (3) a long answer which is the conclusion of the abstract, and (4) a yes/no/maybe answer which summarizes the conclusion

Load pubmedqa articles and questions

```
pqal = pd.read_json('pubmedqa-master/data/ori_pqal.json', orient='index')
print('CONTEXT:', "\n".join(pqal.iloc[0]['CONTEXTS']))
print('QUESTION:', pqal.iloc[0]['QUESTION'])
print('ANSWER:', pqal.iloc[0]['final_decision'])
pqal['final_decision'].value_counts()
```

```
CONTEXT: Programmed cell death (PCD) is the regulated death of cells within an organism. The lace plant (Aponogeton madagascariensis) produces pe
rforations in its leaves through PCD. The leaves of the plant consist of a latticework of longitudinal and transverse veins enclosing areoles. PC
D occurs in the cells at the center of these areoles and progresses outwards, stopping approximately five cells from the vasculature. The role of
mitochondria during PCD has been recognized in animals; however, it has been less studied during PCD in plants.
The following paper elucidates the role of mitochondrial dynamics during developmentally regulated PCD in vivo in A. madagascariensis. A single a
reole within a window stage leaf (PCD is occurring) was divided into three areas based on the progression of PCD; cells that will not undergo PCD
(NPCD), cells in early stages of PCD (EPCD), and cells in late stages of PCD (LPCD). Window stage leaves were stained with the mitochondrial dye
MitoTracker Red CMXRos and examined. Mitochondrial dynamics were delineated into four categories (M1-M4) based on characteristics including distr
ibution, motility, and membrane potential (ΔΨm). A TUNEL assay showed fragmented nDNA in a gradient over these mitochondrial stages. Chloroplasts
and transvacuolar strands were also examined using live cell imaging. The possible importance of mitochondrial permeability transition pore (PTP)
formation during PCD was indirectly examined via in vivo cyclosporine A (CsA) treatment. This treatment resulted in lace plant leaves with a sign
ificantly lower number of perforations compared to controls, and that displayed mitochondrial dynamics similar to that of non-PCD cells.
QUESTION: Do mitochondria play a role in remodelling lace plant leaves during programmed cell death?
ANSWER: yes
```

Retrieval-Augmented Generation

RAG

- An LLM may not be aware of specific content, which may be proprietary or recent, absent in its pre-training data.
  - Meta’s prompting guide: “Employing RAG is more affordable than fine-tuning, which may be costly and negatively impact the foundational model's capabilities.”
- Retrieval Augmented Generation does not require model fine-tuning; it provides an LLM with additional context that is retrieved from relevant data (may be internal or external such as web-scraped) so that it can generate a better-informed response. As illustrated in the diagram:
  - External data is converted into embedding vectors with a separate model and kept in a vector database. Updating the embedding vectors on a regular basis is faster, cheaper, and easier than fine-tuning.
  - Given an unstructured query, retrieve the most relevant (similar) documents (vectors).
  - Pass the original question along with the retrieved context documents to the generation step.
  - With the added context, the same exact LLM model provides a much more relevant and informed answer.

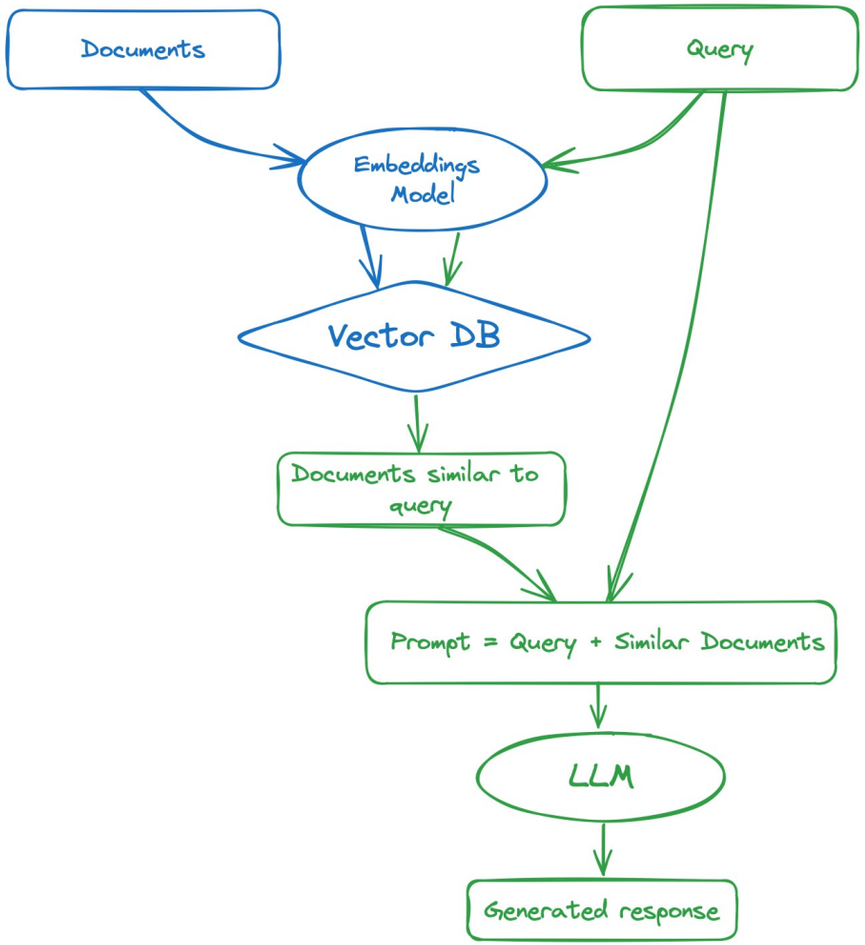


Image: https://huggingface.co/learn/cookbook/rag_zephyr_langchain

Chroma vector database

A Chroma database comprises one or more collections, each of which consists of many individual text documents stored as embedding vectors.

Load chroma client and its default embedding model

```
# Initialize ChromaDB client
db_path = "./database"   # local path to save database
chroma_client = chromadb.PersistentClient(path=db_path)
```

```
cprefix = 'pubmedqa_'
for col in chroma_client.list_collections():   # delete old collections
    if col.name.startswith(cprefix):
        chroma_client.delete_collection(col.name)
collections = dict()      # dict for the new collections to create
```

Create database collection with default embedding model

```
# embedding function that embeds any input text
# metadata states how this database should compute similarities (l2 is default)
# default embedding model is all-MiniLM-L6-v2
default_name = "all-MiniLM-L6-v2"
collections[cprefix + default_name] = chroma_client.get_or_create_collection(
    name=f"{cprefix}{default_name}",
    metadata={"hnsw:space": "cosine"}, # l2 is the default
```

Load and query documents

- Convert text documents as well as queries to word embedding vectors
- Fast retrieval of the top documents that are most similar to the query

See Chroma API reference at https://docs.trychroma.com/api-reference

Load data into database

```
# Each document is a context text string
documents = pqal['CONTEXTS'].apply(lambda x: "\n".join(x)).to_list()
np.max([len(doc) for doc in documents])
```

```
2725
```

```
# Each id is the row number
ids = [str(i) for i in range(len(pqal))]
#metadatas = [{'row': i} for i in ids]
```

```
# Load all documents
collections[cprefix + default_name].add(
    documents=documents,
    ids=ids,
)
```

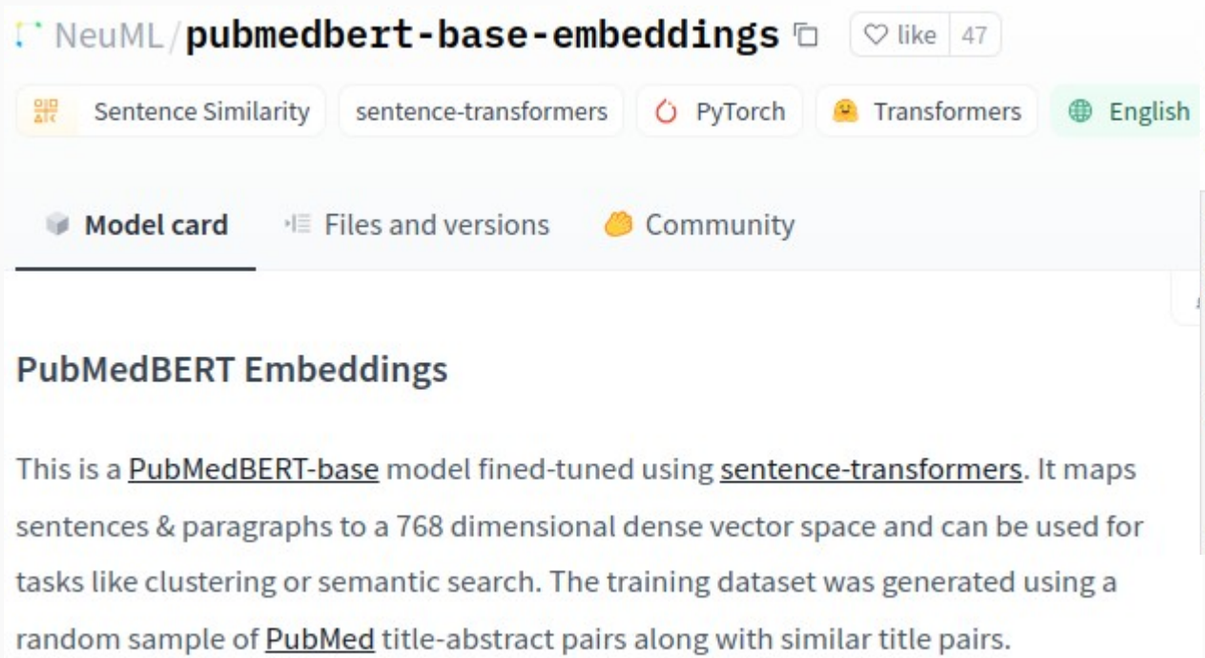
Query and retrieve an example

```
# Each query is a question text string
queries = pqal['QUESTION'].to_list()
gold = 13
result = collections[cprefix + default_name].query(query_texts=queries[gold],
                                                   n_results=1)
print('Gold:', gold, '   Retrieved:', result['ids'])
```

```
Gold: 13    Retrieved: [['13']]
```

Word embeddings

- Chroma uses the default embeddings model: “all-MiniLM-L6-v2”
- Can be any SentenceTransformers model: e.g. “pubmedbert-base-embeddings”



Can be any sentence-transformer model

see https://www.sbert.net/docs/pretrained_models.html

```
# Select other sentence-transformer model by name
other_name = 'NeuML/pubmedbert-base-embeddings'
other_id = other_name.split('/')[-1]
other_model = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name=other_name)

# Create collection with custom embedding function
collections[cprefix + other_id] = chroma_client.get_or_create_collection(
    name=f"{cprefix}{other_id}",
    metadata={"hnsw:space": "cosine"}, # l2 is the default
    embedding_function=other_model,
)
```

- Or implement custom embeddings (inherit from EmbeddingFunction class), e.g.
  - `https://github.com/stephenc222/example-chroma-vector-embeddings`

Retrieval accuracy

For each question, compare whether the embeddings vector for a query sentences retrieves, as the closest, its true paired PubMed document stored in the vector database. Results found that:

- "pubmedbert-base-embeddings" was more accurate than the default model "all-MiniLM-L6-v2"
- "gte-base" embeddings model was most accurate

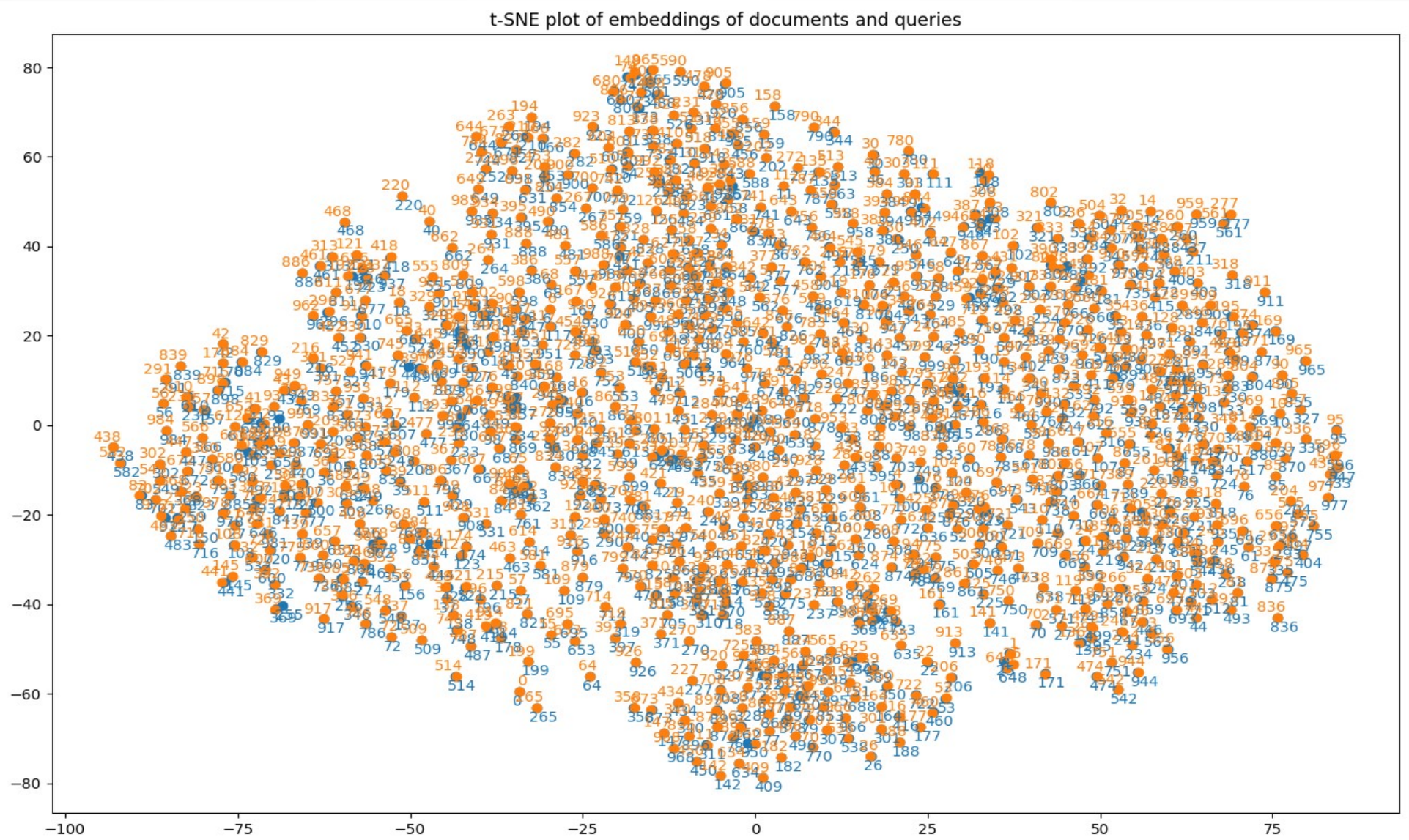
Overall, retrieval accuracy scores are high (>96%); as shown in t-SNE plot (next slide), the embedding vectors of true pairs of documents and queries are grouped close together.

```
# Check accuracy of id of retrieved document given query
accuracy = {}
for name, collection in collections.items():
    y_pred = []
    for row, query_text in tqdm(enumerate(queries)):
        result = collection.query(query_texts=query_text, n_results=1)
        #print(row, result['ids'][0])
        y_pred.append(int(result['ids'][0][0]))
    accuracy[name] = accuracy_score(y_true=list(range(len(queries))), y_pred=y_pred)
pd.DataFrame.from_dict(accuracy, orient='index', columns=['Accuracy'])
```

```
1000it [00:24, 40.94it/s]
1000it [00:27, 36.22it/s]
1000it [00:27, 36.71it/s]
```

	Accuracy
pubmedqa_all-MiniLM-L6-v2	0.966
pubmedqa_pubmedbert-base-embeddings	0.977
pubmedqa_gte-base	0.988

t-SNE plot


t-SNE plot of embeddings of documents and queries
80
60
40
20
0
−20
−40
−60
−80
−100
−75
−50
−25
0
25
50
75

Prompt = Query + Similar Documents

Prompts

Construct prompt with instruction + retrieved document + question (Llama-2 and Gemma required slightly different instructions) and ask for answer.

```
def prepare_prompt_llama(question, context=''):
    """Create prompt for llama model"""

    if context:    # with context
        return f"""
Use the following text to return a concise answer to the question at the end

Text:
{context}

Question:
{question}

Answer:
        """.strip()
    else:         # without context
        return f"""
Question:
{question}

Answer:
        """.strip()
```

```
def prepare_prompt_gemma(question, context):
    """Create prompt for gemma model"""

    if context:   # with context
        return f"""
Use the following text to return an answer to the question.
Return a concise answer to the question in 1 word.

Text:
{context}

Question:
{question}

Answer:
        """.strip()
    else:         # without context
        return f"""
Return a concise answer to the question in 1 word.

Question:
{question}

Answer:
        """.strip()
```

Generate model response

To generate response

```
DEFAULT_ROLE = "You are a helpful AI assistant."
DEFAULT_ROLE = "You are a helpful medical knowledge assistant."
MAX_NEW_TOKENS = 8
TEMPERATURE = None
```

```
def chat(prompt, role,  model, tokenizer, max_new_tokens=MAX_NEW_TOKENS,
         verbose=False, skip_special_tokens=True):
    """Prompt the model and return response"""

    # always prepend with a system prompt
    input_ids = tokenizer(f"{role}\n{prompt}\n".strip(),
                          return_tensors="pt").to("cuda")
    input_len = len(input_ids[0])
    with torch.no_grad():
        outputs = model.generate(**input_ids,
                                 max_new_tokens=max_new_tokens,
                                 top_p=None,
                                 do_sample=bool(TEMPERATURE),
                                 temperature=TEMPERATURE,
        )
    if verbose:
        print('Input tokens:', input_len,
              '  Output tokens:', len(outputs[0]))
    answer = tokenizer.decode(outputs[0][input_len:],
                              skip_special_tokens=skip_special_tokens)
    del [input_ids, outputs]  # reclaim memory
    empty_cache()
    return answer.lower()
```

Example of a prompt, and model response

```
# Example of a prompt and response
row = 13
question, context, gold = queries[row], documents[row], y_true[row]
print(f"Model={model_name}.  Example #{row} with gold:", {gold})
role = ""

# without context
prompt = prepare_prompt(question, context="", model_name=model_name)
print('NO CONTEXT:')
print(chat(prompt, role=role, model=model, tokenizer=tokenizer, verbose=True))
print('-----------------------------------')

# with context
print('WITH CONTEXT:')
prompt = prepare_prompt(question, context=context, model_name=model_name)
print(chat(prompt, role=role, model=model, tokenizer=tokenizer, verbose=True))
print('-----------------------------------')
```

```
Model=google/gemma-7b-it.  Example #13 with gold: {'no'}
NO CONTEXT:
Input tokens: 40   Output tokens: 43

yes
-----------------------------------
WITH CONTEXT:
Input tokens: 367   Output tokens: 375

no

the text does not describe
-----------------------------------
```

Evaluate accuracy

Finally, evaluate accuracy score of all the models' responses:

- Without any context paragraph, performance was poor – Gemma-7b-it (0.551), Llama-7b-chat (0.316). This experiment corresponds to an "open-domain QA" task.
- Given the true context paragraph, Gemma-7b-it (0.679) was slightly more accurate than Llama-7b-chat (0.661). This experiment corresponds to a "closed-domain QA" task.
- When context documents that are most similar to the query string are retrieved with RAG from a vector database: the "gte-base" embeddings model supported the best accuracy, followed closely by the "pubmedbert-base-embeddings" model.

	google/gemma-7b-it	meta-llama/Llama-2-7b-chat-hf
pubmedqa_all-MiniLM-L6-v2	0.672	0.652
pubmedqa_pubmedbert-base-embeddings	0.676	0.658
pubmedqa_gte-base	0.678	0.661
with-gold	0.679	0.661
no-context	0.551	0.316

Conclusion

Conclusion

This tutorial explored the use of open-source LLMs for healthcare NLP tasks, specifically for PubMed question answering and Medical natural language inference, and demonstrated how to overcome these hurdles:

1) If the model has not been instructed for the required downstream task, then fine-tuning can enhance its performance.
2) When the model was pre-trained with stale or non-relevant data for the task, then retrieval-augmented generation (RAG) can be an efficient solution.
3) On consumer devices like laptops, techniques such as quantization and low-rank adaption can be used to load, train and run the models.

We showed how to apply these methods on the popular Llama-2 (Meta AI, 2023) as well as the recently-released Gemma (Google, February 2024) models – the 2B version of the latter has a footprint small enough to potentially fit in mobile devices. Such approaches towards customizing open-source LLM’s are particularly important for conversational and textual applications in the healthcare domain, where privacy, security and mobility are critical concerns.

Appendix

Transformers on Hugging Face

Hugging Face is an online platform with over 350k models and 75k datasets, all open source and publicly available:

- Download pre-trained models from its Model Hub
- Import its Transformers library for training or inference

Example: Electra (2020) is an early encoder-only transformer model, suitable for simple tasks that require understanding input such as sentence classification.

google/**electra-large-discriminator** ♡ like 9

Transformers | PyTorch | TensorFlow | JAX | English | electra | pretraining

Model card | Files and versions | Community 1

ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators

ELECTRA is a new method for self-supervised language representation learning. It can be used to pre-train transformer networks using relatively little compute. ELECTRA models are trained to distinguish "real" input tokens vs "fake" input tokens generated by another neural network, similar to the discriminator of a GAN. At small scale, ELECTRA achieves strong results even when trained on a single GPU. At large scale, ELECTRA achieves state-of-the-art results on the SQuAD 2.0 dataset.

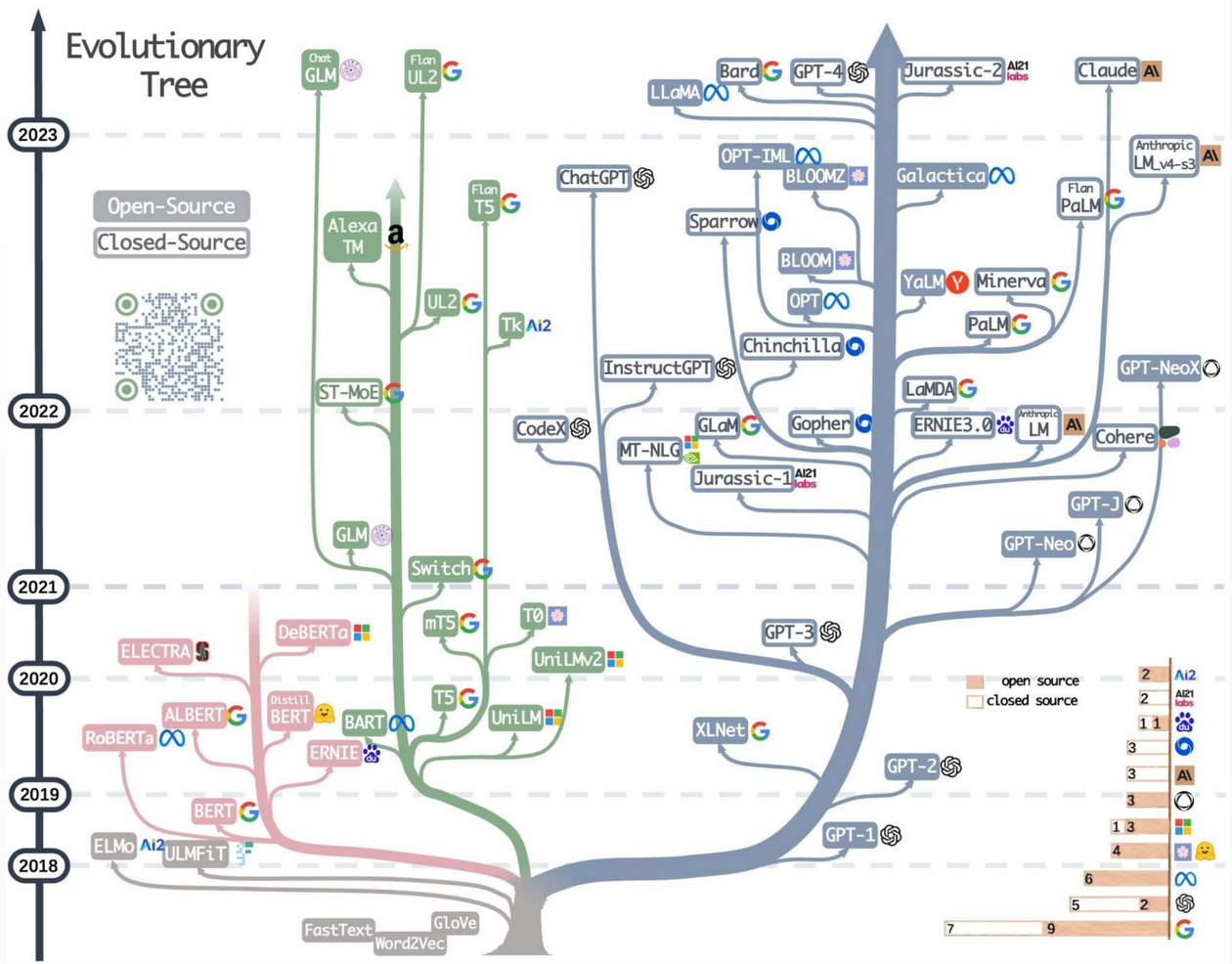


Image: https://www.baeldung.com/cs/bert-vs-gpt-3-architecture

Datasets and AutoTokenizer

Datasets is a Hugging Face library for accessing and sharing datasets, and for data preprocessing (tokenizing, resampling and transformations).

The AutoTokenizer.from_pretrained() class method will be instantiated with one of the tokenizer classes for preparing inputs for a model. The library contains tokenizers for all the models.

```
nli_labels = {'entailment': 0, 'neutral': 1, 'contradiction': 2}
if args.dataset == 'snli':
    dataset = datasets.load_dataset('snli')
else:
    dataset = datasets.DatasetDict()
    def prepare_label(ex):
        """To pre-process mednli dataset examples"""
        lab = ex['label']
        ex['label'] = nli_labels[lab] if lab in nli_labels else -1
        return ex

    for split in args.dataset.keys():
        print('Loading', split, ':', args.dataset[split], '...')

        # By default, "json" loader places all examples in the "train" split
        dataset[split] = datasets\
            .load_dataset('json', data_files=args.dataset[split])['train']\
            .rename_columns({'gold_label': 'label',
                             'sentence1': 'premise',
                             'sentence2': 'hypothesis'})\
            .select_columns(['label', 'premise', 'hypothesis'])\
            .map(prepare_label)
```

```
tokenizer = AutoTokenizer.from_pretrained(args.model_id, use_fast=True)
```

```
def prepare_dataset(examples, tokenizer=tokenizer, max_length=args.max_length):
    """Preprocess an NLI dataset, tokenizing premises and hypotheses"""
    max_length = max_length if max_length else tokenizer.model_max_length
    tokenized_examples = tokenizer(examples['premise'],
                                   examples['hypothesis'],
                                   truncation=True,
                                   max_length=max_length,
                                   padding='max_length')
    tokenized_examples['label'] = examples['label']
    return tokenized_examples
```

```
In [ ]:
# Prepare train and dev set splits
if training_args.do_train:
    train_dataset = dataset['train']
    if args.max_train_samples > 0:
        train_dataset = train_dataset.select(range(args.max_train_samples))
    train_dataset_tokenized = \
        train_dataset.map(prepare_dataset,
                          batched=True,
                          remove_columns=train_dataset.column_names)
    dev_dataset = dataset['dev']
    if args.max_dev_samples > 0:
        dev_dataset = dev_dataset.select(range(args.max_dev_samples))
    dev_dataset_tokenized = \
        dev_dataset.map(prepare_dataset,
                        batched=True,
                        remove_columns=dev_dataset.column_names)
```

https://huggingface.co/docs/datasets/en/index
https://huggingface.co/docs/transformers/main_classes/tokenizer

Auto Classes and TrainingArguments

Set TrainingArguments to configure the Trainer class:

- num_train_epochs — Total number of training epochs to perform
- per_device_train_batch — try to make this as large as you can without getting CUDA out-of-memory errors.
- eval_steps — Interval between two evaluations. Should be an integer as number of update steps, or a float in range [0,1) as ratio of total training steps.
- save_steps — Number of update steps before two checkpoint saves.
- logging_steps — Number of steps between two logs
- report_to — Platform to report the results and logs to, such as "azure_ml", "clearml", "codecarbon", "comet_ml", "dagshub", "dvclive", "flyte", "mlflow", "neptune", "tensorboard", and "wandb".

AutoModel classes automatically retrieve the relevant architecture with the right model fine-tuning head.

```
# Parameters for the Trainer
training_args = TrainingArguments(
    output_dir=args.output_dir,
    do_train=True,
    do_eval=True,
    num_train_epochs=8.0,
    per_device_train_batch_size=32,
    evaluation_strategy='steps',
    save_steps= 0.2,          # checkpoint interval
    logging_steps = 0.1,      # logging interval
    eval_steps = 0.1,         # evaluation interval
    report_to="tensorboard",
)
```

Load the model and tokenizer

Select to load model from HF hub, or previously checkpoint-saved folder This should either be a HuggingFace model ID (see https://huggingface.co/models) or a path to a saved model checkpoint (a folder containing config.json and model.save_tensors)

In [ ]:

```
# Select the model architecture
model_class = AutoModelForSequenceClassification

# Where to load model from
from_checkpoint = args.model_id     # load pre-trained from HF hub
#from_checkpoint = args.output_dir   # load from local folder
#from_checkpoint = os.path.join(args.output_dir, "checkpoint-6740")
```

In [ ]:

```
model = model_class.from_pretrained(from_checkpoint, num_labels=3)
```

Trainer class

Trainer is a complete training and evaluation loop implemented in the Transformers library. You only need to pass it the necessary pieces for training: model, tokenizer, dataset, evaluation function, and hyperparameters

- If you want to use custom evaluation metrics, provide your own compute_metrics() function

```
# If you want to use custom metrics, define your own "compute_metrics" function.
def compute_metrics(eval_prediction: EvalPrediction):
    """computes sentence-classification accuracy"""
    return {'accuracy': (np.argmax(eval_prediction.predictions, axis=1) ==
                         eval_prediction.label_ids).astype(np.float32).mean().item()}
```

```
# If you want to change how predictions are computed, you should
# subclass Trainer and override the "prediction_step" method
# (see https://huggingface.co/transformers/_modules/transformers/trainer.html#Trainer.prediction_step).
# If you do this your custom prediction_step should probably start by
# calling super().prediction_step and modifying the values that it returns
trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=train_dataset_tokenized,
                  eval_dataset=dev_dataset_tokenized,
                  tokenizer=tokenizer,
                  compute_metrics=compute_metrics)
```

```
if training_args.do_train:
    print('Training the model...')
    trainer.train()
    trainer.save_model(args.output_dir)
```

Evaluation with scikit-learn

Evaluating classification tasks:

accuracy_score()

- Accuracy: fraction of correct predictions
- Precision: number of true positives divided by the number of true positives plus the number of false positives
- Recall: number of true positives divided by the number of true positives plus the number of false negatives
- F1-score: harmonic mean of precision and recall

classification_report()

- Macro average: averages the unweighted mean per label
- Weighted average: averages the support-weighted mean per label

confusion matrix()

- C_{ij} is equal to the number of observations known to be in group *i* and predicted to be in group *j*

```python
# Calculate accuracy
accuracy = accuracy_score(y_true=y_true, y_pred=y_pred)
print(f'Accuracy: {accuracy:.3f}')

class_report = classification_report(y_true=y_true, y_pred=y_pred)
print('\nClassification Report:')
print(class_report)

# Generate confusion matrix
conf_matrix = confusion_matrix(y_true=y_true, y_pred=y_pred)
print('\nConfusion Matrix:')
print(conf_matrix)
```

```
Accuracy: 0.809

Classification Report:
              precision    recall  f1-score   support

contradiction       0.87      0.89      0.88       462
     entailed       0.78      0.77      0.78       481
      neutral       0.77      0.76      0.77       479

     accuracy                           0.81      1422
    macro avg       0.81      0.81      0.81      1422
 weighted avg       0.81      0.81      0.81      1422


Confusion Matrix:
[[412  23  27]
 [ 28 372  81]
 [ 34  79 366]]
```

References

References

- Singhal, K., Azizi, S., Tu, T. et al. Large language models encode clinical knowledge. Nature 620, 172–180 (2023). https://doi.org/10.1038/s41586-023-06291-2
- Romanov, Alexey and Shivade, Chaitanya, Lessons from Natural Language Inference in the Clinical Domain, 2018. http://arxiv.org/abs/1808.06752,
- Jin, Qiao and Dhingra, Bhuwan and Liu, Zhengping and Cohen, William and Lu, Xinghua. PubMedQA: A Dataset for Biomedical Research Question Answering. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, 2019
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela, Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, 2020, arXiv:2005.11401
- Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, LoRA: Low-Rank Adaptation of Large Language Models
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer, QLoRA: Efficient Finetuning of Quantized LLMs, 2023, arXiv:2305.14314
- https://www.kaggle.com/code/lucamassaron/fine-tune-gemma-7b-it-for-sentiment-analysis
- https://github.com/gregdurrett/fp-dataset-artifacts