Task 3 – LLM for MCQs

In data science, not all problems fit traditional models like classification or regression. Tasks involving nuanced language or semantic understanding require more than pattern recognition. In such cases, large language models (LLMs), with their ability to grasp context and intent, are essential. Data scientists must know when to move beyond conventional models and apply LLMs for deeper comprehension.

To understand the limits of traditional methods and the importance of semantic reasoning, you'll work with the SWAG dataset—over 113,000 multiple-choice questions based on real-world scenarios. This task will help you explore how LLMs excel at complex inference beyond simple patterns, reinforcing the value of foundation models in practical data science.

1. Hugging face access token

A Hugging Face access token is like a password that grants you permission to use Hugging Face services programmatically. It's a secure way to authenticate yourself when interacting with the Hugging Face API, model hub, or datasets.

You will need this for this project, so let's go through the process of creating one:

1. Go to https://huggingface.co and sign in or create a free account.
2. Create an account if you don't already have one. If you do, just login.
3. Click on your profile picture in the top-right corner.
4. Navigate to Settings → Access Tokens.
5. Click New Token.
6. Give it a name like SWAG_Project_Token.
7. Set the scope to write (since we are going to fine-tune a model later).
8. Click Generate Token and copy the token to your clipboard.
9. Your token should look like something like this: "hf_XXXXXXXXXXXXX"
10. You can use this token in your notebook one of the following ways:

```
from huggingface_hub import notebook_login, login

notebook_login()
login(token="hf_XXXXXXXXXXXXXXX")
```

We highly recommend using Kaggle Notebooks instead of Google Colab. Kaggle provides more reliable GPU access, fewer interruptions, and a smoother experience when working with large models.

2. Loading the dataset (2 points)

Begin by familiarizing yourself with the SWAG dataset through its official documentation on the Hugging Face Datasets Hub. For this task, you are required to load the standard version of the SWAG dataset into your notebook using the Hugging Face datasets library. This version includes a rich set of multiple-choice questions designed to evaluate contextual and commonsense reasoning.

3. Analyse the dataset (7 points)

Carefully review the official documentation of the SWAG dataset to gain a clear understanding of the meaning and purpose of each column. This foundational knowledge will enable you to more accurately interpret the behavior and results of language models applied to this dataset. (Do some minor EDAs)

4. Preprocess the dataset (16 points)

Write a preprocessing function that prepares the SWAG dataset for input into a language model. Your function should:

1. Replicate the sent1 (context) field four times—once for each of the four endings.
2. Combine the replicated context with sent2 to complete the sentence setup.
3. Concatenate the result with each of the four candidate endings (ending0 to ending3) to create four full input sequences per example.
4. Flatten the sequences and pass them through a Hugging Face tokenizer.
5. Unflatten the tokenized output so that each example retains a list of four candidate encodings.

This structure ensures that each data point is aligned with the four-way multiple-choice format expected by models like BERT or RoBERTa in multiple-choice classification tasks.

5. Load a tokenizer

To prepare the text inputs for your model, you now need to load a BERT tokenizer from the Hugging Face Transformers library. This tokenizer will be responsible for converting your sentence pairs (the contextual sentence starts and each of the four candidate endings) into token IDs that the model can understand. Use this version of the BERT tokenizer:

```
from transformers import AutoTokenizer
tokenizer =
AutoTokenizer.from_pretrained("google-bert/bert-base-uncased")
```

6. Apply the preprocessing function (5 points)

Apply your preprocessing function to the entire SWAG dataset using the dataset map method.

> 💡 **Hint: Set batched=True to enable batch processing and improve performance.**

7. Padding (5 points)

In natural language processing tasks, especially those involving multiple-choice questions like SWAG, different examples often have varying lengths. Padding all sequences in the dataset to the maximum possible length would be wasteful, leading to unnecessary computation and memory usage—especially when most sequences are much shorter.

By dynamically padding each batch to the length of the longest sequence within that batch, we ensure that each batch is as efficient as possible, without excessive padding. Hugging Face's DataCollatorForMultipleChoice automates this process for multiple-choice tasks. It:

- Flattens the input data,
- Pads the sequences to the appropriate length within each batch,
- And unflattens them so they can be fed directly into a model like BERT for multiple-choice classification.

8. Load the model

Load the model using the following code:

```
from transformers import AutoModelForMultipleChoice, TrainingArguments, Trainer

model =
AutoModelForMultipleChoice.from_pretrained("google-bert/bert-base-uncased")
```

9. Test the model on the dataset (10 points)

Using the validation split of the SWAG dataset, evaluate the model's ability to answer multiple-choice questions. Your task involves two parts:

1. Test Case Analysis:
   Select a single example from the validation set and pass it through the model. Carefully examine the model's predicted answer against the ground truth label. Discuss the correctness of the prediction, and reflect on any reasoning patterns (if discernible) from the model's output.

2. Comprehensive Validation Set Evaluation:
   Systematically process the entire validation set through the model. Define and compute at least one metric to quantify the model's performance across all examples.

10. Use in-context learning (ICL) (20 points)

In-Context Learning (ICL) is a method where a model is given examples of tasks directly within its input prompt, enabling it to learn from these examples without updating its weights. Instead of fine-tuning the model, we provide a few example question-answer pairs, and the model uses them as guidance to make predictions on new questions.

Your goal is to modify the model's input format to include a few-shot setup using ICL. In-Context Learning (ICL) techniques have diversified over time, enabling models to generalize across tasks by conditioning on examples provided in the prompt. One of the foundational works, *Language Models are Few-Shot Learners*, introduced the idea of few-shot ICL, where models adapt to new tasks by observing a few examples at inference time without fine-tuning. Building on this, *Large Language Models are Zero-Shot Reasoners* demonstrating that prompting models with "Let's think step by step" can unlock reasoning abilities in a zero-shot setting. *Rethinking Few-Shot Learning* analyzed the underlying mechanisms of ICL, revealing that models often rely on pattern recognition rather than true understanding. InstructGPT (*Training Language Models to Follow Instructions*) introduced instruction tuning, showing that aligning models with human intent improves ICL performance. Finally, *Transformers Learn In-Context by Gradient Descent* provided a theoretical perspective, suggesting that models simulate a form of gradient descent internally during ICL, framing it as a process of implicit learning.

You should choose 2 of the techniques mentioned above and process the validation set using your chosen ICL format and evaluate the model's performance again. Compare the two methods with each other and with the baseline model (without ICL).

Evaluate the model's prediction accuracy on the validation set under each scenario and report your findings clearly in a table.

Keep your implementation simple—focus on adapting the prompt structure and observing how the model's predictions change.

11. Fine-tune bert (30 points)

Now that you have your model set up, your next task is to fine-tune it using the SWAG training set.

What is Fine-Tuning?

Fine-tuning is the process of taking a pretrained model (like BERT) and continuing to train it on a specific task or dataset. While BERT is trained on large amounts of general text data, it has not seen the specific format or reasoning patterns required by the SWAG dataset. By fine-tuning, we adjust the model's internal weights to better align with the structure and distribution of the SWAG data, helping it learn to predict more accurate answers for this particular task.

Why is Fine-Tuning Useful?

Without fine-tuning, the model relies purely on its general knowledge, which might not perfectly match the SWAG task. Fine-tuning allows the model to:

- Adapt to the specific question/answer format in SWAG.
- Learn common patterns in the contexts and endings of the dataset.
- Improve performance beyond what is achievable with zero-shot or in-context learning alone.

For this project, you will fine-tune your BERT model using **LoRA (Low-Rank Adaptation)**, a powerful, parameter-efficient fine-tuning method.

What is LoRA?

LoRA (Hu et al., 2021) introduces small, trainable low-rank matrices into the model's architecture, allowing you to adapt the model to a new task without updating all of its parameters. This dramatically reduces the memory and computational costs of fine-tuning large models like BERT.

Using LoRA enables you to:

- Fine-tune large models efficiently, even on limited hardware (like Kaggle GPUs).
- Save time and memory compared to full model fine-tuning.

Gradient Accumulation

When working with large models and limited GPU memory, it's common to use gradient accumulation. Instead of updating the model after every mini-batch, you accumulate gradients over multiple smaller batches and update the weights after a larger "virtual" batch.

For example, if you want an effective batch size of 64 but can only fit 16 samples in memory, you can accumulate gradients over 4 steps before performing an update.

Here's how to proceed:

1. **Prepare the Data**
    - Use the SWAG training set (split="train") for training and the SWAG validation set (split="validation") for evaluation.
    - Apply your existing preprocessing pipeline to the data. Ensure that the model inputs—context, question, and multiple endings—are properly formatted.
2. **Fine-Tuning the Model**
    - Fine-tune the model using LoRA on the SWAG training set (split="train") for at least two epochs (feel free to train longer).
    - Use gradient accumulation to simulate a larger batch size.
    - Set a reasonable batch size per step (e.g., 8 or 16).
    - Set the learning rate (e.g., 2e-5, 1e-5, 2e-3).
    - Optionally, experiment with learning rate scheduling or early stopping for improved results.
3. **Evaluate the Model**

    After training, evaluate your model on the SWAG validation set using the following three key metrics:

    - Accuracy: The percentage of correctly predicted answers. This is your baseline performance measure for classification tasks.
    - Perplexity: Perplexity measures how well the model predicts the correct output, given the input. Lower perplexity indicates the model is more confident and accurate in its predictions. This metric is especially useful in language modeling tasks and for evaluating LLMs.
    - Confusion Matrix: A matrix showing how often each class is predicted correctly or misclassified. This helps identify specific patterns of errors and which classes the model struggles with.
4. **Report Your Results**
    - Provide your final validation accuracy.
    - Compare the fine-tuned model's performance to your earlier results from zero-shot and in-context learning setups.

This fine-tuning step allows the model to directly learn from the SWAG dataset, adapting its weights to better predict outcomes based on the data distribution.

12. In-Context Learning with the Fine-Tuned Model (10 points Bonus)

Now that you have fine-tuned your model on the SWAG dataset, let's explore how combining fine-tuning and in-context learning (ICL) affects performance.

Here's what you need to do:

- Apply the same ICL setup you used earlier (with a few-shot prompt of example question/answer pairs) but now use your fine-tuned model instead of the original, unmodified model.
- Evaluate the model's performance on the validation set again using the same metrics (accuracy, per-class accuracy, etc.).
- Compare the results to your earlier evaluations from:
    - Zero-shot (pretrained BERT, no fine-tuning, no ICL)
    - ICL with the pretrained model (before fine-tuning)
    - Fine-tuned model (without ICL)

13. Analyse the results (5 points)

You've now tried three approaches: fine-tuning, in-context learning, and combining both. Which one gave you the best results—and why do you think that happened?

For example, you might consider:

- Does the fine-tuned model still benefit from ICL, or is the effect less significant now?
- Could there be cases where ICL introduces noise or distracts a model that's already specialized on a task?
- In what kinds of real-world scenarios would combining fine-tuning and ICL be especially useful?

Questions

1. Fine-tuning a large model like BERT can be powerful but also resource-intensive. In what scenarios might fine-tuning be the best approach? When might you prefer ICL instead? Are there cases where you might combine both?
2. In your ICL prompts, you used examples from SWAG. If you had to design a custom ICL prompt for a different task (e.g., predicting medical diagnoses or financial trends), what kinds of examples would you choose? What factors would you consider when selecting them?