Tomoro assessment

Intro

I outline how I am planning to tackle this assessment. At a high-level, I want to educate myself with what the paper is trying to achieve. After I have a basic understanding of what the authors are doing, I want to familiarise myself with the data. With this, basic modelling can act as our benchmark - which we can then enhance to reach an improved model.

Attack plan:

1. Read paper
2. Investigate what data we are working with
3. Think about what modelling is relevant
4. Start writing utils
    - in parallel look at paper + deep-dive data examples
5. Basic benchmarking
6. Improvements
7. Validation

Preliminary paper overview

First pass of reading the paper. I present my notes below, for reference.

Authors create a new dataset, `ConvFINQA`

- Conversational question/answer type dataset
- Questions require multi-step reasoning, specifically within the context of maths/finance
- Specialised domain versus general domain reasoning
- Answering questions requires context which is either given as text or tables

Multihop reasoning is a difficult task to model

- These questions require multiple operations until you reach the answer
    - Example: what is the % increase of variable X w.r.t to its value last year?
        - you need to know the variable X given at the current and last year; then calculate a % change
- Two types of questions
    - Simple: single multi-hop question (single isolated question that requires multiple operations to answer)

    - Hybrid: multiple multi-hop questions (require cross-question reasoning/ dependencies)
- Different math operations
    - Add, subtract, divide, multiply, power

How modelling is approached

- Neural symbolic approach:
    - Combines a retriever to find the relevant facts/context; then a generator that uses question + context to decode the "reasoning program"
    - Reasoning program: They define it as a collection of operations. `op(arg1, arg2)`
        - think of this as a functional expression of the reasoning required to solve the
- Generative GPT like:
    - Relies on the context given during prompting "gold supporting facts"
        - Bad context, horrible output
    - Instructs the output to be like the reasoning program via examples
    - Investigates chain-of-thought

Key learnings:

- The model struggles with long reasoning chains.
- The model excels at number selection questions.
- The model suffers from the lack of domain knowledge.
- GPT-3 can do simple calculations by itself.
- GPT-3 performs better for its familiar program format.
- GPT-3 struggles with new complex task paradigms

Preliminary data overview

In this section, I describe what I've learned through a preliminary data exploration.

Note: Data exploration can be messy. For this reason, I separate the actual work into a different script. Here, I report what I've learned. For further detail, please look at `scripts/data_exploration.ipynb`

In [30]:
```
import pandas as pd

raw_data = pd.read_json('../data/train.json')
raw_data.iloc[0]
```

Out[30]:
```
pre_text      [26 | 2009 annual report in fiscal 2008 , reve...
post_text     [year ended june 30 , cash provided by operati...
filename                                  JKHY/2009/page_28.pdf
table_ori     [[, Year ended June 30, 2009], [2008, 2007], [...
table         [[2008, year ended june 30 2009 2008, year end...
qa            {'question': 'what was the percentage change i...
id                              Single_JKHY/2009/page_28.pdf-3
annotation    {'amt_table': '<table class='wikitable'><tr><t...
qa_0                                                        NaN
qa_1                                                        NaN
Name: 0, dtype: object
```

Must knows:

1. Questions are stored under `.qa` for simple questions (single multi-hop) or `.qa_0` and `.qa_1` for hybrid questions (cross-question dependencies)
2. Each entry has an associated `.table` ; `.pre_text` and `.post_text` attribute that is used to construct our "context" - relevant during prompting
    - Basically this would look like a concatenation of `pre_text` + `table` + `post_text`
3. Each entry has also an `.annotation` attribute which contains the major information for the conversations.
    - Different information is captured if the entry is a simple question versus a hybrid question
    - Importantly, contains the `.dialogue_break` attribute, which splits the question(-s) into simpler subquestions, used to reach the answer
    - Contains `.qa_split` to tell you which info corresponds to which question in the case of hybrid questions
        - for example, exe_ans_list = [0,0,0,1,1] can be read as the first three elements of the lists correspond to `.qa_0` and the latter two correspond to `.qa_1`

In [2]:
```
print('Short example:\n')
print(f'    .qa.question:\n     {raw_data.iloc[0].qa.get("question")}\n')
print(f'    .annotation.dialogue_break:\n     {raw_data.iloc[0].annotation.get("dialogu
print(f'    table:\n     {raw_data.iloc[0].table}\n')
print(f'    pre_text (excerpt):\n     {raw_data.iloc[0].pre_text[0][:100]}...\n')
print(f'    post_text (excerpt):\n     {raw_data.iloc[0].post_text[0][:100]}...\n')
```

```
Short example:

    .qa.question:
     what was the percentage change in the net cash from operating activities from 2008
to 2009

    .annotation.dialogue_break:
     ['what is the net cash from operating activities in 2009?', 'what about in 2008?',
'what is the difference?', 'what percentage change does this represent?']

    table:
     [['2008', 'year ended june 30 2009 2008', 'year ended june 30 2009 2008', 'year en
ded june 30 2009'], ['net income', '$ 103102', '$ 104222', '$ 104681'], ['non-cash expe
nses', '74397', '70420', '56348'], ['change in receivables', '21214', '-2913 ( 2913 )',
'-28853 ( 28853 )'], ['change in deferred revenue', '21943', '5100', '24576'], ['change
in other assets and liabilities', '-14068 ( 14068 )', '4172', '17495'], ['net cash from
operating activities', '$ 206588', '$ 181001', '$ 174247']]

    pre_text (excerpt):
     26 | 2009 annual report in fiscal 2008 , revenues in the credit union systems and
services business ...

    post_text (excerpt):
     year ended june 30 , cash provided by operations increased $ 25587 to $ 206588 for
the fiscal year e...
```

Important to note:

1. I've already found some bad examples - The data isn't perfect!
    - Cases where the maths operations don't seem to be correct
    - Some column names are repeated; effectively not being able to distinguish what numbers are
    - **This means that the accuracy metrics at the end should be taken with a grain of salt!**
2. We need to convert the table into something parsable by the LLM
    - Sometimes column names are supplied sometimes are not
    - I've spent sometime to think about it but there doesn't appear to be a global solution for this
    - You either assume that first entries are always column names or don't
    - Other option is to check the similarity of the first entries versus the rest of the table (in an attempt to understand if its a column name versus a cell value -- but this seems like a very computationally expensive operation)
3. Seems like the `pre_text` and `post_text` can be very noisy / dirty
    - Not always needed, but you can't know this apriori

- There's the option of the `.gold_inds` but this feels like cheating as its the perfect retrieval

Utils and functions

The relevant utilities can be found in the `utils` folder, which at a high level include the data preprocessing infra, model infra, and other evaluator functions.

Please look at each respective file for explanation of their contents - this report will not go through in detail, unless relevant/ interesting.

Please look into the utils folder as that contains the relevant functions + documentation

The folder is split into 4 scripts:

1. data_utils: Functions required for preprocessing and getting the data in the right format
2. model_utils: Functions for calling the models via the OpenAI SDK
3. eval_utils: Functions for validating the results
4. prompts: Where the system prompts are stored

Modelling, Prompting, Experiments

I want to write the basic functionality to parse all this information coherently, and run a few benchmarks. I'm thinking I'll take the API approach as my computer cannot run much...

I am relying to achieve most of the performance improvement through prompt engineering. I present here the thought process of how each prompt was structured and the underlying strategy.

I am using Deepseek's models as they are the cheapest at the moment of writing. I have allocated 5-10$ for this assessment and generate an API key to use their models. I do this by using OpenAI's SDK as suggested by the original Deepseek docs.

Model choice

I have chosen to use Deepseek's services to run my experiments:

- Cheap per token price
- Choice between standard chat-like model + reasoning model which is more in line with what we are trying to achieve

I've purchased a flat number of tokens and generated an API Key.

Should you want to run any experiments locally - please visit platform.deepseek.com and generate your key

There are two models available:

- `deepseek-chat`
- `deepseek-reasoner`

I cannot find explicit reference to what variants of Deepseek v3 and Deepseek R1 these are. However, I did find mentions that these are distilled versions and not the whole 7B param models. It is likely that API requests are being allocated to smaller models if the service is busy.

A quote I found:

```
deepseek-chat: Based on DeepSeek-R1-Lite-Chat, a distilled version
optimized for conversational tasks.
deepseek-reasoner: Based on DeepSeek-R1-Lite-Preview, a distilled
model fine-tuned for reasoning and complex problem-solving.

Both models are part of the 16B parameter "R1-Lite" family, which are
distilled from larger models (e.g., the original 7B-parameter R1
models)
```

Temperature

It is important to note, that we set the `temperature=0.0` as suggested by the original paper. It was however made apparent to us (from here: https://api-docs.deepseek.com/guides/reasoning_model) that the temperature is only adjustable on `deepseek-chat`. The reasoner model does not allow for the modification of it.

This is not perfect as we'd want a temperature of 0.0 when dealing with "math-like" problems. This might cause some downstream uncertainty or randomness in the results we see when using the `reasoner` model

Sequential questions and answers - collecting LLM history

As the exercises requires back and forth question/answer pairs, I've implemented a mechanism that keeps track of the system prompt, context, user questions and LLM answers.

I iteratively update the prompt to be keeping track of all the relevant history and pass it sequentially. This means that when the LLM answers the second and onward sub-questions of a given data entry -- it would have access to what was said and answered before.

This is done through my `add_past_responses` function. Some trial and error was necessary here to make sure this function is versatile for any model type (differences between `deepseek-chat` and `deepseek-reasoner`) but the final form does the following:

```
        inputs = [
            {"role": "system", "content": sys_prompt},
            {"role": "user", "content": _context},
            {"role": "assistant", "content": 'Context acknowledged,
awaiting for question.'},
            # Iteratively add a ({user, content}, {assistant,
content}) pair
            *add_past_responses(history),
            {"role": "user", "content": current_question},
        ]
```

Mathematics operators instead of direct evaluation

As the paper describes, they come up with a domain-specific language. We take upon that idea and make it our own.

The authors suggest, instead of asking the LLM to compute and evaluate the answer directly (effectively giving back a numeric value) - they instead force their model to output the answer as a collection of mathematical operations.

This means the final answer would look something along the lines of `add(a, b)` or more complex answers such as `multiply(a, subtract(b -c))` and so on.

We do this by defining 5 operators, `add`, `subtract`, `divide`, `multiply`, and `power`. We then code up an evaluator function that tries to see if the LLM output is parsable and if yes, if the answer is valid.

Please see more details on how these are coded in the `prompt` sections and also in the `utils.model_utils.py` file.

Prompt engineering

There are some basics that we want to force our LLM to do.

1. Use only the context it was provided and not rely on any external information
2. We want to enforce some output format to be able to easily extract the answer

The below prompts were iteratively generated. I started from number 01, and then progressively altered it.

Prompt 01

The first prompt, naively asks the LLM to give the numerical output. This was my first attempt to just get a basic understanding if the `utils` work in general and what the LLM can or cannot do.

I used a system prompt, asking the LLM to act as a financial analysis expert - this is because the questions revolve around finance, and if not directly relevant, still use mathematics. All of which are mostly found in financial data. This is our attempt to trigger the relevant weights that correspond to that type of "thinking".

It is important to note that asking the LLM to do maths directly isn't smart. This is apparent immediately, and it will be resolved but we are still in exploration mode. Even if its reasoning it correct, the actual evaluation of operations is unlikely to be correct. Purely out of how LLMs work underneath + the data it has seen.

```
SYSTEM_PROMPT_V1 = r"""
    Act as a financial analysis specialist. Your responses must:
    1. Strictly use only the contextual information provided by the
user
    2. Deliver the final answer in this exact format:
        - Unitless numerical value
        - Enclosed in \boxed{} LaTeX formatting

    Never reference external knowledge or assumptions.
    Convert all scaled values to absolute numbers during calculations,
    but omit units in the final answer.
    """
```

Prompt 02

Very similar to the first, still do not ask it to give maths operators - still evaluates the answer directly. The difference here is I want to extract its reasoning. This was an attempt to see what the output is like, and consider if I can use it to extract it for the user. The user likely wants to know where in the document that information is present.

This evolves a bit more, in the later stages so won't be described in detail here. Look for how `Structured outputs` can assist.

This allowed me to explore a bit how the thought process is presented/ structured.

```
SYSTEM_PROMPT_V2 = r"""
    Act as a financial analysis specialist. Your responses must:
    1. Strictly use only the contextual information provided by the
user
    2. Explicitly show your logical reasoning process through
sequential step-by-step explanations
    3. Deliver the final answer in this exact format:
        - Unitless numerical value
        - Enclosed in \boxed{} LaTeX formatting

    Never reference external knowledge or assumptions.
    Convert all scaled values to absolute numbers during calculations,
```

```
    but omit units in the final answer.
    """
```

Prompt 03

We now ask the LLM to use a selection of pre-defined maths operations, and NEVER evaluate the answer. This means we expect the LLM to give an output of looking like `add(a, b)` or similar.

For this to work, we slightly modify the prompt but keep all previous asks (use context only + ignore units + output format).

I've researched a bit on how to structure the language and landed on this approach. Language here can be varied and is up to the developer's approach. It does impact the LLM but it is difficult to know exactly what words/language will lead to the best output without iteratively testing each one.

Importantly:

1. I define the operators
2. I explicitly mention that I am expecting nested operators rather than new lines/ evaluated intermediate steps
3. I define that each operator ONLY takes two arguments
4. Give a short example

```
SYSTEM_PROMPT_V3 = r"""
    Act as a financial computation engine. Required behavior:
    1. Input Processing:
    - Use ONLY context provided in the query
    - Never incorporate external data or assumptions
    2. Calculation Methodology:
    - Perform and display calculations by using ONLY these Python-
style operators:
        - add(a, b) → a + b
        - subtract(a, b) → a - b
        - multiply(a, b) → a * b
        - divide(a, b) → a / b
        - power(a, b) → a^b
    - Each operator must have EXACTLY two arguments
    3. Output Requirements:
    - Final answer must be:
        - A nested combination of allowed operators
        - In unevaluated functional form
        - Expressed as \boxed{operator(...)} LaTeX
    - Include intermediate unit normalization calculations

    Example: For "Revenue per share - Cost per share = (5,000,000
revenue / 2,000,000 shares) - $5"
    Acceptable: \boxed{subtract(divide(5000000, 2000000), 5)}
```

```
    Unacceptable: \boxed{2.5 - 5} or \boxed{-2.5}
    """
```

With this, I write a function called `evaluate_maths` that will be taking the string output of the LLM, and using python's `eval` method to evaluate these operators. I also write a dictionary of these maths operators (exactly as seen in the prompt) for python to use.

Prompt 04: Structured output

Note: Structured output is usually coded by providing a json schema + pydantic objects. However, based on Deepseek's documentation, they expect this to be given during prompting.

See here: https://api-docs.deepseek.com/guides/json_mode

For this reason, I provide a prompt that handles structured outputs and will ~~consider if a pydantic implementation is required (if say I use other model providers)~~ *P.S; I did not use pydantic objects as I did not end up using another model provider.*

This was necessitated after a lot of investigation of how the outputs are produced.

Structured outputs is the mechanism of forcing your LLM to produce its output in a very defined way. The LLM is given a `json` schema and it's output will abide to it. With this, we can force specific attributes, for example quoting where the relevant text is, the thought process, and the final `program` to be evaluated. Without this approach, the LLM sometimes hallucinates. This is also a cause of "randomness" in a way (even if you set temperature to 0.0)

Finally, this prompt has embedded within it a one-shot example at the very end with the desired output.

```
SYSTEM_PROMPT_V4 = r"""
    Act as a financial computation engine that outputs valid JSON.
Required behavior:
    1. Input Processing:
    - Use ONLY context provided in the query
    - Never incorporate external data or assumptions
    2. Calculation Methodology:
    - Perform and display calculations by using ONLY these Python-
style operators:
        - add(a, b) → a + b
        - subtract(a, b) → a - b
        - multiply(a, b) → a * b
        - divide(a, b) → a / b
        - power(a, b) → a^b
    - Each operator must have EXACTLY two arguments
    3. JSON Output Requirements:
    - Structure response as valid JSON with this schema:
```

```
        {
            "user_question": "string",
            "user_context": "string",
            "reasoning": ["step1", "step2", ..., "stepN"],
            "final_answer": "boxed_expression"
        }
    - Maintain atomic values in JSON (no complex objects)
    - Escape special characters properly
    - final_answer must use: \boxed{operator(...)} format

    4. Compliance:
    - Strictly follow JSON syntax
    - No markdown formatting
    - No additional explanations outside JSON structure

    Example valid response:
    {
        "user_question": "Calculate profit per share given 5M revenue
and 2M shares with $5 fixed cost",
        "user_context": "[Row 1] Revenue: 5,000,000\n[Row 2] Shares:
2,000,000\n[Row 3] Fixed cost per share: 5",
        "reasoning": [
            "1. Revenue per share - Cost per share",
            "2. Convert 5M revenue to 5,000,000",
            "3. Divide revenue by shares: 5,000,000/2,000,000",
            "4. Subtract fixed cost per share from revenue per share",
            "4. Use subtract() for subtraction and divide() for
division"
        ],
        "final_answer": "\boxed{subtract(divide(5000000, 2000000),
5)}"
    }
    """
```

Runtime, async implementation, and tenacity

I am using the `deepseek-chat` and `deepseek-reasoner` models through the `OpenAI` SDK.

Runtime can vary GREATLY.

- On average, it takes around 8 minutes to run 100 example entries. These aren't 100 questions, rather, these are 100 entries within the `train.json` file with multiple associated questions
  - Sometimes, this can be as quick as 2 minutes, or as slow as 30-50 minutes on very rare occassions
  - My experimentation did not show a consistent behaviour to understand how long each execution would take, which hints that the slowdown could be on the `Deepseek` side
  - No limiters were hit, or other exception errors

- Sometimes, I was experiencing `ConnectionErrors` which were most likely caused by my VPN/ broadband provider. I have implemented a `@tenacity` version of the chat completion function, such that it would retry it up to a maximum of 3 tries if there were any errors.
- I have implemented an `async`-hronous variant of the main runner, to expedite the process
  - This is now the go-to runner, which you will see in the `model_run.ipynb`

Results

In the following few subsections I'm describing what type of behaviours I've seen when comparing model predictions versus the ground truth

Formatting mismatches between prediction and annotation answer

If you investigate the model predictions versus the annotated answers, one of the biggest sources of error originates from how the two values are given into two different formats.

Specifically, it is very frequently the case, that the model prediction describes the "correct" number, but does not match the exact format the annotation was given. Sometimes, the prediction is given as a percentage and the annotation is a decimal or vice versa.

It is not immediately the fault of the LLM or how we prompt it. There is a significant responsibility to how the `step_by_step_answers` are given in the original dataset. It is common that the "broken down" answers, do not match the "global" answer - this means that the question phrasing might be inconsistent with what we are evaluating against.

I've spent some time to write a quick fix around this - purely to understand how much we are missing out on. This is only for internal usage - the LLM still gets the answer wrong if we are being very strict.

- However, I'd rather our LLM be very rigid i.e. always give the answer in a set format and then we do our manipulation after.
- This allows us to operate under certainty in its behaviour rather than guessing what the format would be.

You can see the "fixed" error rates, under the `% correct (percentage fix)` label.

Issues with column names within the original annotation tables

Another important source of error, comes from the original annotation tables. It was made apparent through experimentation, that the actual column names of some entries are not explicit or can be confusing.

For example, read the following excerpt of the "golden" table. This is the perfect extraction of the relevant information needed to answer a particular question.

Note: I have not made any modifications to the below - this is directly extracted from the raw data

```
Prints the following
  ...
  the net cash from operating activities of year ended june 30
2009 2008 is $ 206588 ;
  the net cash from operating activities of year ended june 30
2009 2008 is $ 181001 ;
  ...
```

See how there are two separate sentences with the exact same row names / column names but with differing values. Even a human cannot know which one to choose.

This is purely a labelling issue.

In [ ]:
```
from utils.data_utils import process_data_table
# process_data_table is used for easy extraction of `gold_inds`
# no modification is done
all_prompts = process_data_table(raw_data)

print(all_prompts[0].get('gold_inds'))
```

```
2008 the net cash from operating activities of year ended june 30 2009 2008 is $ 206588
; the net cash from operating activities of year ended june 30 2009 2008 is $ 181001 ;
the net cash from operating activities of year ended june 30 2009 is $ 174247 ;
```

Issue with opening/closing parenthesis

Sometimes, the model answer is correct, but is unparasable because it is missing an opening or closing parenthesis.

It is easy to spot these cases via visual inspection, however, we cannot consider them correct. They are not parsable through our evaluator and would not work in a production pipeline.

I've spent some time to write code that infers which parenthesis is missing and append it - but this is not trivial and would require a significant effort. It seemed out of scope for my experimentation.

I assume that if you write something that can do the above, then there should be a significant yield in performance

Off by a magnitude of 10

Sometimes, the model answer is on the right track - but is off by a magnitude of 10. This is commonly the case when it is trying to do conversions or incorrectly adds a 0 to a number.

Again, we cannot consider these answers as correct - even though the model is *almost* right.

Metrics

There are multiple ways to measure the accuracy of our approach.

We defined a correct answer to be any answer that:

1. Is parsable through our evaluator: i.e. has a valid form, with valid maths operators
2. Is within 0.05 units of the annotated answer

We present the complete metrics below but a few comments that can be helpful for better undertanding:

1. Prompt changes have the biggest impact
    - Locking down the perfect prompt is not trivial but worth investing into
2. Even if the `deepseek-reasoner` model is supposedly more aligned with what we are after, limitations on setting its `temperature` parameter introduce randomness and decrease its performance
    - I was initially surprised at the reduced performance and ran the experiments multiple times
    - I've read its documentation and identified that the biggest source of uncertainty / differing behaviour between runs could be due to the `temperature` parameter
    - As we want to answer math-like questions, we want a low temperature which guarantees deterministic answers
3. Additionally, the `deepseek-reasoner` model does not accept JSON outputs/ structured outputs.
    - This again reduces its performance slightly as it has more variation in what it outputs
    - I've reworked my evaluator function and `find_answer` functions to try to catch the answer as best as possible - but there are very varied answers
        - At some point you'd want to classify these answers as wrong as if the format is too far off you can't expect good results when deployed in production
4. Using `step_by_step_questions` versus the "straight" question is an easier modelling task at the individual level but combining them together introduces uncertainty
    - Although performance DID increase when using the `step_by_step` approach - we still saw cases where the model "trips" up at an early stage and causes issues downstream
    - We initially thought that since each question is simpler, it would lead to more accurate global predictions

- However, as they are iteratively added, the model is still likely to make mistakes (wrong num of parenthesis, unparsable answers) at every step
- As there are now multiple steps to answer a question, there are more chances of making such mistakes
- There is a tradeoff here that must be made. If a question is very complex (and requires multiple steps) do you break it down? or do you supply the whole thing at once?

In [4]:
```
import os, sys

sys.path.append('..')
from utils.eval_utils import evaluate_experiment

print('Experiment 1: Evaluated mathematics directly')
metrics_exp1 = evaluate_experiment('../results/exp1.json')

print('Experiment 2: Evaluated mathematics directly + forced reasoning')
metrics_exp2 = evaluate_experiment('../results/exp2.json')

print('Experiment 3: Using maths operators + no evaluation')
metrics_exp3 = evaluate_experiment('../results/exp3.json')

print('Experiment 4: Using maths operators + no evaluation + step by step questioning')
metrics_exp4 = evaluate_experiment('../results/exp4.json')

print('Experiment 5: Using maths operators + no evaluation + structured outputs + one-s
metrics_exp5 = evaluate_experiment('../results/exp5.json')

print('Experiment 6: Using maths operators + no evaluation + structured outputs + one-s
metrics_exp6 = evaluate_experiment('../results/exp6.json')

print('Experiment 7: Using reasoning model + maths operators + no evaluation')
metrics_exp7 = evaluate_experiment('../results/exp7.json')

print('Experiment 8: Using reasoning model + maths operators + no evaluation + step by
metrics_exp8 = evaluate_experiment('../results/exp8.json')

print('Experiment 9: Using reasoning model + maths operators + no evaluation + step by
metrics_exp9 = evaluate_experiment('../results/exp9.json')
```

```
Experiment 1: Evaluated mathematics directly
Metrics:
            % correct: 0.451
            % correct (percentage fix): 0.451

Experiment 2: Evaluated mathematics directly + forced reasoning
Metrics:
            % correct: 0.534
            % correct (percentage fix): 0.548

Experiment 3: Using maths operators + no evaluation
Metrics:
            % correct: 0.408
            % correct (percentage fix): 0.465

Experiment 4: Using maths operators + no evaluation + step by step questioning
Metrics:
            % correct: 0.56
            % correct (percentage fix): 0.625

Experiment 5: Using maths operators + no evaluation + structured outputs + one-shot + s
tep by step questions
Metrics:
            % correct: 0.566
            % correct (percentage fix): 0.65

Experiment 6: Using maths operators + no evaluation + structured outputs + one-shot
Metrics:
            % correct: 0.363
            % correct (percentage fix): 0.492

Experiment 7: Using reasoning model + maths operators + no evaluation
Metrics:
            % correct: 0.508
            % correct (percentage fix): 0.548

Experiment 8: Using reasoning model + maths operators + no evaluation + step by step qu
estions
Metrics:
            % correct: 0.5
            % correct (percentage fix): 0.599

Experiment 9: Using reasoning model + maths operators + no evaluation + step by step qu
estions + short context
Metrics:
            % correct: 0.435
            % correct (percentage fix): 0.511
```

Our best performing model

Experiment 5 has shown the highest performance.

This is:

- Model is the `deepseek-chat` model
- Instructed to give an unevaluated answer using the maths operators
- Uses breakdown step-by-step questions
- Uses structured outputs
- Has a one-shot example of the desired behaviour

Deep dive on the best model

In [66]:
```python
import pandas as pd
import numpy as np
experiment_5 = pd.json_normalize(
    pd.DataFrame
    .from_records(metrics_exp5)
    .stack()
    )

# Slicing entries that are not parsable
unparsable = experiment_5[~experiment_5['parsable']]

# Slicing to wrong entries
wrong = experiment_5[experiment_5['close_match'] == 0.0]
# Slicing answers that are correct but needed a % fix
needed_fix = wrong[(wrong['close_match'] == 0.0) & (wrong['close_match_perc_fix'] == 1.

# Entries where the answer is parsable, but not accurate
# Trying to understand what the model is doing
others = experiment_5[(experiment_5['close_match_perc_fix'] == 0.0) & (experiment_5['pa
# others = experiment_5.iloc[~experiment_5.index.isin([needed_fix.index] + [unparsable.
```

In [57]:
```python
# Very little % of unparsable answers!
print(f'% of unparsable answers: {np.round(
    100 * unparsable.shape[0] / experiment_5.shape[0], 2
)}%')
```

```
% of unparsable answers: 2.02%
```

In [58]:
```python
# Approx. 8-10% need a fix for their format (percentage vs decimal)
print(f'% of answers that needed a fix for wrong percentage/decimal format: {np.round(
    100 * needed_fix.shape[0] / experiment_5.shape[0], 2
)}%')
```

```
% of answers that needed a fix for wrong percentage/decimal format: 8.42%
```

In [98]:
```python
# There are occurrences where the mismatch comes from unit conversion
# Very hacky way of getting a feel of how many entries are like this

# Remove 0s and decimals in an attempt to see if the actual "information" of the number
# between the prediction and the annotation answer
print(f'''Number of cases where mismatch is due to unit conversion: {
    # count how many cases are identical
```

```
    sum(
        # Remove 0s and remove . signs
        others['eval_ans'].astype(str).str.replace('0', '').str.replace('.', '')
        ==
        others['annot_ans'].astype(str).str.replace('0', '').str.replace('.', '')
        )
}''')
print('Example: (off by a factor of 1e9; billion)')
print(f'    {others.loc[287].eval_ans}')
print(f'    {others.loc[287].annot_ans}')
```

```
Number of cases where mismatch is due to unit conversion: 12
Example: (off by a factor of 1e9; billion)
    4400000000.0
    4.4
```

Ways to get better results

1. We did not perform any fine-tuning - If we did, we would surely reach the expert-level accuracy mentioned in the paper
    - By fine-tuning, we hope to mitigate occurences of wrong decimal/percentage formatting
    - We can add functionality of "Not enough information" catches such that the model stops from making a bad prediction
2. Better context extractor - we relied upon the data however we've seen that it can be very dirty/ noisy
    - Adding a tailored context extractor, can mimic the "golden" indicators and remove unnecessary noise
3. Column name predictor - we've seen cases where the column names in the original tables are duplicated / non-descriptive
    - Adding a module where it tests similarity between column name vs cells (to see if we are choosing the right value)
    - or try to predict what contents we are seeing by an NER approach (Named entity recognition) can help us generate column names for missing tables/columns
4. Reduce verbosity of model
    - This should reduce occurrences of hallucinations of the model
    - Be more concrete in its answers
5. Force specific formatting / LaTeX
    - Wrapper functions were required to extract the answer even with the usage of structured outputs
    - There were remnants of LaTeX that were removed by secondary functions
    - By tuning prompting or forcing pydantic objects the model wouldn't be able to output such results