Assignment 4: Fact-checking Outputs from ChatGPT

Academic Honesty: Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all work you submit must be your own!**

Goals The primary goal with this assignment is to give you hands-on experience analyzing outputs from large language models (LLMs) like ChatGPT. You will understand what sorts of non-factual outputs LLMs can generate, and what is involved in the process of verifying those outputs against Wikipedia. In addition, you will conduct some error analysis of the mistakes your fact-checking model makes.

Dataset and Code

IMPORTANT: Please use `Python3.10` and the specified version of packages listed in `requirements.txt`. Otherwise, your code might lead to issues with the autograder!

We have included a `requirements.txt` file. You can install the dependencies here by running:

```
pip install -r requirements.txt
```

Note that you can do this inside a virtual environment. For example, with conda you can create a virtual environment with specified Python versions by:

```
conda create -n A4Environment python=3.10
```

Data The dataset and overall setup for this project is taken from FActScore (Min et al., 2023).[1] FActScore investigates the problem of detecting errors in biographies of people generated by ChatGPT. The FActScore paper explores methods to decompose claims into facts, then searches over Wikipedia to find articles that (potentially) support those facts. They conduct human annotation of these steps to understand the performance of systems in different parts of this pipeline (decomposition and fact-checking). The decomposition and retrieval have been done for you.

The **facts** you are given are propositions extracted from ChatGPT outputs. These are simple natural language sentences. Although FActScore explores different ways of producing these facts from biographies, we use their *human-annotated* facts for our analysis, to remove any potential errors in this step. These facts are then validated against **passages** that are retrieved by Wikipedia. Each fact is hand-labeled by humans with either "S" for supported, "NS" for not supported, or "IR" for irrelevant. **However, we will only predict S vs. NS to simplify the task.**

Passages have been retrieved using BM25, a sparse retrieval model (i.e., one that does not use neural networks). The FActScore paper also discusses experiments with a dense retriever, but passages from this model only give slightly better performance.

In the provided Python code, `FactExample` stores the examples themselves. These consist of a fact (a string), passages (a list of dicts consisting of a title and text), and the label. Note that the passages themselves are not tokenized or preprocessed in any way.

The raw fact data is located at `data/dev_labeled_ChatGPT.jsonl` and look like this (note: this example is not in the dataset given to you):

```
{"input": "Question: Tell me a bio of Lanny Flaherty.",
 "output": "Lanny Flaherty is an American actor born on December 18, 1949, in
  Pensacola, Florida. He has appeared in numerous films, television shows, and
```

[1] Original GitHub: `https://github.com/shmsw25/FActScore`

```
  theater productions throughout his career, which began in the late 1970s.
[REMOVED]
"annotations": [
  {"text": "Lanny Flaherty is an American actor born on December 18, 1949, in
            Pensacola, Florida.",
   "is-relevant": true,
   "human-atomic-facts": [
     {"text": "Lanny Flaherty is an American.", "label": "S"},
     {"text": "Lanny Flaherty is an actor.", "label": "S"},
     {"text": "Lanny Flaherty was born on December 18, 1949.", "label": "NS"} [...]
```


Note that the `output` field is what ChatGPT generated about this person, which may or may not be correct. The `human-atomic-facts` are annotated facts with labels indicating whether each part of the output is true or not, according to humans who were able to consult Wikipedia.

The retrieved passages are in `passages_bm25_ChatGPT_humfacts.jsonl` and they look like this. Note that only a single passage was retrieved in this case, but the passages you have are generally more numerous. This is information from Wikipedia, so is likely to be more reliable than what the model generated.


```
{"name": "Lanny Flaherty",
 "sent": "Lanny Flaherty is an American.",
 "passages": [{"title": "Lanny Flaherty",
   "text": "<s>Lanny Flaherty Lanny Flaherty (born July 27, 1942) is an
   American actor.</s><s>Career. He has given his most memorable performances
   in \"Lonesome Dove\", \"Natural Born Killers\", \"\" and \"Signs\". Flaherty
   attended University of Southern Mississippi after high school. He also
   had a brief role in \"Men in Black 3\", and appeared as Jack Crow in Jim
   Mickles 2014 adaptation of \"Cold in July\". Other film appearances include
   \"Winter People\", \"Millers Crossing\", \"Blood In Blood Out\", \"Tom and
   Huck\" and \"Home Fries\" while television roles include guest appearances
   on \"The Equalizer\", \"New York News\" and \"White Collar\" as well as a 2
   episode stint on \"The Education of Max Bickford\" as Whammo.</s><s>Personal
   life. Flaherty resides in New York City.</s>"}]}
```


Framework code The framework code you are given consists of two files:

1. `facthecking_main.py`: This loads the data, instantiates the `FactChecker` model to use, executes it on the data, and reports accuracy, similar to the driver classes in previous projects. You will not be modifying this file.

2. `facthecker.py`: This file contains `FactExample`, `EntailmentModel`, and the `FactChecker` classes you will implement. This is similar to `models.py` in previous assignments.

Getting started Run:

```
python factchecking_main.py --mode random
python factchecking_main.py --mode always_entail
```

This will run two low-performing baselines and print results. You will implement stronger `FactChecker` models that will perform better!

Part 1: Word Overlap (40 points)

The first method we will explore for fact-checking is bag-of-words overlap. A supported fact might be expected to have high unigram overlap with a passage that supports it, and low overlap with passages that don't support it. This sounds like a somewhat naive baseline, but it's one that might work pretty well!

You should implement a method that predicts supported vs. not supported based on this criterion. You will have to decide on several details of your implementation. How do you want to handle tokenization? Do you want to stem? Do you want to remove stopwords? The decisions you are making here look similar to those in the bag-of-words method in Assignment 1, but what works best may be different!

Your method should eventually compute a real-valued score that you use to make your classification decision. This could be a cosine similarity of tf-idf vectors, Jaccard similarity of word sets, metrics based on precision/recall, existing lexical metrics like ROUGE or BLEU, or anything similar that computes a discrete (non-neural/embedding-based) word overlap score.

Your method should get at least 75% accuracy on the dataset to receive full credit. Try tuning the preprocessing method as well as changing the classification threshold to achieve the best performance. You can tune this threshold on the development set to optimize for accuracy. It would be best practice to tune this on a separate set, but we won't have separate dev and test sets here.

Part 2: Textual Entailment (40 points)

Second, you will explore a method that goes beyond the surface form of the words. The most appropriate type of system explored in prior work is textual entailment. These systems should theoretically be able to decide whether a fact (the "hypothesis") is logically entailed by a source document (the "premise").

For this part, you will use a pre-trained model, specifically a DeBERTa-v3 base model. This is a pre-trained instance of DeBERTa (He et al., 2020) fine-tuned on the MNLI (Williams et al., 2018), FEVER (Thorne et al., 2018), and ANLI (Nie et al., 2020). This model can take a premise-hypothesis pair and return a decision: entailment, neutral, or contradiction. Your task is to use this information to determine supported vs. not supported for the fact.

You can run

```
python factchecking_main.py --mode entailment --cuda
```

with the optional `--cuda` flag to do the entailment model queries on a GPU. You should do this only if you have Pytorch with CUDA installed and a GPU enabled. **Note that the autograder runs on CPU, so you will need to calibrate the runtime of your assignment on CPU as well.**

Using the entailment model The model itself is loaded in `factchecking_main.py`. In `factcheck.py`, the `EntailmentModel` class includes most of the boilerplate code needed to do inference with this model. However, we stop short of mapping logits to an actual prediction. You will have to decide what strategy makes the most sense for extracting a binary S/NS decision from a three-class entailment/neutral/contradiction decision. You can either use the discrete entailment decision, or you can derive a decision based on a threshold of one of the entailment/neutral/contradiction probabilities (or some combination thereof).

Entailment models are typically designed to take sentences as input, not entire passages.[2] Therefore, you will have to do some kind of sentence splitting and likely some "cleaning" as well, due to noise in the data

[2]The ANLI dataset does contain paragraphs as passages. However, in our experimentation, we did not find that feeding whole-passage premises into the model worked well. You are free to experiment with it, though!

introduced by Wikipedia. You should then loop over the sentences and systematically compare the fact to each sentence in the passages, then take the "max" over the passages.

If your implementation is uses the entailment classifier discretely (e.g., taking the returned entailment label), then the sentence can be considered entailed if it is entailed by *any* part of any of the passages. If your implementation uses entailment scores and sets a threshold, then the score for that fact should be the max over the entailment scores returned by running on any example in the passage. This resembles the method presented in Laban et al. (2022).

Your method should get at least 83% accuracy on the dataset to receive full credit.

Optimization Running the entailment model on all of the data may be slow. To speed things up, you can implement pruning based on the word overlap method. If there is very low word overlap, then entailment is extremely unlikely. You can discard examples that fail to meet a low word overlap threshold, then run entailment on the remaining instances.

Your code must run before the autograder times out; aiming to complete in around 10 minutes on your machine is a good target. Second, your code must run without causing out-of-memory (OOM) issues with the autograder. We included a code snippet to remove variables that are no longer needed and explicitly delete large objects using the `del` function and `gc.collect()` function. **Please use the specified versions of packages in `requirements.txt`; otherwise it might lead to out-of-memory or other issues with the autograder!**

Part 3: Error Analysis (20 points)

Part 3 involves a written submission. See the end of the assignment PDF for submission instructions.

Finally, you should conduct some error analysis on the results of the **entailment** model (your model from Part 2).[3]

There are two broad types of errors you should investigate. **False positives** are cases where the model predicted "supported" but the ground truth label is "not supported." **False negatives** are the opposite, where the model predicted "not supported" but the ground truth label is "supported."

You should manually examine **10 errors** of each type (10 false positives and 10 false negatives), or as many errors as your system makes if it does not make 10. You should categorize them into a set of a few **fine-grained categories** that you see as suitable. You should aim for 2-4 fine-grained error categories collectively between false positives and false negatives; you do not need to come up with many categories, but it's important to analyze the data accurately. **These categories should not just be "false positive" or "false negative"**, but should identify common language or other patterns in the examples where the model was wrong. To receive full credit for this part, your fine-grained categories should be non-trivial and the statistics and examples in your writeup should support your categorizations.

In your writeup, include:

1. Brief definitions of the fine-grained error categories you used in your analysis (1-2 sentences per category)
2. Aggregate statistics about the errors you examined: how many errors were of each fine-grained type?
3. **Three** examples discussed in more detail. For each of these, include: (a) the text of the example itself; (b) the ground truth label; (c) your model's predicted label; (d) the fine-grained error type you gave it; (e) 1-3 sentences describing why you think the fine-grained error type label applies.

[3] If you are not able to get Part 2 working, you are allowed to use the model from Part 1.

Deliverables and Submission

Written Submission You will upload your written submission as either a text file or a PDF on **Canvas** (submission available closer to the deadline). You will peer-review each others' written submissions.

Your peer assessment of others' work should address: (1) Was the error analysis completed as specified above? (2) Are you convinced by the analysis? Do the labels given and explanations make sense? You will answer these two questions but not necessarily assign a final grade. Your assessments will be given to the course staff, who will make the final decision about grades.

Code Submission You will upload your code for the first two parts on Gradescope in a single file.

Make sure that the following commands work before you submit:

```
python factchecking_main.py --mode word_overlap
python factchecking_main.py --mode entailment
```

These commands should run without error and train in the allotted time limits.

References

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. DeBERTa: Decoding-enhanced BERT with Disentangled Attention. *ArXiv*, abs/2006.03654.

Philippe Laban, Tobias Schnabel, Paul N. Bennett, and Marti A. Hearst. 2022. SummaC: Re-visiting NLI-based models for inconsistency detection in summarization. *Transactions of the Association for Computational Linguistics*, 10:163–177.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2023. FActScore: Fine-grained Atomic Evaluation of Factual Precision in Long Form Text Generation. *arXiv 2305.14251*.

Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2020. Adversarial NLI: A new benchmark for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4885–4901, Online, July. Association for Computational Linguistics.

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. FEVER: a large-scale dataset for fact extraction and VERification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 809–819, New Orleans, Louisiana, June. Association for Computational Linguistics.

Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*.