Assignment 5: 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**

Please use up-to-date versions of Python and PyTorch for this assignment. In addition, you will need to install the transformers package from Hugging Face as well as a few dependencies. 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.

If you are on the CS lab machines, you can use:

```
source /lusr/opt/miniconda/bin/activate cs371n-gpu
```

to access a virtual environment that should have all dependencies needed for the project. This includes a GPU-enabled version of Pytorch; for CPU, you can use:

```
source /lusr/opt/miniconda/bin/activate cs371n-cpu
```

**Data** The dataset and overall setup for this project is taken from FActScore (Min et al., 2023). 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/devlabeledChatGPT.jsonl and look like this (note: this example is not in the dataset given to you):

---

1Original GitHub: https://github.com/shmsw25/FActScore
Lanny Flaherty is an American actor born on December 18, 1949, in Pensacola, Florida. He has appeared in numerous films, television shows, and theater productions throughout his career, which began in the late 1970s.

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.

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

1. factchecking_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. factchecker.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 (35 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 (35 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. 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.

---

**Part 3: Error Analysis (30 points)**

Finally, you should conduct some error analysis on the results of the entailment model (your model from Part 2). You will submit a writeup of your findings. See the end of the assignment PDF for submission instructions.

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. For example, one kind of fine-grained error could be “the gold standard is wrong and the model is actually correct”. 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.”**

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.

“Average” solutions on this part will receive full credit. It benefits you to be thorough for your own learning and as preparation for the final project. But you should not feel like you need to pad your analysis.

---

\(^{3}\)If you are not able to get Part 2 working, you are allowed to use the model from Part 1.
with lots of additional words to try to be “thorough.” Instead, aim to be precise and descriptive without being verbose.

**OPTIONAL Part 4: Dependency Parsing (0 points)**

**This part is optional.**

A final method you can explore is dependency parsing. Dependency parsing can capture relationships between words and enable identification of syntactic relationships that go beyond the surface form. Broadly, we want to do something similar as word overlap, but look at dependency relations. Rather than checking if a source document contains the tokens *Flaherty* and *American*, we probably want to see whether *American* is actually being attributed to *Flaherty* for higher precision.

The easiest dependency parser to use is spaCy’s. It is invoked in `DependencyRecallThresholdFactChecker’s get_dependencies` method. This method takes a string as input and returns a set of useful dependencies to look at, expressed as (head, label, child) tuples. You can check the intersection of the set of arcs from the fact and the set of arcs from the retrieved documents in order to compute an overlap score, similar to word overlap.

See if you can get this working better than word overlap—but this might be challenging! Do some error analysis of both false positives and false negatives to profile why dependencies may not be as effective for this problem as we expect.

**Deliverables and Submission**

You will submit both your code and writeup to Gradescope. These are submitted as *two separate uploads* to Gradescope.

**Written Submission** You should upload to Gradescope a PDF or text file of your answers to the questions. This can be handwritten and scanned/photographed if that works best for you.

*Note that you can submit the written assignment independently of the code.* If you are unable to get the code fully working, please write up what you did and answer as many questions as possible, even partially, so we can assign you appropriate partial credit.

**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
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**


