Assignment 2

Due to: 20/12/2021 (dd/mm/yyyy)

Credits: Andrea Galassi, Federico Ruggeri, Paolo Torroni

Summary: Fact checking, Neural Languange Inference (**NLI**)

Intro

This assignment is centred on a particular and emerging NLP task, formally known as **fact checking** (or fake checking). As AI techniques become more and more powerful, reaching amazing results, such as image and text generation, it is more than ever necessary to build tools able to distinguish what is real from what is fake.

Here we focus on a small portion of the whole fact checking problem, which aims to determine whether a given statement (fact) conveys a trustworthy information or not.

More precisely, given a set of evidences and a fact to verify, we would like our model to correctly predict whether the fact is true or fake.

In particular, we will see:

- Dataset preparation (analysis and pre-processing)
- Problem formulation: multi-input binary classification
- Defining an evaluation method
- Simple sentence embedding
- Neural building blocks
- Neural architecture extension

The FEVER dataset

First of all, we need to choose a dataset. In this assignment we will rely on the FEVER dataset.

The dataset is about facts taken from Wikipedia documents that have to be verified. In particular, facts could face manual modifications in order to define fake information or to give different formulations of the same concept.

The dataset consists of 185,445 claims manually verified against the introductory sections of Wikipedia pages and classified as `Supported`, `Refuted` or `NotEnoughInfo`. For the first two classes, systems and annotators need to also return the combination of sentences forming the necessary evidence supporting or refuting the claim.

2.1 Dataset structure

Relevant data is divided into two file types. Information concerning the fact to verify, its verdict and associated supporting/opposing statements are stored in **.jsonl** format. In particular, each JSON element is a python dictionary with the following relevant fields:

- **ID**: ID associated to the fact to verify.
- **Verifiable**: whether the fact has been verified or not: `VERIFIABLE` or `NOT VERIFIABLE`.
- **Label**: the final verdict on the fact to verify: `SUPPORTS`, `REFUTES` or `NOT ENOUGH INFO`.
- **Claim**: the fact to verify.
- **Evidence**: a nested list of document IDs along with the sentence ID that is associated to the fact to verify. In particular, each list element is a tuple of four elements: the first two are internal annotator IDs that can be safely ignored; the third term is the document ID (called URL) and the last one is the sentence number (ID) in the pointed document to consider.

Some Examples

Verifiable

```
{"id": 202314, "verifiable": "VERIFIABLE", "label": "REFUTES",
"claim": "The New Jersey Turnpike has zero shoulders.", "evidence":
[[[238335, 240393, "New_Jersey_Turnpike", 15]]]}
```

Not Verifiable

```
{"id": 113501, "verifiable": "NOT VERIFIABLE", "label": "NOT ENOUGH
INFO", "claim": "Grease had bad reviews.", "evidence": [[[133128,
null, null, null]]]}
```

2.2 Some simplifications and pre-processing

We are only interested in verifiable facts. Thus, we can filter out all non-verifiable claims.

Additionally, the current dataset format does not contain all necessary information for our classification purposes. In particular, we need to download Wikipedia documents and replace reported evidence IDs with the corresponding text.

Don't worry about that! We are providing you the already pre-processed dataset so that you can concentrate on the classification pipeline (pre-processing, model definition, evaluation and training).

You can download the zip file containing all set splits (train, validation and test) of the FEVER dataset by clicking on this link. Alternatively, run the below code cell to automatically download it on this notebook.

Note: each dataset split is in .csv format. Feel free to inspect the whole dataset!

```
import os
import requests
import zipfile

def save_response_content(response, destination):
    CHUNK_SIZE = 32768

    with open(destination, "wb") as f:
        for chunk in response.iter_content(CHUNK_SIZE):
            if chunk: # filter out keep-alive new chunks
                f.write(chunk)

def download_data(data_path):
    toy_data_path = os.path.join(data_path, 'fever_data.zip')
    toy_data_url_id = "1wArZhF9_SHW17WKNGeLmX-QTYw9Zscl1"
    toy_url = "https://docs.google.com/uc?export=download"

    if not os.path.exists(data_path):
        os.makedirs(data_path)

    if not os.path.exists(toy_data_path):
        print("Downloading FEVER data splits...")
        with requests.Session() as current_session:
            response = current_session.get(toy_url,
                                   params={'id': toy_data_url_id},
                                   stream=True)
        save_response_content(response, toy_data_path)
        print("Download completed!")

        print("Extracting dataset...")
        with zipfile.ZipFile(toy_data_path) as loaded_zip:
            loaded_zip.extractall(data_path)
        print("Extraction completed!")

download_data('dataset')
```

```
Downloading FEVER data splits...
Download completed!
Extracting dataset...
Extraction completed!
```

Classification dataset

At this point, you should have a reay-to-go dataset! Note that the dataset format changed as well! In particular, we split the evidence set associated to each claim, in order to build `(claim, evidence)` pairs. The classification label is propagated as well.

We'll motivate this decision in the next section!

Just for clarity, here's an example of the pre-processed dataset:

Claim: "Wentworth Miller is yet to make his screenwriting debut."

Evidence: "2 He made his screenwriting debut with the 2013 thriller film Stoker . Stoker Stoker (film)"

Label: Refutes

Problem formulation

As mentioned at the beginning of the assignment, we are going to formulate the fact checking problem as a binary classification task.

In particular, each dataset sample is comprised of:

- `A claim to verify`
- `A set of semantically related statements (evidence set)`
- `Fact checking label: either evidences support or refute the claim.`

Handling the evidence set from the point of view of neural models may imply some additional complexity: if the evidence set is comprised of several sentences we might incur in memory problems.

To this end, we further simplify the problem by building (claim, evidence) pairs. The fact checking label is propagated as well.

Example:

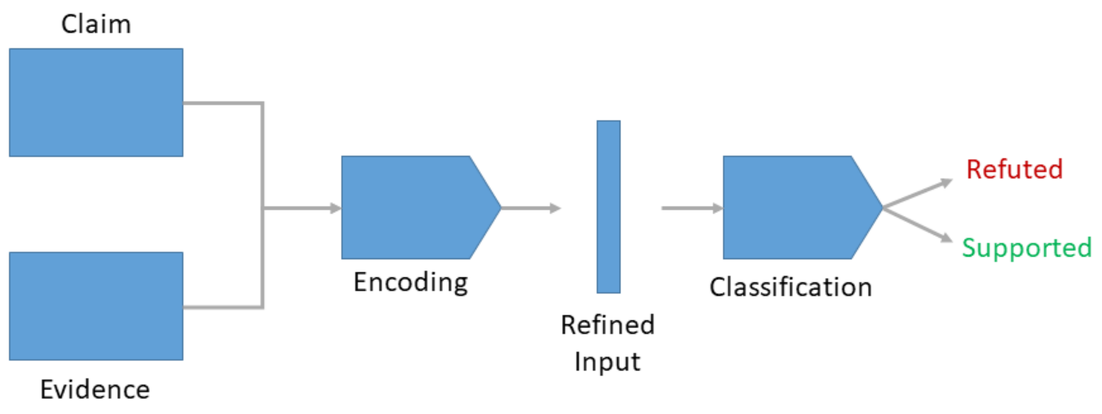
```
Claim: c1
Evidence set: [e1, e2, e3]
Label: S (support)
```

--->

```
(c1, e1, S),
(c1, e2, S),
(c1, e3, S)
```

4.1 Schema

The overall binary classification problem is summed up by the following (simplified) schema



Don't worry too much about the **Encoding** block for now. We'll give you some simple guidelines about its definition. For the moment, stick to the binary classification task definition where, in this case, we have 2 inputs: the claim to verify and one of its associated evidences.

Architecture Guidelines

There are many neural architectures that follow the above schema. To avoid phenomena like the writer's block, in this section we are going to give you some implementation guidelines.

In particular, we would like you to test some implementations so that you explore basic approaches (neural baselines) and use them as building blocks for possible extensions.

5.1 Handling multiple inputs

The first thing to notice is that we are in a multi-input scenario. In particular, each sample is comprised of a fact and its asssociated evidence statement.

Each of these input is encoded as a sequence of tokens. In particular, we will have the following input matrices:

- Claim: `[batch_size, max_tokens]`
- Evidence: `[batch_size, max_tokens]`

Moreover, after the embedding layer, we'll have:

- Claim: `[batch_size, max_tokens, embedding_dim]`
- Evidence: `[batch_size, max_tokens, embedding_dim]`

But, we would like to have a 2D input to our classifier, since we have to give an answer at pair level. Therefore, for each sample, we would expect the following input shape to our classification block:

- Classification input shape: `[batch_size, dim]`

How to do that?

We inherently need to reduce the token sequence to a single representation. This operation is formally known as **sentence embedding**. Indeed, we are trying to compress the information of a whole sequence into a single embedding vector.

Here are some simple solutions that we ask you to try out:

1. Encode token sequences via a RNN and take the last state as the sentence embedding.

2. Encode token sequences via a RNN and average all the output states.

3. Encode token sequences via a simple MLP layer. In particular, if your input is a `[batch_size, max_tokens, embedding_dim]` tensor, the matrix multiplication works on the **max_tokens** dimension, resulting in a `[batch_size, embedding_dim]` 2D matrix. Alternatively, you can reshape the 3D input tensor from `[batch_size, max_tokens, embedding_dim]` to `[batch_size, max_tokens * embedding_dim]` and then apply the MLP layer.

4. Compute the sentence embedding as the mean of its token embeddings (**bag of vectors**).

5.2 Merging multi-inputs

At this point, we have to think about **how** we should merge evidence and claim sentence embeddings.

For simplicity, we stick to simple merging strategies:

- `**Concatenation**: define the classification input as the concatenation of evidence and claim sentence embeddings`

- `**Sum**: define the classification input as the sum of evidence and claim sentence embeddings`

- `**Mean**: define the classification input as the mean of evidence and claim sentence embeddings`

For clarity, if the sentence embedding of a single input has shape `[batch_size, embedding_dim]`, then the classification input has shape:

- `**Concatenation**: `[batch_size, 2 * embedding_dim]``
- `**Sum**: `[batch_size, embedding_dim]``
- `**Mean**: `[batch_size, embedding_dim]``

A simple extension

Lastly, we ask you to modify previously defined neural architectures by adding an additional feature to the classification input.

We would like to see if some similarity information between the claim to verify and one of its associated evidence might be useful to the classification.

Compute the cosine similarity metric between the two sentence embeddings and concatenate the result to the classification input.

For clarity, since the cosine similarity of two vectors outputs a scalar value, the classification input shape is modified as follows:

- `**Concatenation**: `[batch_size, 2 * embedding_dim + 1]``
- `**Sum**: `[batch_size, embedding_dim + 1]``
- `**Mean**: `[batch_size, embedding_dim + 1]``

Performance evaluation

Due to our simplifications, obtained results are not directly compatible with a traditional fact checking method that considers the evidence set as a whole.

Thus, we need to consider two types of evaluations.

A. **Multi-input classification evaluation**

This type of evaluation is the easiest and concerns computing evaluation metrics, such as accuracy, f1-score, recall and precision, of our pre-processed dataset.

In other words, we assess the performance of chosen classifiers.

B. **Claim verification evaluation**

However, if we want to give an answer concerning the claim itself, we need to consider the whole evidence set.

Intuitively, for a given claim, we consider all its corresponding (claim, evidence) pairs and their corresponding classification outputs.

At this point, all we need to do is to compute the final predicted claim label via majority voting.

Example:

```
Claim: c1
Evidence set: e1, e2, e3
True label: S

Pair outputs:
(c1, e1) -> S (supports)
(c1, e2) -> S (supports)
(c1, e3) -> R (refutes)

Majority voting:
S -> 2 votes
R -> 1 vote

Final label:
c1 -> S
```

Lastly, we have to compute classification metrics just like before.

Shortly speaking, implement both strategies for your classification metrics.

Tips and Extras

8.1 Extensions are welcome!

Is this task too easy for you? Are you curious to try out things you have seen during lectures (e.g. attention)? Feel free to try everything you want!

Don't forget to try neural baselines first!

8.2 Comments and documentation

Remember to properly comment your code (it is not necessary to comment each single line) and don't forget to describe your work!

8.3 Organization

We suggest you to divide your work into sections. This allows you to build clean and modular code, as well as easy to read and to debug.

A possible schema:

- Dataset pre-processing
- Dataset conversion
- Model definition
- Training
- Evaluation
- Comments/Summary

Evaluation

Which are the evaluation criteria on which we'll judge you and your work?

1. Pre-processing: whether you have done some pre-processing or not.
2. Sentence embedding: you should implement all required strategies (with an example and working code for each). That is, we, as evaluators, should be able to test all strategies without writing down new code.
3. Multiple inputs merging strategies: you should implement all required strategies (with an example and working code for each).
4. Similarity extension: you should implement the cosine similarity extension (with an example and working code).
5. Voting strategy: you should implement the majority voting strategy and provide results.
6. Report: when submitting your notebook, you should also attach a small summary report that describes what you have done (provide motivations as well for abitrary steps. For instance, "We've applied L2 regularization since the model was overfitting".

Extras (possible extra points):

1. Any well defined extension is welcome!
2. Well organized and commented code is as important as any other criteria.

Contact

For any doubt, question, issue or help, you can always contact us at the following email addresses:

Teaching Assistants:

- Andrea Galassi -> a.galassi@unibo.it
- Federico Ruggeri -> federico.ruggeri6@unibo.it

Professor:

- Paolo Torroni -> p.torroni@unibo.it

Note: We highly recommend you to check the course useful material for additional information before contacting us!

FAQ

Question: Can I do something text pre-processing?

Answer: You have to! If you check text data, the majority of sentences need some cleaning.

Question: The model architecture schema is not so clear, are we doing end-to-end training?

Answer: Exactly! All models can be thought as:

1. Input
2. (word) Embedding
3. Sentence embedding
4. Multiple inputs merging
5. Classification

Question: Can I extend models by adding more layers?

Answer: Feel free to define model architectures as you wish, but remember satisfy our requirements. This assignment should not be thought as a competition to achieve the best performing model: fancy students that want to show off but miss required assignment objectives will be punished!!

Question: I'm struggling with the implementation. Can you help me?

Answer: Yes sure! Contact us and describe your issue. If you are looking for a particular type of operation, you can easily check the documentation of the deep learning framework you are using (google is your friend).

Question: Can I try other encoding strategies or neural architectures?

Answer: Absolutely! Remember to try out recommended neural baselines first and only then proceed with your extensions.

Question: Do we have to test all possible sentence embedding and input merging combinations?

Answer: Absolutely no! Feel free to pick one sentence embedding strategy and try all possible input merging strategies with it! For instance, pick the best performing sentence embedding method and proceed with next steps (extras included). Please, note that you still have to implement all mentioned strategies!

Question: I'm hitting out of memory error when training my models, do you have any suggestions?

Answer: Here are some common workarounds:

1. Try decreasing the mini-batch size
2. Try applying a different padding strategy (if you are applying padding): e.g. use quantiles instead of maximum sequence length
3. Check the efficiency of your custom code implementation (if any)
4. Try to define same length mini-batches to avoid padding (**It should not be necessary here!**)

Question: I'm hitting CUDNN_STATUS_BAD_PARAM error! What I'm doing wrong?

Answer: This error is a little bit tricky since the stack trace is not meaningful at all! This error occurs when the RNN is fed with a sequence of all 0s and pad masking is enabled (e.g. from the embedding layer). Please, check your conversion step, since there might be an error that leads to the encoding of a sentence to all 0s.
