MT Exercise 2

Topic: Data, Pre- and Postprocessing

Due date: Tuesday, April 4 2023, 14:00

Submission Instructions:

- **Submission file format: zip**
- Please follow our file naming convention: olatusername_mt_exercise_xx.zip, for instance `mmuster_mt_exercise_02.zip`
- For this exercise you will submit scripts and PDF files. Submit everything in the same zip folder.
- Please submit via the exercise submodule on OLAT. Submission is open only until Tuesday, 14:00.
- You can work alone or in groups of two. If you submit as a group, please include both usernames in the submission.

1 Cleaning Parallel Training Data (0.5 points)

For NMT systems, cleaning parallel data is an important task. This is because NMT systems are sensitive to certain kinds of noise in the training data. As an example, NMT systems react very strongly if some of the target sentences are in the source language (i.e. untranslated):

Training data	BLEU on test set
1 million target sentences in the correct language	28.9
+ 50k target sentences in the source language	11.1

If some of the target sentences are actually untranslated, we often observe that NMT systems learn a copying behaviour that leads to very bad translations. In short: sentences in the wrong language probably need to be removed from our training data.

In this exercise you will implement a language identification filter that removes sentence pairs if they are in the wrong languages. The first step is to develop a language identification model that you can use in a subsequent step for filtering.

Ngram language models for language identification

One simple method for language identification are character ngram language models. Implement a character ngram language model that you can train on any text file as a command line argument:

```
python train.py --input data/train.de --model model.de --ngram-order 5
```

The tool should also save this model so that it can be loaded later, to predict the probability of a string of characters to be in the language the model was trained on:

```
echo "Das ist ein Testsatz." | python3 predict.py --languages de en it nl \
    --models models/model.de  models/model.en models/model.it  models/model.nl

# example output
{'de': -336.7882148002182, 'en': -1438.5037969193295,
 'it': -1526.7216520126992, 'nl': -1261.3552585985071}
```

Some implementation details:

- We already provide you with skeleton code for `train.py` and all the code for `predict.py`, so that you can focus on implementing the language model itself.
- Implement a simple count-based character ngram language model. The existing code we already wrote for you assumes that all probabilities are stored in a dictionary called `self.probs`, where ngrams are keys and the values are log probabilities.
- There is no need to implement a smoothing method for unknown ngrams. Just use a very low default log probability (such as `-100.0`).

Data to train and assess language identification

We prepared some training and development data[1] for you to download:

```
mkdir -p lid_data
wget https://files.ifi.uzh.ch/cl/archiv/2022/mt22/lid_data.tar.gz
mv lid_data.tar.gz lid_data/
tar -xzvf lid_data/lid_data.tar.gz
rm lid_data/lid_data.tar.gz
ls lid_data
# expected output:
# dev.de  dev.en  dev.it  dev.nl  train.de  train.en  train.it  train.nl
```

There are four languages in total for training (DE, EN, IT and NL), and the training files have 10000 lines each. For each language, there is also a dev set with 1000 lines. You can use the dev sets to compute accuracies for your language identification models.
For this purpose, write a python script: `test_accuracy.py`. It should calculate the accuracies of your trained models over the dev sets, one value for each language as well as the total of all of them. You can focus on accuracy only, no need for recall or f-measure. Be sure to make use of the code you already wrote and import where appropriate.

```
python test_accuracy.py --lid-languages de en it nl \
--lid-models models/model.de models/model.en models/model.it models/model.nl \
--dev-sets lid_data/dev.de lid_data/dev.en lid_data/dev.it lid_data/dev.nl
```

Implementing a language filter for noisy parallel data

In the end, your script should work as follows:

```
python clean.py --src-lang [source language] --trg-lang [target language] \
--src-input [noisy source sentences] --trg-input [noisy target sentences] \
--src-output [clean source sentences] --trg-output [clean target sentences]
```

Input and output files should always be parallel, i.e. lines at the same index should correspond. Internally, your script should make use of trained language identification models (see previous section).

Testing your cleaning tool on noisy parallel data

You will test your solution on the following data:

```
wget https://files.ifi.uzh.ch/cl/archiv/2022/mt22/noisy_data.tar.gz
tar -xzvf noisy_data.tar.gz
rm noisy_data.tar.gz
ls noisy_data
# expected output:
# noisy.de  noisy.en
```

[1] This data is from IWSLT2017: `http://workshop2017.iwslt.org/`.

The data is supposed to be in DE and EN, but is noisy in the sense that the language in the source or target (or both) can be wrong. There is also one surprise language mixed in (meaning: not DE, EN, IT or NL).

Use your trained language models and filtering tool to filter this noisy parallel data set. Save the clean source and target files produced by your tool and check the results manually. Summarize your findings by answering the following questions:

1. Eyeballing the cleaned parallel data, what is your impression? Does the filtering work well?
2. How do different character ngram orders influence language identification accuracy, and by extension filtering results?
3. Can you think of a way to also filter out the surprise language, without having a trained language model in this language?

Submission: 1) Python and/or shell scripts that filter the noisy parallel data set if run, 2) a PDF with your findings.

2 Impact of Postprocessing on Translation Quality (0.5 points)

In this exercise, you are going to investigate how BLEU scores are influenced by different methods of postprocessing. The NMT toolkit we are working with is JoeyNMT[2], a tool built with Pytorch, for educational purposes. For now, you do not have to train your own model: you can download one that was already trained for you.

2.1 Installing JoeyNMT

Make sure virtualenv and Python3 are installed on your system, by trying those commands:

```
virtualenv # if this command fails: pip install virtualenv
python3 # if this command fails, install Python3 for your OS
```

Clone a convenient repository we created for you, in the desired place:

```
git clone https://github.com/moritz-steiner/joeynmt-toy-models
cd joeynmt-toy-models
```

Execute this script to create a virtualenv:

```
./scripts/make_virtualenv.sh
```

Activate this virtualenv:

```
source venvs/torch3/bin/activate
```

You will now install several pieces of software inside your virtual environment, among which `joeynmt`. Due to some recent updates in several Python packages, directly installing `joeynmt` will cause problems later on. You therefore need to install some packages manually first. Execute the following commands:

```
pip uninstall setuptools
pip install setuptools==59.5.0
pip install torchtext==0.11.2
```

Download and install software by running the following script (this includes scripts for preprocessing, Python packages such as torch and joeynmt):

```
./scripts/download_install_packages.sh
```

Those scripts take care of a lot of details for you, make sure to study them closely and understand all the steps they take.

[2] `https://github.com/joeynmt/joeynmt`

2.2 Downloading a Trained JoeyNMT Model

We will use a pretrained JoeyNMT transformer model for German→English found here:

`https://github.com/joeynmt/joeynmt#iwslt14-deen`

This script explains what the individual steps are and in which order they were executed to reproduce this trained model. Have a look at all of those commands.

`https://github.com/joeynmt/joeynmt/blob/main/scripts/get_iwslt14_bpe.sh`

Then download the model tar ball, and unpack:

```
wget https://cl.uni-heidelberg.de/statnlpgroup/joeynmt2/transformer_iwslt14_deen_bpe.tar.gz
tar -xzvf transformer_iwslt14_deen_bpe.tar.gz
```

Some logistics to rename and move files to a reasonable folder:

```
# only run this if extracting the tar ball was successful
rm transformer_iwslt14_deen_bpe.tar.gz
mkdir models
mv transformer_iwslt14_deen_bpe models/
```

Take care not to move files around as they might be referenced in the scripts or configs. Be sure to change those corresponding paths as well if you really want to move files to a different directory.

2.3 Translating with a Trained Model

For testing translation, you will be using this raw test set pair:

```
data/test.raw.de
data/test.raw.en
```

These files are not preprocessed, because JoeyNMT has built in pre-tokenizers, tokenizers, casing systems as well as a function to learn subword tokenization (BPE). You will learn about these functions in more detail in the future.
For now it is important to understand the basic series of steps necessary for translation:

```
raw -> tokenized -> truecased -> byte-pair encoding (BPE)
```

Consistency and reproducibility is paramount here. All of these steps should be undone in the same way after translation. In our case JoeyNMT handles this automatically, so testing the output hypothesis translation against the reference is possible without further postprocessing. This is not the always the case, so special attention has to be given to the pre- and postprocessing of your data.

2.4 Investigate impact of postprocessing on BLEU

First, we will translate our test file `data/test.raw.de` to English. Then, you will investigate the following question: how does postprocessing influence BLEU scores? The motivation for this is that reporting BLEU scores is not entirely standardized:

- some people report BLEU on detokenized text, others on tokenized text
- if text is still tokenized, different people use different tokenizers
- some people report BLEU on lowercased text – others dont

This creates some unintended differences between evaluations (and opportunities for light cheating) in research papers and it is not obvious how those choices influence the result.

Compute BLEU scores with SacreBLEU[3] (already installed if you followed the instructions above), using different versions of the translation (called `hyp`) and the reference (called `ref`). Take a look at `scripts/evaluate.sh` to help you get started.

Details about SacreBLEU implementation

In order to avoid tokenization differences, SacreBLEU expects the hypothesis and reference to be detokenized, and by default applies its own internal tokenization before computing BLEU. To turn off any internal tokenization, use:

```
cat [hyp] | sacrebleu --tokenize none [ref]
```

Likewise, by default SacreBLEU computes *cased* BLEU, meaning that casing matters for computing ngram precisions. You can make the tool lowercase the texts as follows:

```
cat [hyp] | sacrebleu -lc [ref]
```

Retokenize differently

Instead of SacreBLEU's internal tokenization, you could use any other tokenization method. Have a look at the `evaluate.sh` script for an example on how to use the Moses tokenizer. Try also what happens when you tokenize the hypothesis and reference differently, which is generally considered very bad practice. You might try the one found in:

```
tools/moses-scripts/scripts/tokenizer/tokenizer_PTB.perl
```

You can also write a simple one yourself or search for another one online. Be sure to mention your choice in the submission.

Investigate how those choices impact BLEU and fill in the table below by postprecossing the files in different ways. Make sure to compare like with like, for instance: if the hypothesis is tokenized, the reference you compare to must also be tokenized (except for the deliberately differently tokenized task of course). You are free to modify the scripts we provided, write new scripts or run the commands directly, but make sure to document your choice accordingly. In

Postprocessing steps (for hyp *and* ref!)	**sacreBLEU settings**	**BLEU**
Full		
Full but lowercased	`-lc`	
Tokenize with Moses tokenizer instead	`--tokenize none`	
Tokenize `hyp` and `ref` differently	`--tokenize none`	

the end, you should fill in BLEU scores in the following table:

Summarize your findings by answering the following questions:

1. Which method achieves the highest BLEU scores? Why?
2. Following up on the last question, what are some flaws in the translation setup we used here? How could it be improved?
3. Describe the effect of using two different tokenizers and how this relates to your observations in the BLEU score.
4. Describe what an ideal standardized way to report BLEU scores in a research paper should look like in your opinion.
5. How might the results have looked like if we used the opposite translation direction? Try to support your thoughts with good reasons.

Submission: A PDF document with all the BLEU scores and your findings.

In case of problems or if exercises are unclear please post in our OLAT forum.
Good luck!

[3] `https://github.com/mjpost/sacreBLEU`