User Manual — My Punctuation Prediction Project (BiLSTM & BiLST

This is the user manual I prepared for my punctuation prediction project. I implemented two variants:

1) a baseline BiLSTM tagger, and

2) a BiLSTM with transfer learning (pretrained embeddings).

Below I describe exactly how I set up the environment, prepared the data, trained and evaluated the models, and ran inference on new text. Everything here is what I actually ran on my machine, step by step, so anyone can reproduce my results.

0) What I used (Prerequisites)

- Python 3.10–3.12
- Terminal (macOS/Linux). On Windows I use Git Bash or WSL.
- (Optional) NVIDIA GPU with CUDA for faster training; CPU is fine for my model size.
- git (optional).

1) My project structure

I keep the repository like this:

```
punct_bilstm_a2z/
├─ data/
│  ├─ train.txt           # my training text with inline labels (e.g., ,COMMA  .PERIOD  ;SEMICOLON)
│  ├─ test.txt            # (optional) raw text I want to punctuate
│  └─ processed/          # created by preprocessing (train/val/test jsonl)
├─ embeddings/
│  └─ glove.6B.100d.txt   # only for the TL model (placed here by me)
├─ runs/                  # training outputs (checkpoints, vocabs, tensorboard logs)
├─ results/               # evaluation reports I save
├─ src/
│  ├─ preprocess.py
│  ├─ train.py
│  ├─ eval.py
│  ├─ infer.py
│  └─ (dataset, model, vocab, utils, etc.)
└─ tools/
   └─ plot_curves.py      # optional (I can ignore this if I don’t need plots)
```

2) How I set up Python

From the project root, I create and activate a virtual environment, then install deps:

```
python -m venv .venv
```

```
source .venv/bin/activate             # Windows: .venv\Scripts\activate
```

```
pip install --upgrade pip
```

If requirements.txt exists:

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

Otherwise I install the basics I use:

```
pip install torch numpy scikit-learn tqdm matplotlib tensorboard
```

3) Placing my data

I put my training file at data/train.txt. This file uses inline labels such as ,COMMA and .PERIOD. I optionally keep a data/test.txt containing raw, unpunctuated text that I want to restore and present later.

If I want to exclude ultra-rare labels like EXCLAMATIONMARK and QUESTIONMARK, I either remove them from train.txt or let my preprocessing step drop/remap them (see next).

4) Preprocessing (what I actually run)

This step converts the inline-labeled text into token/label JSONL files and makes a validation split. I run:

```
python -m src.preprocess --input data/train.txt --outdir data/processed --val_ratio 0.10
```

If I want to include a separate test file during preprocessing (when supported by my script), I use:

```
python -m src.preprocess --input data/train.txt --test data/test.txt --outdir data/processed --val_
```

When I need to drop/remap ultra-rare labels (only if my preprocess.py exposes these flags), I do:

```
python -m src.preprocess --input data/train.txt --outdir data/processed --val_ratio 0.10 --drop_labe
```

or

```
python -m src.preprocess --input data/train.txt --outdir data/processed --val_ratio 0.10 --remap_lab
```

After this, I check:

```
ls -l data/processed
```

I should see train.jsonl and val.jsonl (and test.jsonl if I passed --test).

5A) Training the baseline BiLSTM (my commands)

I train the baseline model (random-initialized embeddings) with class-weighted loss:

```
python -m src.train --data_dir data/processed --epochs 12 --class_weights balanced
```

Notes (what I actually use):

- Batch size: usually 32
- Learning rate: 1e-3
- Dropout: 0.3
- Sequence length: 50 with overlapping windows (stride ~16)
- Hidden size: 128 per LSTM direction (2 layers)
- I log runs to runs/bilstm and save the best checkpoint (best.pt) + vocabs there.

5B) Training the BiLSTM with transfer learning (my commands)

First I make sure I have pretrained GloVe 6B.100d vectors at embeddings/glove.6B.100d.txt.

Then I run:

```
python -m src.train --data_dir data/processed --emb_path embeddings/glove.6B.100d.txt --epochs 12 --
```

My extra tip: If my trainer supports it, I sometimes lower the embedding LR a bit (e.g., --emb_lr 5e-4 with main --lr 1e-3) to make early training more stable. I keep everything else (sequence length, batch size, etc.) identical so I can compare fairly to the baseline.

6) How I evaluate a saved checkpoint

I evaluate on validation or test like this (pointing to my best.pt):

```
python -m src.eval --data_dir data/processed --ckpt runs/bilstm/best.pt --split val
```

or:

```
python -m src.eval --data_dir data/processed --ckpt runs/bilstm/best.pt --split test
```

I look at the per-class precision/recall/F1, macro F1, weighted F1, and I save the classification report and confusion matrix under results/. For my write-up, I also compute a punctuation-only Macro F1 over {COMMA, PERIOD, SEMICOLON}. This avoids the O class dominating the score.

7) How I run inference (restoring punctuation)

For quick tests from the terminal, I do:

```
printf "c is a compiled language it gives control to the programmer\n" | python -m src.infer --ckpt
```

To punctuate an entire file and save it, I do:

```
python -m src.infer --ckpt runs/bilstm/best.pt < data/test.txt > outputs/test_punctuated.txt
```

My inference script removes existing punctuation, runs the same windowing as in training, predicts a label per token, and then reconstructs the text by adding the predicted mark after each word. I keep a tiny ruleset to avoid consecutive punctuation and to ensure clean sentence ends.

8) My training schedule & defaults (for reproducibility)

- Embedding dim: 100
- BiLSTM: 2 layers, 128 hidden per direction (output 256 per token)
- Attention: lightweight per-step attention (Linear→softmax over time)
- Class weights: balanced (I compute from label distribution); or I switch to Focal Loss if needed
- Optimizer: Adam (lr 1e-3 by default)
- Scheduler: ReduceLROnPlateau on validation Macro-F1 (factor 0.5, patience 3)
- Dropout: 0.3
- Gradient clipping: 1.0
- Epochs: 8–15 typically; I pick the best checkpoint by validation Macro-F1
- Seeds: I set Python/NumPy/Torch seeds for consistent runs

9) Troubleshooting (what I ran into and fixed)

- “val.jsonl not found” → I forgot to preprocess; I reran step 4 and made sure --outdir matches --data_dir.

- “ModuleNotFoundError: 'src.model'” → I was not at the project root; I now always run modules like: python -m src.train

- “Missing vocabs on eval” → I trained to a different runs/... folder; I pointed --ckpt to the correct one.

- “Too many commas” → class_weights=balanced helps; Focal Loss can help too; with TL I also reduce the embedding LR slightly.

- “Baseline and TL logs mixed” → I use different run names/folders when my trainer supports --run_name.

10) My quick command cheat-sheet

Preprocess:

```
python -m src.preprocess --input data/train.txt --outdir data/processed --val_ratio 0.10
```

Train (baseline):

```
python -m src.train --data_dir data/processed --epochs 12 --class_weights balanced
```

Train (TL):

```
python -m src.train --data_dir data/processed --emb_path embeddings/glove.6B.100d.txt --epochs 12 --
```

Evaluate:

```
python -m src.eval --data_dir data/processed --ckpt runs/bilstm/best.pt --split val
```

Infer (file → file):

```
python -m src.infer --ckpt runs/bilstm/best.pt < data/test.txt > outputs/test_punctuated.txt
```

TensorBoard (if I want to inspect logs):

```
tensorboard --logdir runs
```