iNLP Assignment-2; Roll: 2023201044

Code Documentation:

The generator.py file contains implementations of different language models, including:

1. Feedforward Neural Network (FFNN) - Uses a fixed-size context window.
2. Recurrent Neural Network (RNN) - Processes sequential data.
3. Long Short-Term Memory (LSTM) - An advanced RNN variant handling long-term dependencies.

Additionally, the script includes:

- Data preprocessing functions (loading corpora, tokenization, GloVe embedding loading).
- Training and evaluation utilities.
- Hyperparameter optimization using Optuna.

---

1. Data Loading and Preprocessing

load_corpus(corpus_path)

Reads a text corpus from the specified file path and returns the raw text.

load_glove(file_path)

Loads pre-trained GloVe word embeddings into a dictionary, mapping words to their vector representations. *Note* that the embedding size used is 300.

Tokenization Functions

clean_and_tokenize_1(text) & clean_and_tokenize_2(text)

Preprocess textual data by:

- Removing unnecessary sections.
- Lowercasing text.
- Handling URLs, numbers, and punctuation.
- Tokenizing text into sentences and words.

These pre-processing functions have been borrowed from the Assigment-1.

---

2. Vocabulary and Data Preparation

build_vocab(sentences, glove_dict)

Creates a vocabulary of words present in both the corpus and the GloVe embeddings.

build_ngrams(sentences, vocab, n=5)

Converts tokenized sentences into input-output pairs for n-gram models (FFNN).

build_embedding_matrix(vocab, glove_dict, embedding_dim=300)

Creates an embedding matrix from the vocabulary using pre-trained GloVe vectors. Its shape is |V|xE where E is embedding size.

---

3. Model Implementations

Feedforward Neural Network (FFNN)

class FFNN(nn.Module)

A neural network with:

- Embedding layer initialized with GloVe.
- Two hidden layers with GELU (Gaussian Error Linear Unit) activation and LayerNorm (Layer Normalization). They are used to improve model performance and stability. Layer Normalization normalizes activations across the feature dimension, making training more stable. It helps mitigate the issue of exploding or vanishing gradients, especially in deep neural networks.
- Dropout regularization.
- Fully connected output layer.

Recurrent Neural Network (RNN)

class RNNModel(nn.Module)

A sequence-based model that consists of:

- Embedding layer.
- 2-layer RNN with Tanh activation.
- Fully connected output layer.
- Handles variable-length sequences using pack_padded_sequence.

Long Short-Term Memory (LSTM)

class LSTMModel(nn.Module)

Similar to RNN but uses LSTM cells to retain long-term dependencies.

---

4. Training and Evaluation

train(model, optimizer, loss_fn, train_loader, val_loader, num_epochs, early_stop_patience)

Trains the model, monitors validation loss, and implements early stopping.

evaluate(model, loss_fn, data_loader)

Computes the average loss on a dataset.

calculate_perplexity(model, data_loader)

Computes the perplexity metric for a trained model.

calculate_perplexities_and_save(model, data_loader, sentences, vocab, file_path)

Computes the perplexity of each sentence for a trained model, the average perplexity and save them in text file. Note the average perplexity is computed per sentence (and not token).

---

5. Hyperparameter Optimization

optimize_hyperparams(train_dataset, val_dataset, test_dataset, vocab, glove_dict, n, num_trials, num_epochs)

Uses Optuna to tune hyperparameters such as:

- Batch size: [32, 64]
- Hidden layer dimensions: [128, 256]
- Dropout rate: [0.2, 0.5]
- Optimizer type: ["adam" with lr=0.01, "sgd" with lr=0.01 and momentum=0.9]

A similar function optimize_hyperparams_seq() is used for RNN and LSTM models. The number of epochs used was 15-20.

---

6. Prediction

predict_top_k_words(model, input_sentence, vocab, index_to_word, k=5, n=5)

Given an input sequence, predicts the top k most probable next words.

Note: The functions with subscript *_seq* are meant to be used for sequence models. The implementation remains similar except that a lengths tensor is used to mask <PAD> tokens. It ensures that the model does not compute loss on padding positions. Use of pack_padded_sequence and pad_packed_sequence efficiently handles variable-length sequences.

---

7. Best hyper-parameters and the pretrained model name:

get_best_params_and_model(corpus_path, lm_type, N=None)

Given the corpus and LM (FFNN, RNN or LSTM), the best set of hyper-parameters obtained during hyper-parameter optimization and the corresponding pre-trained model name is returned. The pre-trained models are trained for maximum epochs of 20. For sequence models like RNN and LSTM, the epochs for training may be insufficient, especially with larger corpus like Ulysses corpus (with almost 4x the number of sentences in Pride and Prejudice corpus).

---

8. Running the Script

main(lm_type, N, corpus_path, k)

- Accept an input sentence whose next word prediction is to be returned.
- Loads and processes the corpus.
- Builds vocabulary and prepares training/testing data. The train-val-test split followed for Pride and Prejudice (corpus 1) is R-800-1000 sentences (where R is the count of remaining sentences) and for Ulysses (corpus 2) is R-2000-1000 sentences.
- Loads or trains a model based on the specified lm_type (FFNN, RNN, or LSTM).
- Computes perplexity for different datasets and writes perplexities to text files.
- Generates predictions for the given input sentence.

Command-line execution

The script supports execution via command-line arguments:

python generator.py <lm_type> <N> <corpus_path> <k>

where:

- lm_type: 'f' for FFNN, 'r' for RNN, 'l' for LSTM.
- N: Context window size (only for FFNN). If sequence model, any non-negative value works.
- corpus_path: Path to the dataset.
- k: Number of top predictions to generate.

The last section (commented) summarizes the average perplexity for each combination of (lm_type, N, corpus) and presents in a tabular format.

Results:

Model	Train PPL	Validation PPL	Test PPL
FFNN(n=3) - Pride and Prejudice	66.00	145.91	106.24
FFNN(n=3) - Ulysses	163.47	208.36	230.88
FFNN(n=5) - Pride and Prejudice	74.20	139.52	98.29
FFNN(n=5) - Ulysses	152.45	211.74	232.99
RNN - Pride and Prejudice	145.67	203.12	196.14
RNN - Ulysses	312.80	255.25	255.06
LSTM - Pride and Prejudice	79.86	132.98	126.12
LSTM - Ulysses	206.26	254.97	246.59

The pretrained models can found in the G-drive link.

Analysis:

From the perplexity (PPL) values in the table, here are some key observations and inferences:

1. Higher Perplexity for Ulysses
    - Across all models, Ulysses has significantly higher perplexity values compared to Pride and Prejudice.
    - This aligns with the fact that Ulysses has a more complex and less typical sentence structure in English. Models struggle more with predicting the next word due to its unique style.
2. FFNN vs. RNN vs. LSTM Performance
    - Feedforward Neural Networks: The FFNN (n=3) performs slightly worse than the n=5 version, indicating longer context help improve learning.
    - Recurrent Neural Networks: RNN performs much worse, particularly for Ulysses, showing that vanilla RNNs might struggle with long-term dependencies and complex structures.
    - Long Short-Term Memory: LSTM has the lowest perplexity among sequence models, indicating its ability to capture long-term dependencies better than a vanilla RNN. This is especially evident for the Pride and Prejudice corpus that contains longer sentences than Ulysses, where RNN has much higher perplexity than LSTM.

Performance Rankings:

1. Corpus 1: Pride and Prejudice

Method	**Average Perplexity**					
	N=1		**N=3**		**N=5**	
	Train	Test	Train	Test	Train	Test
Laplace Smoothing	406.68	393.48	493.52	3563.89	355.16	5553.17
Good Turing Smoothing	14602.0	13592.2	11131.6	184.88	109209.4	1471.5
Linear Interpolation	409.85	432.8	49.2	444.20	42.75	468.43

2. Corpus 2: Ulysses

Method	**Average Perplexity**					
	N=1		**N=3**		**N=5**	
	Train	Test	Train	Test	Train	Test
Laplace Smoothing	796.10	680.87	2330.6	16746.49	1862.6	24417.3
Good Turing Smoothing	17656.5	13950.2	51386.9	906.2	184264.9	1866.0
Linear Interpolation	898.6	810.7	60.39	1430.7	54.3	1640.5

1. FFNN > LSTM > RNN >> Laplace Smoothing > Linear Interpolation > GT (small n)
2. FFNN > LSTM > RNN >> Linear Interpolation > GT > Laplace Smoothing (larger n)

Comparison of current LMs with those in A1:

1. In A1, we saw that with larger N, test perplexities increased for Laplace Smoothing and Linear Interpolation N-gram models.

2. Linear Interpolation performed better than other smoothing techniques.
3. Overfitting is attributed to large difference in train and test perplexities.
4. Current LMs perform better than any previous model (A1) on unseen sentences (lower perplexity). Besides, the models tackle overfitting issue to a great extent.
5. For any model, the perplexity is worse for Ulysses corpus than for Pride and Prejudice. This is again attributed to the complex linguistic structures.

Reasons for performance difference:

1. Overfit: N-gram models overfit due to reliance on exact sequence counts. Neural LMs, particularly LSTMs, learn distributed word representations (embeddings) that generalize better, leading to lower perplexities on unseen data.
2. Context Leverage: Unlike N-gram models, which have a fixed memory window, neural models can capture long-range dependencies through hidden states (RNN, LSTM) or deep representations. This is especially beneficial for complex texts like Ulysses, where sentence structures are less predictable.