

Assignment 1: Author Attribution using Word2Vec

Natural Language Processing Course

1 Goal

The goal of this assignment is to give you practice with implementing Word2Vec from scratch and use it for some realistic tasks. In this case, we will have you work on developing stylometric analysis techniques for author attribution. Author attribution is a fundamental problem in natural language processing with applications in forensics, plagiarism detection, and literary analysis. By implementing Word2Vec yourself, you will gain a good understanding of word embeddings and their role in capturing semantic and stylistic patterns in text.

2 Problem Statement

You will be provided with text data from 10 authors for training. Your first goal is to train a Word2Vec model to create an embedding per word (or whatever is the granularity of token you choose for the tasks ahead).

Your submission will be evaluated on a **different set of authors**, some of which may not appear in the training data. This requires you to develop methods that can quickly generalize to new writing styles beyond the specific authors in your training set. You will be evaluated on two tasks.

2.1 Task 1: Author Verification and Ranking

Given a text chunk from a specific author (not necessarily from any author in the training set), you must identify which among several candidate text chunks is written by the same author.

Input:

- A query text chunk X from author Y_i
- A set of candidate text chunks $\{Z_1, Z_2, \dots, Z_T\}$
- Exactly one $Z_j \in \{Z_1, Z_2, \dots, Z_T\}$ is written by the same author Y_i

Output: A ranked list of candidate chunks ordered by decreasing similarity to the query author's style. The candidate predicted as most likely from the same author should be ranked first.

2.2 Task 2: Author Clustering

Given a collection of unlabeled text chunks written by k unknown authors (not necessarily from the authors in the training set), you must group the chunks by author identity.

Input:

- A set of n text chunks: $\{X_1, X_2, \dots, X_n\}$
- These chunks are written by k distinct authors
- Each author has contributed at least m chunks: $mk < n$

- The number of authors k and minimum chunks per author m will be provided

Output: A list of length n where each element is the predicted author cluster ID (integer from 0 to $k - 1$). For example, an output `[0, 2, 0, 1, 2, 1, 0]` indicates that chunks 1, 3, and 7 belong to the same author (cluster 0), chunks 4 and 6 belong to another author (cluster 1), and chunks 2 and 5 belong to a third author (cluster 2).

3 Data

You will receive text samples from multiple authors during development. Use this data to develop and validate your approach.

Important: Final evaluation will use:

- **Variable chunk sizes:** Text chunks will range from 50 to 500 words
- **Unseen authors:** Some test authors may not appear in your training data
- **Different time periods:** Authors may span different literary eras

Your methods must generalize beyond the training authors and handle variable-length inputs.

4 Implementation Details

4.1 Word2Vec Implementation Requirement

You are **required to implement Word2Vec from scratch using PyTorch**. You may implement either Skip-gram or CBOW (Continuous Bag of Words) architecture, or any extension thereof. Pre-trained embeddings or the gensim library are **not allowed**. Your implementation should include:

- Training word embeddings on the provided author text data
- Proper handling of vocabulary construction
- Negative sampling or hierarchical softmax for efficient training

4.2 Allowed Libraries

- PyTorch for model building and Word2Vec implementation
- NumPy for numerical operations
- Any other standard Python library included in the default installation

Note: gensim, pre-trained word embeddings (GloVe, FastText, etc.), and any pre-trained language models are **strictly prohibited**.

Any additional packages can be requested over Piazza and are only allowed after verification.

4.3 Scope and Approach Clarification

The primary focus of this assignment is **Word2Vec-based text representation**. You should use Word2Vec embeddings as your core text representation method.

Allowed:

- Custom vocabulary construction (character n-grams, word n-grams, subwords, etc.)

- Lexical and stylometric features (sentence length, punctuation patterns, word frequency, etc.) as *supplementary* features
- Any aggregation method to combine word embeddings (average, weighted average, TF-IDF weighting, etc.)
- Test-time adaptation for Task 2 (e.g., fine-tuning embeddings on test chunks)
- Simple distance-based classifiers (cosine similarity, K-Means, K-NN)

Not Allowed:

- Neural network classifiers (MLP, CNN, LSTM, Transformers, etc.)
- Any deep learning model beyond Word2Vec itself
- Pre-trained embeddings or models of any kind
- External datasets beyond what is provided

The goal is to understand and implement Word2Vec, not to build complex neural architectures for classification.

4.4 Input Format

4.4.1 Task 1 Input

You will receive a JSON file with the following structure:

```
{
  "query_id": "<unique_id>",
  "query_text": "<text_chunk>",
  "candidates": {
    "cand_1": "<text_chunk>",
    "cand_2": "<text_chunk>",
    ...
  }
}
```

4.4.2 Task 2 Input

You will receive a JSON file with the following structure:

```
{
  "num_authors": k,
  "min_chunks_per_author": m,
  "chunks": [<text_1>, <text_2>, <text_3>, ...]
}
```

where **chunks** is a list of n text strings.

4.5 Output Format

4.5.1 Task 1 Output

Your model's predictions should be saved in a JSONL file, with each line formatted as follows:

```
{"query_id": "<query_id>", "ranked_candidates": ["cand_3", "cand_1", "cand_2", ...]}
```

where **ranked_candidates** is an ordered list of candidate IDs from most likely to least likely to be from the same author as the query.

4.5.2 Task 2 Output

Your model’s predictions should be saved in a JSON file formatted as follows:

```
[0, 2, 0, 1, 2, 1, 0, 1, 2, ...]
```

This is a list of length n where the i -th element represents the predicted cluster ID (0 to $k - 1$) for text chunk X_i .

Sample input and prediction files will be provided for reference.

5 Evaluation Criteria

5.1 Task 1: Mean Reciprocal Rank (MRR)

Your model will be evaluated using the Mean Reciprocal Rank metric:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$$

where rank_i is the position of the correct same-author chunk in your ranked list for query i , and $|Q|$ is the total number of queries.

5.2 Task 2: Hungarian Algorithm Accuracy

Since cluster IDs are arbitrary labels, we use the Hungarian algorithm to find the optimal mapping between your predicted clusters and the true author labels. The evaluation process is:

1. Construct a confusion matrix C where C_{ij} represents the number of chunks assigned to predicted cluster i that truly belong to author j
2. Apply the Hungarian algorithm to find the optimal one-to-one assignment between predicted clusters and true authors that maximizes the total number of correct assignments
3. Compute accuracy as the fraction of correctly assigned chunks under this optimal mapping

The Hungarian algorithm implementation will be provided to you in `hungarian_eval.py`.

6 Code

6.1 Programming Language

You may program the software in Python only. The version that will be used to test your code is Python 3.10.

6.2 Execution Commands

6.2.1 Training

Executing the command `./run_model.sh <train_dir>` will train your model.

- `<train_dir>` is a directory containing raw text files (e.g., `author1.txt`, `author2.txt`). You must read all files in this directory to construct your training corpus.
- You are responsible for creating your own train/validation split from this data.

This should save your model and any necessary files for inference in the current working directory.

6.2.2 Inference

Task 1 and Task 2 are evaluated separately with different test files:

Task 1: Executing the command `./run_model.sh test1 <task1_test_file> <output_dir>` will:

- Load your trained model
- Process the Task 1 test file (author verification queries)
- Save `task1_predictions.jsonl` in the output directory

Task 2: Executing the command `./run_model.sh test2 <task2_test_file> <output_dir>` will:

- Load your trained model
- Process the Task 2 test file (clustering data)
- Save `task2_predictions.json` in the output directory

Note on Task 2: Since Task 2 is an unsupervised clustering task, you are permitted to perform test-time adaptation—that is, you may finetune or adapt your embeddings using the unlabeled test chunks provided in `task2_test_file`. This is a form of transductive learning and does not violate any rules since you do not have access to the ground truth labels. This allows your model to better capture the stylistic patterns specific to the test authors.

You can assume that the test files and any model files saved during training exist in the present working directory.

6.3 Execution Flow

The TA will execute your scripts as follows:

```
bash install_requirements.sh
bash run_model.sh data/train_data/
bash run_model.sh test1 task1_test.json predictions/
bash run_model.sh test2 task2_test.json predictions/
```

Your code will be tested on a Linux system with 16GB of memory and 4 cores. You are responsible for making sure that it runs smoothly on Linux without any errors. GPU support is not guaranteed; your code should run on this CPU machine.

6.4 Time Limits

Your code must complete within the following time limits:

- **Training:** 30 minutes maximum
- **Task 1 Inference:** 5 minutes maximum
- **Task 2 Inference:** 30 minutes maximum (allows for test-time adaptation)

Scripts that exceed these limits will be terminated and will receive zero marks for the corresponding component.

7 What to Submit?

7.1 Submission Format

Submit a zip file named `<entry_number>.zip` through Moodle. Upon unzipping, it should create a directory named `<entry_number>` containing the following files:

- `compile.sh` (can be empty if no compilation needed)
- `install_requirements.sh` for setting up the environment
- `run_model.sh` for training and testing the model
- `writeup.txt` with specific format described below
- All your source code files

You will be penalized for any submissions that do not conform to this requirement.

7.2 Writeup Format

The `writeup.txt` should have the following structure:

First line: The number 2 (this is a legacy format indicator for Python, which is the only allowed language for this assignment).

Second line: Must start with this honor code exactly as written:

“Even though I have taken help from the following students and LLMs in terms of discussing ideas and coding practices, all my code is written by me.”

Third line: List names of all students and LLMs you discussed/collaborated with. If you never discussed the assignment with anyone else, write “None”.

After these three lines: You are welcome to describe your approach, methodology, features used, hyperparameters, and any other relevant details about your implementation.

8 Submission Guidelines

Your submission must adhere to the following instructions:

- The assignment is to be done **individually**.
- You should use Python 3.10 for this assignment. Only aforementioned packages are allowed. Any additional packages can be requested over Piazza and are only allowed after verification.
- Using any external source of data is prohibited.
- Use of pre-trained embeddings (GloVe, FastText, Word2Vec) or the gensim library is strictly prohibited. You must implement Word2Vec from scratch.
- Use of pre-trained language models (BERT, GPT, etc.) or generative models for any purpose (including synthetic data generation) is strictly prohibited.
- Your code will be tested on a Linux system with 16GB of memory and 4 cores. GPU support is not guaranteed. You are responsible for making sure that it runs smoothly on CPU without any errors.

- You must not discuss this assignment with anyone outside the class. Make sure you mention the names in your write-up in case you discuss with anyone from within the class. Please read academic integrity guidelines on the course home page and follow them carefully.
- We will run plagiarism detection software. Any person found guilty will be awarded a suitable penalty as per IIT rules.
- Your code will be automatically evaluated. You will get a minimum of 20% penalty if it does not conform to output guidelines.
- You cannot ask LLMs to write code for you. However, you are allowed to give the LLM your code in case you are stuck in debugging, so that you can learn about your mistakes fast. Make sure you mention which LLMs you used in your writeup.
- Please do not search the Web for direct solutions to this problem.
- Please do not use ChatGPT or other large language models for creating direct solutions (code) to the problem. Our TAs will ask language models for solutions to this problem and add generated code to plagiarism software. If plagiarism detection software can match with TA code, you will be caught.

9 Code Verification Before Submission

Your submission will be auto-graded. This means that it is absolutely essential to make sure that your code follows the input/output specifications of the assignment. Failure to follow any instruction will incur a significant penalty.

Before submitting:

- Test your code on the provided sample data
- Verify that output files are generated in the correct format
- Ensure your code runs without errors on Linux
- Check that all required files are present in your submission

Details of code verification procedures will be shared on Piazza.

10 Deadline

The assignment must be submitted by **9th February, 2025, 11:59 PM**. As per class rules, it can be submitted up to a week late with a 10% penalty per day.

11 Clarifications

For any doubts or clarifications, please use the Piazza forum. Ensure your code is well-commented, follows the submission guidelines, and is tested thoroughly to avoid any penalties related to non-conformance to output formats or runtime errors. It is your responsibility to seek clarification on any aspect of the assignment you find unclear before the deadline.