ITCS 6101/8101 Homework 2: Implementation (60 points)

February 3, 2026

In this assignment, you are given a code base that implements Mikolov's Skip-gram model for learning word embeddings, in two versions: C++ with STL (Section 1) and Python with NumPy (Section 2). You are asked to choose one of them and complete the following tasks:

1. Implement the **gradient update logic** for training embeddings, using the gradient formulas and parameter update equations shown in **Section 5.5.2 Learning skip-gram embeddings** from the textbook.
2. Implement functions that use the trained embeddings to answer **word similarity** and **word analogy** questions.
3. Run the complete code to **train word embeddings** on text that contains a lower-cased, tokenized version of the first 1 Billion characters in Wikipedia.
4. Use the trained word embeddings to answer **similarity** and **analogy questions**.
5. Write a **report** summarizing experimental results and insights.

The skeleton code and data are linked from the course webpage. Organize your code in folders as shown in Table 1, where:

- The `data/` folder contains two text files: `wiki-1B.txt` to use in the experimental evaluation, and the much smaller `wiki-1B.txt` for debugging.
- The `vocab.txt` file contains the vocabulary learned from the text corpus, mapping token types to their frequencies, restricted to those that appear at least 5 times in the corpus.
- The `vec.txt.1` and `vec.txt.2` will contain the learned *target* and *context* embeddings, respectively.
- The `traces.txt` will contain the exact output that your code produces when running the training and testing commands. Ensure that the trace file shows *the time it takes your code to train* embeddings by enclosing the training command with calls to `date`, as shown in Section 1. Document the training time in your report.

At a minimum, you should report results from training word embeddings with the configurations below, although it is highly recommended to try and compare multiple configurations:

- First, train and save embeddings of size 200, with 5 negative samples for each positive context word, with a maximum skip length between words of 5 (context size), for at least 3 iterations.
  - Note that the NumPy implementation, even when vectorized, is much slower than the C++ implementation. For examples, using the parameters above on an Apple M4 Pro, training the C++ code took less than 5 minutes, whereas the NumPy code needed over 5 hours to complete training.
- Second, train and save an additional set of embeddings where you increase the context size from 5 to 15.

Once the embeddings are trained, use the target embeddings to evaluate the quality of top-K outputs for the following:

1. **Similarity**: Print and evaluate the most similar words for each of the following words: *book*, *trip*, *paris*, *stop*, *write*, *language*, *beautiful*, *bad*, *quickly*, *amazing*.
   - *Context size* Compare the embedding performance between the small (5) and large (15) context sizes, in terms of the top-K words identified obtained for these similarity questions.
2. **Analogy**: Print and evaluate the highest scoring words for the following triplets: *paris is to france what tokyo is to X*, *balloon is to air what bucket is to X*, *novel is to writer what music is to X*, *white is to snow what red is to X*, *liquid is to water what solid is to X*, *water is to liquid what ice is to X*.

hw02/
- **report.pdf**
- c++/
  - src/
    - LMEmbedMain.cc
    - LMEmbedModel.[h|cc]
    - TestEmbeddings.cc
    - Vocabulary.[h|cc]
    - VocabularyWord.[h|cc]
    - Utils.[h|cc]
  - Makefile
  - vocab.txt, vec.txt.1, vec.txt.2
  - traces.txt
- numpy/
  - lm_embed_main.py
  - lm_embed_model.py
  - test_embeddings.py
  - vocabulary.py
  - vocab.txt, vec.txt.1, vec.txt.2
  - traces.txt
- data/
  - wiki-1B.txt
  - wiki-1M.txt

Table 1: Folder structure.

When you evaluate the top outputs, try to explain any unexpected results or mistakes, by referring to properties of the algorithm or the data. Whenever you formulate a hypothesis, it is important that it is supported empirically. As such, feel free to evaluate the trained embeddings on words and analogy examples that go beyond the ones above. All the required results and analyses should be included in an appropriately edited homework report, using proper indentation, section titles, and formatting. If you include formulas, make sure that you use appropriate formatting of equations. The PDF of the report `report.pdf` should be submitted on Canvas, together with the code in the folder above. The assignment will be graded based on **both** the code and the quality of the report.

1 C++ implementation details

The target and context embeddings are stored in the arrays `VocabularyWord::embed1_` and `VocabularyWord::embed2_`, respectively. The `LMEmbedMain.cc` contains the main function for launching the training. After completing the code, you will compile the entire package by running the `make` command with the provided `Makefile`, which will create two executables, `wvembed` for training and `testembed` for testing.

For example, you can train with the parameter minimal settings above by running the command below:

```
c++> date; ./wvembed -train ../data/wiki-1B.txt -savevocab vocab.txt -output
        vec.txt-size 200 -context 5 -subsample 1e-4 -negative 5 -iter 3
           >!  trace.txt; date
```

Similarly, you can test the embeddings by running the command below:

```
c++> ./testembed vocab.txt vec.txt.1
```

You will need to write code in the sections marked with `YOUR CODE HERE` as detailed below, but feel free to implement auxiliary functions if you think that makes the code more readable or more efficient:

- `Utils.cc`: complete the `getSimilarity()` and `dotProduct()` function.
- `VocabularyWord.cc`: complete the `similarity()` function.
- `Vocabulary.cc`: complete the two `getTopWords()` functions.

- `LMEmbedModel.cc`: complete the `train()` function.

2 NumPy implementation details

The target and context embeddings are stored in the NumPy arrays `Vocabulary.etarget` and `Vocabulary.econtext`, respectively. The file / class naming and the command line is similar to the C++ version.

You will need to write code in the sections marked with `YOUR CODE HERE` as detailed below, but feel free to implement auxiliary functions if you think that makes the code more readable or more efficient:

- `vocabulary.py`: complete the code in functions `similarity()`, `getSimilarWords()`, and `getAnalogyWords()`.
- `lm_embed_model.py`: complete the `train()` function.

3 Bonus exercises

Any non-trivial and insightful extra work can be worth bonus points. For example:

1. Determine if adding the target and context embeddings improves the answers to similarity or analogy questions. Determine if answering analogy questions works well without normalizing the embeddings first.
2. Train only one embedding matrix for both target and context words, compare embedding performance with current approach.
3. Implement both the C++ and NumPy versions and compare their efficiency in terms of running time.
4. Improve the running time, e.g., through parallel / multi-threaded execution, by reading tokens form files asynchronously with the gradient updates, etc.
5. Explore how the performance of word embeddings changes as a function of using more iterations, more data, more negative examples, multiple context sizes, etc.
6. Implement sparse word embeddings using *tf.idf* or *ppmi* encodings, and compare them against dense skip-gram embeddings.
7. Identify instances of gender bias using the methods introduced on slides 90 and 91 in the lecture on word embeddings.

4 Submission

Electronically submit on Canvas a hw02.zip file that contains the hw02 folder in which your code is in the required files, as well as the `report.pdf` and `traces.txt` files.

On a Linux system, creating the archive can be done using the command:
`> zip -r hw02.zip hw02`.

Please observe the following when handing in homework:

1. Structure, indent, and format your code well.
2. Use adequate comments, both block and in-line to document your code.
3. Verify your code runs correctly when used in the directory structure shown above. We will not debug your code.
4. For your report, it is recommended to use an editor such as Overleaf for Latex, Word, or Jupyter-Notebook that allows editing and proper formatting of equations, plots, and tables with results.
5. Verify that your Canvas submission contains the correct files by downloading and unzipping it after posting on Canvas.