COMP4650/COMP6490 Document Analysis
2025 Semester 2

Assignment 2 (Full Release)

Due 23:55 on Wednesday 8 October 2025 AEDT (UTC/GMT +11)

Overview

This assignment consists of four questions. The first and the second questions involve training and using static word embeddings. The third question uses a recurrent neural network (RNN) for name generation. The last question is about sentiment classification using both a trained Transformer classifier and an LLM.

Throughout this assignment, you will develop a better understanding of

1. static word embeddings, including how they are trained and how they are used,
2. the implementation of an RNN and how it can be trained and used for sequence generation, and
3. how to train a Transformer model for sequence-level classification.

Submission

- The answers to this assignment (including your Python code files) have to be submitted online through Wattle.
- You will produce an **answers file** with your responses to each question. Your answers file must be a PDF file named `u1234567.pdf` where `u1234567` should be replaced with your Uni ID. You will also need to include the **Python files** that you have modified as part of the submission. Specifically, follow the instructions below to prepare your final submission file:
  1. Create a folder named after your Uni ID (e.g., u1234567).
  2. Inside the folder, place the following files: your answers PDF (e.g., u1234567.pdf), `train_word2vec.py`, `features.py`, `text_classification.py`, `rnn_name_generator.py`, `sentiment_classifier.py`.
  3. Compress the folder into a ZIP file named after your Uni ID (e.g., u1234567.zip)
- Submit this single ZIP file to Wattle. **Please make sure NOT to include any data file.**
- **No late submission will be permitted without a pre-arranged extension**. A mark of 0 will be awarded if your answers (including your code files) are not submitted by the due date without a valid extension request.

Marking

This assignment will be marked out of 40, and it will count towards 20% of your final course mark.

Your answers to coding questions will be marked based on the quality of your code (is it efficient, is it readable, is it extendable, is it correct) and the solution in general (is it appropriate, is it reliable, does it demonstrate a suitable level of understanding).

Your answers to discussion questions will be marked based on how convincing your explanations are (are they clearly written, are they sufficiently detailed, are they well-reasoned, are they backed by appropriate evidence, do they use appropriate aids such as tables and plots where necessary).

This is an individual assignment. Group work is not permitted. Assignments will be checked for similarities. You are allowed to use generative AI tools to help you with non-essential parts of the assignment, such as to check how a Python package or function works, to understand a concept you are not sure about, or for proofreading. You are not allowed to use any generative AI tool to help you directly answer the questions (including writing code for you).

Question 1: Training Static Word Embeddings (8 marks)

In class, you have learned about static word embeddings and the method called skip-gram with negative sampling (SGNS) that can be used to train static word embeddings. In this question, we will use the implementation of the SGNS method from Gensim's Word2Vec class to train word2vec models. The training corpus we will use is from the 20newsgroups dataset that was used in Assignment 1, but we include much more text from the original 20newsgroups dataset as training data for our word2vec models.

Read the documentation of the Word2Vec class at the URL below to understand how to use this class to train word2vec models: `https://radimrehurek.com/gensim/models/word2vec.html`

Specifically, note that Word2Vec implements both the SGNS method and another method called CBOW (continuous bag of words). We will use the SGNS method in this assignment.

In the provided starting code, implement the function `process_training_data` inside the file `train_word2vec.py` to pre-process the training data file. Then in the function `train_model`, modify the code to train a word2vec model using the sentences given and the specified hyperparameters `window` and `seed`. Use the following settings for the other hyperparameters:

- Set the dimensionality of the word embeddings to be 200.
- Learn embeddings of only those words that have occurred at least 10 times in the given corpus.
- Set the negative sample size to be 10.
- Run 5 epochs of training.
- Set `workers` to 1.

Now you are ready to train word2vec models in the main program. Train models with different configurations as described below and answer the questions in your answers PDF.

(A) Train a word2vec model with a window size of 2. When preparing the training data, do not remove stop words. Use a random seed of 1. Let's denote this model *M1*.

Using this trained word2vec model, write Python code to obtain the vectors for 'baseball', 'basketball', and 'computer'. Calculate the cosine similarity between 'baseball' and 'basketball' and the cosine similarity between 'baseball' and 'computer'. In your answers PDF, report the two cosine similarities, state whether the cosine similarities are what you have expected, and briefly explain why there is a clear difference between the two cosine similarities.

(B) Train another word2vec model *M2* with the same configuration as in (A) except that random seed is now set to be 2. Obtain the vectors for 'baseball' and 'basketball' from *M2* and calculate their cosine similarity. Compare it with the cosine similarity between 'baseball' and 'basketball' from (A). Are the two cosine similarities the same? If not, are they close? Report what you observe and briefly explain why in your answers PDF.

(C) Calculate the cosine similarity between *M1*'s vector for 'baseball' and *M2*'s vector for 'basketball'. Report the cosine similarity in your answers PDF and briefly explain why the similarity is not high.

(D) Using either *M1* or *M2*, find the top-20 most similar words to 'would' and the top-20 most similar words to 'greece'. Report these words in your answers PDF and briefly explain why these words are close to 'would' and 'greece', respectively, in the learned embedding space. (Note that because our training corpus is relatively small, some of the top-20 most similar words may be noisy words. You can focus on explaining those top-words that make sense to you.)

(E) Train a third word2vec model *M3* using a window size of 5. Do not remove stop words. Use any random seed.

Get the top-20 similar words to 'would' and 'greece' by *M3*. Report these words in your answers PDF and explain in what way they are different from those you have found in (D) above. Explain why you think there is the difference.

(F) Train a fourth word2vec model *M4* using a window size of 5. Remove stop words this time when pre-processing the data. Use any random seed.

Get the top-20 similar words to 'would' and 'greece' by *M4*. Report these words in your answers PDF and explain in what way they are different from those you have found in (E) above. Explain why you think there is the difference.

Question 2: Static Word Embeddings for Text Classification (10 marks)

Recall that in Assignment 1 you have used logistic regression with TF-IDF vectors for text classification. With static word embeddings, we can represent documents using a dense vector derived from the static word embeddings of the words inside the document and use these dense vector representations of documents for text classification. In particular, in this question, we will use the *average* (i.e. *mean*) of the static word embeddings from a document to represent that document. Specifically, let $\mathcal{D}$ represent the set of words inside a document d after tokenisation, and let $\mathbf{e}_w$ represent the static word embedding of word w. Then we will use the following vector representation $\mathbf{e}_d$ to represent d:

$$\mathbf{e}_d = \frac{1}{|\mathcal{D}|} \sum_{w \in \mathcal{D}} \mathbf{e}_w.$$

(Hint: In your implementation, you might find it useful to use the `numpy.mean()` function to compute the mean of a set of vectors.)

You are given the same data file as used in Q2 of Assignment 1 for this question. Follow the following steps to implement the code.

1. First, inside `features.py`, implement the function `get_document_vector`. This function converts a tokenised document into a dense vector by averaging the embedding vectors of the words in the document. Note that the first parameter of this function is a document represented as a list of sentences, where each sentence is a list of words (i.e., tokens).
2. Next, inside `text_classification.py`, implement the function `tokenise_document`. This function turns a string representing a document into a list of sentences where each sentence is a list of tokens. Note that you need to first split the input string into sentences. You can use `nltk.tokenize.sent_tokenize()` to perform sentence splitting. Note also that you can choose to remove or keep stop words here. See parts (B) and (C) below for more details.

You will use the word2vec word embedding models that you have trained in Question 1 to construct your document vectors and answer the following questions. Note that you can specify which word2vec model to use inside the main program of `text_classification.py`.

Answer the following questions in your answers PDF:

(A) Recall from Assignment 1 that the text classification task at hand is topic-based document classification, where the two classes of documents cover different topics. Without running any experiments, which word2vec model among *M1*, *M3* and *M4* from Question 1 do you think is likely to work better for this task? Provide your justification to support your hypothesis. Note that this Part (A) is a theory question and you do not need to train any text classifier at this point.

(B) Note that when training *M1* and *M3* in Question 1, stop words were not removed in the training corpus. When you use either *M1* or *M3* to construct your document embeddings for text classification in Question 2, how do you think stop words in the documents to be classified should be handled when tokenising the documents? Choose one of the options below and justify your answer without running any experiments. (1) The stop words should not be removed from the documents. (2) It does not make any difference whether stop words are removed or not. (3) It is better to remove the stop words from the documents.

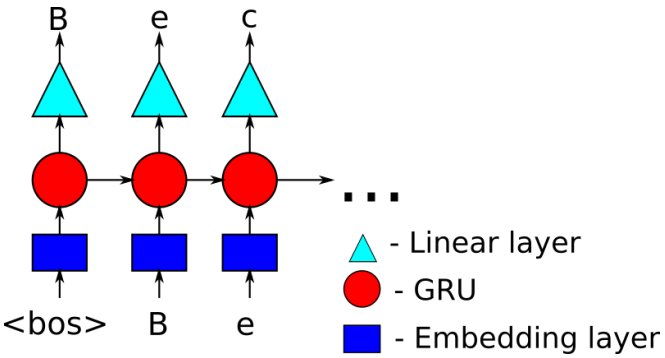
(C) On the other hand, when training *M4* in Question 1, stop words were removed. When using *M4* for the classification task in Question 2, how do you think stop words in the documents to be classified should be handled when tokenising the documents? Justify your answer without running any experiments.

(D) You can run `text_classification.py` to train a logistic regression model for our text classification task and observe its performance on the test dataset. Now run experiments to verify your hypotheses for Part (A), Part (B) and Part (C) above. Please list the different settings you have tried and use a table to present the results you have obtained. Discuss whether your empirical results are aligned with your hypotheses in the earlier parts. If your earlier hypotheses earlier contradict the experiment results, explain what you think are possible reasons for the empirical results you have obtained.

(E) List two possible ways to train better word2vec embeddings for a topic-based classification task (assuming that you have unlimited storage space, computation power, and access to data) and briefly explain why you think they can potentially improve the quality of word2vec embeddings. You do not need to run any experiments for this Part (E).

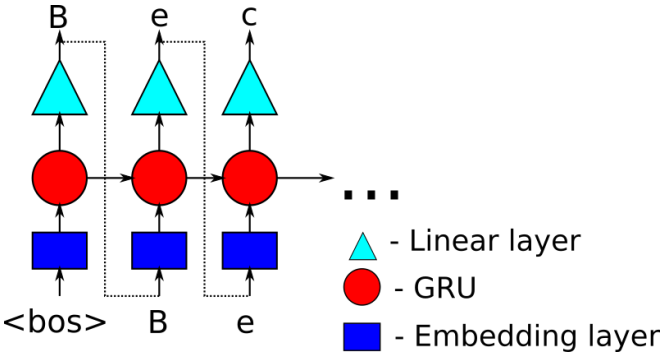
Question 3: RNNs for Sequence Modeling (10 marks)

Your task in this question is to develop an autoregressive RNN model which can generate people's names. The RNN will generate each character of a person's name given all previous characters. Your model should look like the following when training:



Note that the input is shown here as a sequence of characters but in practice the input will be a sequence of character ids. There is also a `softmax` non-linearity after the linear layer but this is not shown in the diagram. The output (after the `softmax`) is a categorical probability distribution over the vocabulary, what is shown as the output here is the ground truth label. Notice that the input to the model is just the expected output shifted to the right one step with the `<bos>` (beginning of sentence token) prepended.

The three dots to the right of the diagram indicate that the RNN is to be rolled out to some maximum length. When generating sequences, rather than training, the model should look like the following:



Specifically, we choose a character from the probability distribution output by the network and feed it as input to the next step. Choosing a character can be done by sampling from the probability distribution or by choosing the most likely character (otherwise known as `argmax` decoding).

The character vocabulary consists of the following:

`''''` The null token padding string.

`<bos>` The beginning of sequence token.

`.` The end of sequence token.

`a-z` All lowercase characters.

`A-Z` All uppercase characters.

`0-9` All digits.

`'' ''` The space character.

Part I (7 marks)

Starter code is provided in `rnn_name_generator.py`, and the list of names to use as training and validation sets are provided in `names_small.json`.

To complete this part of the question, you will need to complete three functions and one class method: the function `seqs_to_ids`, the `forward` method of class `RNNLM`, the function `train_model`, and the function `gen_string`. In each case you should read the description provided in the starter code.

`seqs_to_ids`: Takes as input a list of names. Returns a 2D numpy matrix containing the names represented using token ids. All output rows (each row corresponds to a name) should have the same length of `max_length`, achieved by either truncating the name or padding it with zeros. For example, an input of: [“`Bec.`”, “`Hannah.`”, “`Siqi.`”] with a `max_length` set to 6 should return (normally we will use `max_length = 20` but for this example we use 6)

```
[[30 7 5 2 0 0]
[36 3 16 16 3 10]
[47 11 19 11 2 0]]
```

Where the first row represents “`Bec.`” and two padding characters, the second row represents “`Hannah`”, the third row represents “`Siqi.`” with one padding character.

`forward`: A method of class `RNNLM`. In this function you need to implement the GRU model shown in the diagram above. The layers have all been provided for you in the class initialiser.

`train_model`: In this method you need to train the model by mini-batch stochastic gradient decent. The optimiser and loss function are provided to you. Note that the loss function takes logits (output of

the linear layer before `softmax` is applied) as input. At the end of every epoch you should print the validation loss using the provided `calc_val_loss` function.

`gen_string`**:** In this method you will need to generate a new name, one character at a time. You will also need to implement both sampling and `argmax` decoding.

Your code should all be in the original `rnn_name_generator.py` file. Other files will not be marked. For this question **you should not import additional libraries**. Use only those provided in the starter code. (You may uncomment the import `tqdm` statement if you want to use it as a progress bar.)

In your **answers PDF**, please answer the following questions.

(A) After training, what is the most likely name your model generates using `argmax` decoding.

(B) After training, generate 10 different names using sampling and include them in your answers PDF. (For example, one of the names sampled from the model solution was `Dasbie Miohmazie`.)

Part II (3 marks)

If the names used for training have been grouped by their origins (e.g., German names, Chinese names, Vietnamese names, Italian names, etc.), how would you train your name generator differently in order to generate more realistic names? Justify your answer, but you do not need to run any experiments.

Question 4: Sentiment Classification (12 marks)

For this question you have been provided with a labelled movie review dataset. The dataset consists of 50,000 review articles written for movies on IMDb, each labelled with the sentiment of the review – either positive or negative. The overall distribution of labels is balanced (25,000 `pos` and 25,000 `neg`).

Part I: Training a Transformer-based Sentiment Classifier (9 marks)

Your task in this part is to train a transformer-based classifier to predict the sentiment label from the review text.

Specifically, you will train a transformer encoder using `PyTorch`. An input sequence is first tokenised and a special `[CLS]` token prepended to the list of tokens. Similar to BERT[1], the final hidden state (from the transformer encoder) corresponding to the `[CLS]` token is used as a representation of the input sequence in this sentiment classification task.

First, implement the `get_positional_encoding` function in `sentiment_classifier.py`. This function computes the following positional encoding:

$$\mathrm{PE}_{t,2i} = \sin\left(\frac{t}{10000^{\frac{2i}{d}}}\right), \quad \mathrm{PE}_{t,2i+1} = \cos\left(\frac{t}{10000^{\frac{2i}{d}}}\right), \quad t \in \{0, \ldots, T-1\},$$

where d is the dimension, T the length of the sequence and $i \in \{0, \ldots, \frac{d}{2} - 1\}$. You can assume d is an even number for this question.

Next, complete the `__init__` method of class `SentimentClassifier` by creating a `TransformerEncoder`[2] consists of a few (determined by parameter `num_tfm_layer`) `TransformerEncoderLayer`[3] with the specified input dimension (`emb_size`), number of heads (`num_head`), hidden dimension of the feedforward sub-network (`ffn_size`) and dropout probability (`p_dropout`).

[1] `https://en.wikipedia.org/wiki/BERT_(language_model)`

[2] `https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html`

[3] `https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoderLayer.html`

Now implement the `forward` method of class `SentimentClassifier`. You should implement the following steps sequentially in the function:

1. add the positional encoding to the embeddings of the input sequence;
2. apply dropout (i.e. call the dropout layer of `SentimentClassifier`);
3. call the transformer encoder you created in the `__init__` method; (Hint: make sure you set the parameter `src_key_padding_mask` to an appropriate value.)
4. extract the final hidden state of the `[CLS]` token from the transformer encoder for each sequence in the input (mini-batch);
5. use the linear layer to compute the *logits* (i.e. unnormalised probabilities) for binary classification, and return the *logits*.

You should read the documentation before implementing these methods.

Lastly, complete the `train_model` function to learn the classifier using the training dataset. You should optimise the model parameters through mini-batch stochastic gradient descent with the `Adam`[4] optimiser. Make sure you evaluate the model on the validation set after each epoch of training, and save the parameters of the model that achieves the best accuracy on the validation set.

You are now able to run `sentiment_classifier.py` which prepares the training, validation and test datasets, trains a sentiment classifier and evaluates its accuracy on the test set. Note that this approach does not directly use the combined training and validation datasets to re-train the model, and we adopt it here for simplicity. Tuning hyper-parameters systematically is not required for this question (although you are free to do so).

Depending on whether you are using GPU or CPU and the computing resources you have, training the model may take a long time on your machine. You are free to reduce the number of epochs in the main program of `sentiment_classifier.py` to manage your training time. This question will not be marked based on the accuracy level you are able to achieve, although we expect correct implementation of the code.

In your answers PDF, report the final accuracy on the test set achieved by the model you have trained.

Part II: LLM for Sentiment Classification (3 marks)

In this part, based on what you have learned in class and your own experience using LLMs, you will use ChatGPT to perform sentiment classification.

Randomly select 10 examples from the dataset, i.e., 10 movie reviews from the data we have provided. Use a prompt to ask ChatGPT to predict the sentiment label of the movie review. Report the prompt you use in your answers PDF. Compute the accuracy of ChatGPT on the 10 examples, i.e., the percentage of examples that are classified correctly by ChatGPT. Report this accuracy.

Use no more than 10 sentences to explain why ChatGPT (or pre-trained large language models in general) can perform very well on this task compared with the Transformer-based classifier you have trained in Part (A), although both are based on a Transformer architecture.

[4] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`