# Guide to predicting the author of a text by n-gram inputs on a simple network

Ling 441

October 6, 2025

## 1 Overview

You don't necessarily need to follow these suggestions exactly, (with the exception of the sys.argv code below which must be followed) but they are here to give you help if you need it. If you choose a different way to code the assignment, make sure that you get the required results.

The plan is to create a neural model that trains on five books, learning to (try to) assign the correct author that corresponds to each n-gram representation that is fed into the network from the books. Then, based on training, it tries to predict the author for a held-out test set for each author, where the test set was not trained on.

One absolute requirement of the code you submit is that it needs to allow a command-line argument that can alter the number of iterations for which you train. The reason for this is so that I can do a preliminary run of everyone's submissions with a drastically reduced number of iterations on each so that it doesn't take an hour and a half to check if people's code runs without errors.

If you are developing on Colab, include the code in this Colab page.

```
import sys
if len(sys.argv) > 2 or len(sys.argv) == 1:
    num_iter = 1000 #Replace 1000 with your own choice of number
else:
    num_iter = int(sys.argv[1]) #This MUST be wrapped in int()
```

If developing a .py file directly, you just need the following, assuming that you are not using any of your own command-line arguments in addition:

```
import sys
if len(sys.argv) == 1:
    num_iter = 1000 #Replace 1000 with your own choice of number
else:
    num_iter = int(sys.argv[1]) #This MUST be wrapped in int()
```

A simple way to code this is, in testing, to assign a probability score, using softmax, for each author for each n-gram that is encountered. The probabilities can then be converted to logarithms so that the log probs can be added up for each test set. This has the same effect as multiplying the raw probabilities. We need to use logs here to avoid underflow. The prediction for each test set will be the aggregate log prob score with the highest value (i.e. lowest magnitude negative number.)

In order to have the softmax function and log calculation available, you can create an instance of the softmax class in the initialization of your model with:

```
        self.softmax = nn.Softmax(dim=1)
```

You'll need dim=1 if you have a batch size of 1, since, after unsqueezing the outputs, the batch is dim 0.

For converting probs to log probs, just wrap the output in torch.log(): log_probs = torch.log(model.softmax(pred))

## 2 Hyperparameters and running time

Set the following hyperparameters:

1. The number of iterations you are going to run in training, set with the sys.argv conditional described above.
2. The dimension of the word embeddings.

3. The learning rate.
4. The value of n: i.e. trigrams if n=3, quadrigrams if n=4, etc.

Suggested: 50000 data points, embedding_dim 32, lr=0.003, n=4, but you can try changing these.

The main limitation is that your code should take no more than three minutes for me to run it on a machine with an AMD Athlon Gold 3150U, CPU max MHz: 2400.0000, CPU min MHz: 1400.0000. With the above hyperparameters, I found that it took three minutes, including printing out training accuracy every 1000 data points. On Colab, it took a bit longer: three-and-a-half minutes. An hour and a half is the maximum it should take to run the code of 30 students.

## 3 Preparing data

When going through each of the five training files you should:

1. lower-case each line
2. substitute all non-alphabetic characters with spaces except for regular apostrophes
3. split each line into words
4. continue to the next line if the line has fewer than n words
5. add each word not already encountered in any of the five train files to a global vocabulary list.
6. add each n-gram in the line to a list of words that is the value of a dictionary whose key is the author. You should do this every time an n-gram is encountered without checking if the same n-gram has been encountered before.

After going through all the training files, add one more word to the global vocabulary list, with a name like 'OOV' for 'out of vocabulary'. This will be needed when encountering words in a test set that were never encountered in any of the train files. (But see below for an alternative.)

At this point you can either go through the test files and create n-grams for them, or else do it when it comes to testing. When creating an n-gram, make sure, for each word, to check if it is the global vocabulary list, and, if not, add it as 'OOV' rather than the actual word.

An alternative to the 'OOV' route is to also add the words in the test files to the global vocabulary list. In the OOV case, each word in a test set that was not trained on will have the **same** random embedding vector and it will not be modified in training, since it won't be encountered. On the other hand, if these words are each given a different embedding by putting them into the global vocab list, each will have a **different** random embedding that doesn't get modified. It's hard to say which alternative works better and you can choose one or the other.

Whichever route you take, you'll need to assign an index to each vocabulary item so that an embedding can be assigned to each. You can do that with a dictionary wd2ix = {}, which you can create by enumerating through the vocab list and giving the value torch.tensor(i) to each word's key.

You should also set up a dictionary that assigns an index to each of the five authors.

## 4 Setting up the model

The model shouldn't be too different from what you did for the German definite article with the following additions:

- In __init__():
  - You'll also need to create an instance of the nn.Embedding class with a line like self.embeddings = nn.Embedding(num_wds, embedding_dim), where num_wds is the number of vocabulary items.
  - You'll need a line like the one mentioned earlier to create an instance of the nn.Softmax class.
  - Assuming that you'll just have a single linear layer, when you set it up with an instance of nn.Linear(), its input dimension should be the embedding dimension times the value of $n$ which is the number of words in an n-gram.

    - Its output dimension should be 5, since we have five different authors to classify on.

- in forward():
    - The input to forward() can be an n-gram. To get the vector of the input n-gram you can use torch.cat() to concatenate the embeddings of each of the words in the n-gram, where each embedding is self.embeddings(wd2ix[ngram[$i$]]), and $i$ is the index of the word within the n-gram.
    - Note that torch.cat() takes a tuple as its main argument, so you need to wrap the listed embeddings in an inner set of round brackets.
    - You'll also need to give torch.cat() the flag axis=0 (outside that inner set of brackets) so that it concatenates them into one strung-out vector.

## 5 Optional enhancements

If you want to try adding more layers,, with non-linearities in between, that's fine, but keep in mind that this could make the code take longer to run, and some hyperparameters may need to be changed in order to avoid your code taking longer than three minutes to run.

## 6 Training

First set up the loss function and optimizer. An optional enhancement to the optimizer, to prevent over-fitting, is to add the argument weight_decay=x, where x is roughly one-tenth of the learning rate.

It is highly recommended that you train on random sequences of n-grams from the five training sets. A simple way to do this is to create each data point by first randomly choosing an author with random.choice and then randomly choosing a trigram from that author's list of trigrams, again with random.choice(). Rather than having epochs, where we visit each trigram once in each epoch, here we have a roughly equal chance of encountering each trigram each time. If there are around 25,000 data points and we train randomly on 50,000, each data point has roughly an 87% chance of being visited at least once.

After each 1,000 training examples, have it print out the accuracy on the past 1,000. If you can get up to 70% of a set of 1,000 predicting the correct author, that should be enough learning for testing. Otherwise, in training, use the same kind of code that we have used in the past.

## 7 Testing

Go through each test set one at a time, create an aggregate log prob vector, initialized as 0. (when you add each log prob set to it, it should broadcast to the right dimensions), and, in the scope of torch.no_grad() and instead of counting correct predictions, have the output of the model for each data point softmaxed and then changed to the log of each value to add to the aggregate log prob vector. At the end of the testing of each author, the prediction is the author with the highest score. You can get the index with:

```
torch.argmax(aggregate_log_probs, axis=1).item()
```

You should be able to get at least four of the five authors correctly predicted. There seems to be a similarity between the train set of Elizabeth Barrett Browning and the test set of Samuel Coleridge to the extent that Coleridge's test set often gets mis-predicted as being by E.B. Browning.