CS187 Lab 2-3: Language modeling with neural networks

September 22, 2023

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab2-3.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

In lab 2-1, you built and tested n-gram language models. Recall that some problems with n-gram language models are:

1. They are profligate with memory.
2. They are sensitive to very limited context.
3. They don't generalize well across similar words.

As promised, in this lab, you'll explore neural models to address these failings. You will:

1. Build and test a neural n-gram language model.
2. Build and test a neural RNN language model.
3. Use language models for classification (*Federalist Papers* author identification).

Preparation – Loading packages and data

[ ]:
```
import json
import math
import random
import wget

import torch

from tokenizers import Tokenizer
from transformers import PreTrainedTokenizerFast

from tqdm.auto import tqdm
```

[ ]:
```
# Set random seeds
SEED = 1234
torch.manual_seed(SEED)
random.seed(SEED)

# GPU check, sets runtime type to "GPU" where available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

The corpus used throughout this lab is the *Federalist* papers. We've trained and provided neural language models on papers authored by Hamilton and Madison, respectively, which we download here.

[ ]:
```
# Prepare to download needed data
def download_if_needed(source, dest, filename):
    os.path.exists(f"./{dest}{filename}") or wget.download(source + filename,␣
 ↪out=dest)

source_path = "https://github.com/nlp-course/data/raw/master/Federalist/"
data_path = "data/"

os.makedirs(data_path, exist_ok=True)
# Download files, including pretrained language models
for filename in ["federalist_data_raw2.json",
```

```
                  "tokenizer.pt",
                  # language models:
                  # Hamilton        Madison
                  "ffnn_lm_h.pt",   "ffnn_lm_m.pt", # feedforward NN
                  "rnn_lm_h.pt",    "rnn_lm_m.pt"   # RNN
                 ]:
    download_if_needed(source_path, data_path, filename)

# Read in the raw data
dataset = json.load(open(data_path + "federalist_data_raw2.json"))

# Read in the pretrained tokenizer
hf_tokenizer = torch.load(data_path + "tokenizer.pt")
```

First, let's split the dataset into training, validation, and test sets. Since we have provided pretrained models, you won't be using the training set in this lab. In the problem sets you will have opportunities to train models yourself.

For this lab, we use a test set `testing`, which is the same as we used in lab 1-2. But for the validation set, we have separate ones for papers authored by Hamilton (`validation_hamilton`) and papers authored by Madison (`validation_madison`).

[ ]:
```
# Split training, validation, and test sets
TRAIN_RATIO = 0.9
# Extract the papers of unknown authorship
testing = list(filter(lambda ex: ex['authors'] == 'Hamilton or Madison',
                      dataset))
# Change gold labels in-place
for ex in testing:
  ex['authors'] = 'Madison'

# Extract the papers by Madison
dataset_madison = list(filter(lambda ex: ex['authors']=='Madison', dataset))
random.seed(SEED)
random.shuffle(dataset_madison)
training_size_madison = int(math.floor(TRAIN_RATIO * len(dataset_madison)))
validation_madison = dataset_madison[training_size_madison:]

# Extract the papers by Hamilton
dataset_hamilton = list(filter(lambda ex: ex['authors']=='Hamilton', dataset))
random.seed(SEED)
random.shuffle(dataset_hamilton)
training_size_hamilton = int(math.floor(TRAIN_RATIO * len(dataset_hamilton)))
validation_hamilton = dataset_hamilton[training_size_hamilton:]

# We only consider the first 200 tokens of each document for speed
def truncate(document_set, k=200):
  for document in document_set:
```

```
    document['tokens'] = document['tokens'][:k]
truncate(validation_madison)
truncate(validation_hamilton)
truncate(testing)

print (f"Madison validation size:  {len(validation_madison)} documents\n"
       f"Hamilton validation size: {len(validation_hamilton)} documents")
```

Note that, unlike in labs 1-2 and 1-3, here we consider *all* word types in the data. Let's look at an example:

[ ]:
```
print (f"Example (Madison): {validation_madison[0]['tokens']}\n\n"
       f"Example (Hamilton): {validation_hamilton[0]['tokens']}")
```

1 The n-gram feedforward network

In lab 2-1, you built an n-gram language model using a lookup table. However, that model assigns zero probability to any n-gram that doesn't appear in the training text (without smoothing). In this lab, we consider a neural-network-based approach, which can address this issue.

Recall that in n-gram language modeling, we made the assumption that the probability of a word only depends on its previous $n-1$ words:

$$\begin{aligned}\Pr(w_1, w_2, \ldots, w_M) &= \Pr(w_1) \cdot \Pr(w_2, \ldots, w_M \mid w_1) \\ &= \Pr(w_1) \cdot \Pr(w_2 \mid w_1) \cdot \Pr(w_3 \ldots, w_M \mid w_1, w_2) \\ &\cdots \\ &= \prod_{i=1}^{M} \Pr(w_i \mid w_1, \cdots, w_{i-1}) \\ &\approx \prod_{i=1}^{M} \Pr(w_i \mid w_{i-n+1}, \cdots, w_{i-1}),\end{aligned}$$

and we used the empirical frequencies to estimate these conditional probabilities:

$$\Pr(w_i \mid w_{i-n+1}, \cdots, w_{i-1}) = \frac{\sharp(w_{i-n+1}, \cdots, w_{i-1}, w_i)}{\sum_{x'} \sharp(w_{i-n+1}, \cdots, w_{i-1}, x')}$$

We can immediately see the problem with using a large n: the numerator would be 0 for any n-grams unseen in the training data.

One way of solving this issue is to use a "smoother" function. We parameterize the conditional probabilities using a neural network that computes a function f, which we use to estimate the probabilities

$$\Pr(w_i \mid w_{i-n+1}, \cdots, w_{i-1}) \approx f_i(w_{i-n+1}, \cdots, w_{i-1}),$$

where f is a function returning a vector of size V (V being the vocabulary size). The j-th element of the returned vector f_j stores the probability of generating the j-th word in the vocabulary. (We're being a little fast and loose with notation here. Strictly speaking, we should define these probability estiamets in terms of word ids, that is, indices into the vocabulary.)

To specify f, we can use a feedforward neural network. We'll represent words not by their ids or a one-hot representation but isntead we map each word type in the vocabulary to a trainable vector called an *embedding* of size `embedding_size`.

Why do we represent words with such embeddings? To answer this question, let's consider two alternative representations: (1) word indices and (2) one-hot vectors (which we used in lab 1-1). (We cannot directly use the strings themselves because they are of varying lengths.) A desirable word representation system should be such that *the similarity of words can be reflected in the closeness of word representations*. Ideally, if two words have similar meaning and syntactic function, they should have similar representations, in order to alleviate the burden of learning such similarities by the rest of the model. For option (1), closeness in terms of word indices is meaningless: the 365-th word in the vocabulary is probably not more similar to the 366-th word than it is to other words, since the assignment of index in the vocabulary is arbitrary. For option (2), two different word types always have orthogonal vector representations, but we would hope that similar words can be placed near each other (at least we don't want to eliminate that possibility from the beginning).

Therefore, we use an embedding, a vector representation for each word type in the vocabulary, which has been separately learned in a manner that has been shown to cluster similar words together. There are many such embeddings; the particular embedding we'll use is *word2vec*, a mapping from words to vectors of embedding size 128 trained under a task called "masked language modeling". If you are interested in more details, you should read the original word2vec paper. For our purposes, we can treat the embedding as just given to us.

Now let's get back to the parameterization of $f(w_{i-n+1}, \cdots, w_{i-1})$. We first map each word in $\langle w_{i-n+1}, \cdots, w_{i-1} \rangle$ to its embedding $\langle x_{i-n+1}, \cdots, x_{i-1} \rangle$ ($n-1$ vectors each of size `embedding_size`), and we concatenate these embeddings to a vector (of size `(n-1) * embedding_size`). Then, we apply a linear projection to project it down to size `hidden_size`, followed by a nonlinear function, and another linear projection to project to size V, followed by a softmax to normalize to probabilities. In this case, the nonlinear function we use is not a sigmoid. Instead, we use a Rectified Linear Unit (ReLU), which is simply a componentwise function that clips negative numbers at zero:

$$ReLU(x) = \max(0, x)$$

We use $n = 5$ in this lab.

[ ]:
```
n = 5
```

Implement the missing part of the `forward` function below. This function takes the previous words (the entire previous history, not just the n-gram context) as input, and returns the probabilities of generating the next word (the target). (This design decision to take the entire history, even though the $n-1$-gram context is all that is needed, is for consistency with the RNN language model that will be introduced later, which does use the full context.)

The returned value should be a dictionary, with word types as keys and their respective probabilities as values.

[ ]:
```
class FFNNLM(torch.nn.Module):
  def __init__(self, n, tokenizer, embedding_size, hidden_size):
    super().__init__()
    self.n = n
    self.tokenizer = tokenizer
    self.vocab = self.tokenizer.get_vocab()
    vocab_size = len(self.vocab)
    self.pad_index = self.tokenizer.pad_token_id

    # Create modules
    self.embed = torch.nn.Embedding(vocab_size, embedding_size)           #␣
↪Embedding
    self.sublayer1 = torch.nn.Linear((n-1) * embedding_size, hidden_size) #␣
↪First layer
    self.sublayer2 = torch.nn.ReLU()                                      #␣
↪Second layer
    self.hidden2output = torch.nn.Linear(hidden_size, vocab_size)         #␣
↪Last layer

  def forward(self, history_words):
    """Computes the distribution over the next word given context␣
↪`history_words`.
    Arguments:
      history_words: a list of word strings; could be an empty list when␣
↪generating
                     the first word.
    Returns:
      the distribution over the next word given the context, stored as a␣
↪dictionary,
      with word types as keys, and probability values as values. The␣
↪probability of
      generating an unknown word is stored in `dictionary["[UNK]"]`"""
    # Switch to "evaluation" mode
    self.eval()
    # Convert strings to word ids
    context = self.tokenizer(history_words, \
                             is_split_into_words=True, \
                             return_tensors='pt')['input_ids'] \
                .long() \
                .to(device) # bsz = 1, context_len
    context_len = context.size(1)

    # For generating the first words, we feed in a special but arbitrary␣
↪beginning-of-sentence
    # symbol. Here, we'll use `"[PAD]"`, which is also what we use for padding.␣
↪In future
```

```
    # labs we'll be using other tokens for this purpose, but as long as␣
↪training and evaluation
    # use the same beginning-of-sentence symbol, it doesn't matter which␣
↪particular symbol we use.
    if context_len < self.n - 1:
      # Pad to the left if we don't have enough context words
      padding = context.new(1, self.n - 1 - context_len).fill_(self.pad_index)
      context = torch.cat([padding, context], 1)
    else:
      # TODO: Prepare proper context (the previous n-1 words) from the full␣
↪history
      context = ...

    embeddings = self.embed(context)            # 1, n-1, embedding_size
    embeddings = embeddings.view(1, -1)         # 1, (n-1) * embedding_size
    # TODO: finish feedforward and set logits from output
    # Logits should be a tensor of size (1, vocab_size)
    # The structure of the network is
    #   embeddings -> sublayer1 -> sublayer2 -> hidden2output -> softmax
    logits = ...

    # Normalize to get probabilities
    probs = torch.softmax(logits, -1).view(-1) # vocab_size

    # Match probabilities with actual word types
    distribution = {}
    for i, prob in enumerate(probs):
      word = self.tokenizer.decode(i)
      distribution[word] = prob.item()
    return distribution
```

Now, let's load the pretrained feedforward language models for Hamilton and Madison. The model `ffnn_lm_madison` was trained on documents authored by Madison, whereas `ffnn_lm_hamilton` was trained on documents authored by Hamilton.

[ ]:
```
# Create and load feedforward LM for Madison
ffnn_lm_madison = FFNNLM(n, hf_tokenizer,
                         embedding_size=128,
                         hidden_size=128
                        ).to(device)
ffnn_lm_madison.load_state_dict(torch.load(data_path + 'ffnn_lm_m.pt',␣
↪map_location=device))

# Create and load feedforward LM for Hamilton
ffnn_lm_hamilton = FFNNLM(n, hf_tokenizer,
                          embedding_size=128,
                          hidden_size=128,
```

```
                        ).to(device)
ffnn_lm_hamilton.load_state_dict(torch.load(data_path + 'ffnn_lm_h.pt',␣
 ↪map_location=device))
```

1.1 Sampling from an n-gram feedforward network

Recall from lab 2-1 that we can sample a sequence of text from a model using the functions below. Again, the `sample` function here takes as an argument the full context instead of just the previous $n-1$ words, for consistency with the later RNN model.

[ ]:
```
def sample(model, context):
    """Returns a token sampled from the `model` assuming the `context`"""
    distribution = model(context) # calls internally to model.forward(context)
    prob_remaining = random.random()
    for token, prob in sorted(distribution.items()):
        if prob_remaining < prob:
            return token
        else:
            prob_remaining -= prob
    raise ValueError

def sample_sequence(model, start_context, count=100):
    """Returns a sequence of tokens of length `count` sampled successively
       from the `model` starting with the `start_context`
    """
    random.seed(SEED) # for reproducibility
    context = list(start_context)
    result = list(start_context)
    for i in range(0, count):
        next = sample(model, tuple(context))
        result.append(next)
        context = context + [next]
    return result
```

Let's try to sample from our models. (Don't expect much fluency in the samples, since the dataset it is trained on is small.)

[ ]:
```
print(' '.join(sample_sequence(ffnn_lm_madison, ('constitution', 'proposed',␣
 ↪'by', 'the'))), "\n")
print(' '.join(sample_sequence(ffnn_lm_hamilton, ('constitution', 'proposed',␣
 ↪'by', 'the'))))
```

[ ]:
```
grader.check("ffnn_sample")
```

1.2 Evaluating text according to an n-gram feedforward network

Now let's use our language model to score text. Since the n-gram feedforward network is able to score with zero context – internally, it pads on the left with instances of the padding token that

the tokenizer provided – `ffnn_lm_hamilton.forward([])` will return the probability distribution $\Pr(x_1)$ for the first word in a document.

[ ]:
```
Pr_x1 = ffnn_lm_hamilton([])
topk = 9

# Sort by probabilities
for i, word in enumerate(sorted(Pr_x1, key=lambda word: Pr_x1[word],␣
 ↪reverse=True)[:topk]):
    print (f"top {i+1} word: {word:<8} Pr(x1): {Pr_x1[word]:.3f}")
```

Define a function `neglogprob` that takes a token sequence and a language model and returns the negative log probability of the *entire* token sequence according to the model (using log base 2). Note that you need to use the probability for the unknown word type `"[UNK]"` if a token does not appear in the vocabulary.

[ ]:
```
# TODO
def neglogprob(tokens, model):
    """Returns the negative log probability of a sequence of `tokens`
       according to a `model`
    """
    score = ...
    return score
```

[ ]:
```
grader.check("ffnn_neglogprob")
```

[ ]:
```
round(neglogprob(["constitution",], ffnn_lm_madison), 2)
```

Define a function `perplexity` that takes a token sequence and a language model and returns the perplexity of the *entire* token sequence according to the model.

[ ]:
```
# TODO
def perplexity(tokens, model):
    """Returns the perplexity of a sequence of `tokens` according to a `model`
    """
    ...
```

What's the perplexity of each document in the validation set under the language model trained on papers authored by Madison? What about Hamilton? Let's start with one document from each author.

[ ]:
```
document_madison = validation_madison[0]['tokens']
document_hamilton = validation_hamilton[0]['tokens']
```

Calculate the perplexity of each model on `document_madison` and `document_hamilton`.

[ ]:
```
# TODO
ppl_madison_model_madison_document = ...
ppl_hamilton_model_madison_document = ...
```

```
ppl_madison_model_hamilton_document = ...
ppl_hamilton_model_hamilton_document = ...
```

[ ]:
```
grader.check("ffnn_ppl")
```

Now, let's compare those perplexity values.

[ ]:
```
print (f"Author    Madison Model    Hamilton Model\n"
       f"Madison      {ppl_madison_model_madison_document:5.1f}          ␣
↪{ppl_hamilton_model_madison_document:5.1f}\n"
       f"Hamilton     {ppl_madison_model_hamilton_document:5.1f}          ␣
↪{ppl_hamilton_model_hamilton_document:5.1f}")
```

Question: What do you find? Why?

Type your answer here, replacing this text.

Now, let's revisit our motivation for parameterizing conditional probabilities using a feedforward neural network instead of through counting.

Question: Compare the pros and cons of feedforward neural language model and the original n-gram language model (possibly with smoothing). Which is better?

Type your answer here, replacing this text.

2 The recurrent neural network

One limitation of n-gram language models (both the original one and the neural one) is that they only model context up to a fixed number of words. However, natural language exhibits long-term dependencies, well beyond $n = 5$. In this part of the lab, we consider an approach based on recurrent neural networks (RNN), which can consider variable amounts of context.

Different from n-gram language modeling, RNN-based language models do not make the approximation that the probability of a word only depends on its previous $n - 1$ words. That is, we use the unapproximated chain rule:

$$\Pr(w_1, w_2, \ldots, w_N) = \prod_{i=1}^{N} \Pr(w_i \,|\, w_1, \cdots, w_{i-1})$$

and we again specify the conditional probabilities using a neural network:

$$\Pr(w_i \,|\, w_1, \cdots, w_{i-1}) = f(w_1, \cdots, w_{i-1}),$$

where we use an RNN to parameterize f. (Notice the change in the first index of the context, highlighted in red; we're using the whole history as context now, not just the last $n - 1$ words.)

The inputs to RNNs, like in the feedforward case, are embeddings of words, and we project the *final* output state of the RNN to a vector of size V, followed by a softmax to normalize the probabilities.

Implement the missing part of the `forward` function of an RNN language model below. This function takes the previous words as input, and returns the probabilities of generating the next word. The returned value should be a dictionary, with word types as keys and their respective probabilities as values.

> Hint: You might find torch.nn.RNN documentation helpful. Make sure that you understand the input and output shapes.

[ ]:
```
class RNNLM(torch.nn.Module):
  def __init__(self, tokenizer, embedding_size, hidden_size):
    super().__init__()
    self.tokenizer = tokenizer
    self.vocab = tokenizer.get_vocab()
    vocab_size = len(self.vocab)
    self.pad_index = self.tokenizer.pad_token_id

    # Create modules
    self.embed = torch.nn.Embedding(vocab_size, embedding_size)
    self.rnn = torch.nn.RNN(input_size=embedding_size,
                            hidden_size=hidden_size,
                            num_layers=1,
                            batch_first = True)
    self.hidden2output = torch.nn.Linear(hidden_size, vocab_size)

  def forward(self, history_words):
    """Computes the distribution over the next word given context␣
↪`history_words`.
    Arguments:
      history_words: a list of word strings; could be an empty list when␣
↪generating
                     the first word.
    Returns:
      the distribution over the next word given the context, stored as a␣
↪dictionary,
      with word types as keys, and probability values as values. The␣
↪probability of
      generating an unknown word is stored in `dictionary["[UNK]"]`"""
    # Switch to "evaluation" mode
    self.eval()
    # Convert strings to word ids
    context = self.tokenizer(history_words,
                             is_split_into_words=True,
                             return_tensors='pt'
                            )['input_ids'] \
                  .long() \
                  .to(device) # 1, context_len
    context_len = context.size(1)
```

```
    # For generating the first word, we feed in a special beginning-of-sentence␣
↪symbol <pad>,
    # which is also what we use for padding. In future labs we'll be using␣
↪<bos>, but as long
    # as training and evaluation use the same beginning-of-sentence symbol, it␣
↪doesn't matter
    # which particular symbol we use.
    if context_len == 0:
      context = context.new(1, 1).fill_(self.pad_index)
      context_len = context.size(1)

    # Initialize hidden
    hidden = None

    # TODO: finish feedforward and set logits
    # Logits should be a tensor of size (1, vocab_size)
    # Note that you should project the `output` from rnn, not the `hidden`,
    # using self.hidden2output
    # The structure of the network is
    #   embeddings -> the output of RNN at the last step -> hidden2output ->␣
↪softmax
    logits = ...

    # Normalize to get probabilities
    probs = torch.softmax(logits, -1).view(-1) # vocab_size

    # Match probabilities with actual word types
    distribution = {}
    for i, prob in enumerate(probs):
      word = self.tokenizer.decode(i)
      distribution[word] = prob.item()
    return distribution
```

Now, let's load the pretrained RNN language models for Hamilton and Madison. The model `rnn_lm_madison` was trained on documents authored by Madison, whereas `rnn_lm_hamilton` was trained on documents authored by Hamilton.

[ ]:
```
# Create and load RNN LM for Madison
rnn_lm_madison = RNNLM(hf_tokenizer,
               embedding_size=128,
               hidden_size=128,
               ).to(device)
rnn_lm_madison.load_state_dict(torch.load('data/rnn_lm_m.pt',␣
↪map_location=device))

# Create and load feedforward LM for Hamilton
rnn_lm_hamilton = RNNLM(hf_tokenizer,
```

```
                embedding_size=128,
                hidden_size=128,
                ).to(device)
rnn_lm_hamilton.load_state_dict(torch.load('data/rnn_lm_h.pt',␣
↪map_location=device))
```

2.1 Sampling from an RNN model

Let's try to sample from our models. The samples might be bad since the dataset is small.

[ ]:
```
print(' '.join(sample_sequence(rnn_lm_madison, ('constitution', 'proposed',␣
↪'by', 'the'))), "\n")
print(' '.join(sample_sequence(rnn_lm_hamilton, ('constitution', 'proposed',␣
↪'by', 'the'))))
```

[ ]:
```
grader.check("rnn_sample")
```

2.2 Evaluating text according to an RNN model

Again, let's evaluate the models on a document from Hamilton and an artitle from Madison. We'll select the first Madison document and the fisrt Hamilton document in the validation set.

[ ]:
```
document_madison = validation_madison[0]['tokens']
document_hamilton = validation_hamilton[0]['tokens']
```

Calculate the perplexity of each RNN model on each document.

[ ]:
```
# TODO
rnn_ppl_madison_model_madison_document = ...
rnn_ppl_hamilton_model_madison_document = ...
rnn_ppl_madison_model_hamilton_document = ...
rnn_ppl_hamilton_model_hamilton_document = ...
```

[ ]:
```
grader.check("rnn_ppl")
```

Now, let's compare those perplexity values.

[ ]:
```
print (f"Author      Madison Model        Hamilton Model\n"
       f"Madison        {rnn_ppl_madison_model_madison_document:5.1f}          ␣
↪     {rnn_ppl_hamilton_model_madison_document:5.1f}\n"
       f"Hamilton       {rnn_ppl_madison_model_hamilton_document:5.1f}          ␣
↪      {rnn_ppl_hamilton_model_hamilton_document:5.1f}")
```

Question: Which type of model is better? The RNN language models or the feedforward language models? What are the possible reasons?

Type your answer here, replacing this text.

3 Authorship attribution using language models

In lab 1-3, you saw how to use a Naive Bayes model to determine authorship:

$$\begin{aligned}\operatorname*{argmax}_i \Pr(c_i \mid \mathbf{x}) &= \operatorname*{argmax}_i \frac{\Pr(\mathbf{x} \mid c_i) \cdot \Pr(c_i)}{\Pr(\mathbf{x})} \\ &= \operatorname*{argmax}_i \Pr(\mathbf{x} \mid c_i) \cdot \Pr(c_i)\end{aligned}$$

In this lab, the language models trained on Madison documents can be used to calculate $\Pr(\mathbf{x} \mid \text{Madison})$, and the language models trained on Hamilton documents can be used to calculate $\Pr(\mathbf{x} \mid \text{Hamilton})$. Therefore, they can also be used for authorship attribution.

Recall that for numerical stability issues, we operate in log space (with base 2). With a little abuse of notation, let's denote the *log posterior* as

$$\log \Pr(\mathbf{x} \mid c_i) + \log \Pr(c_i),$$

where the priors $\Pr(c_i)$ from lab 1-3 are given below.

[ ]:
```
prior_madison = 15 / (15+51)
prior_hamilton = 51 / (15+51)
```

Let's consider a document from the test set.

[ ]:
```
document = testing[0]['tokens']
```

Use the feedforward neural language models to calculate the log posteriors for `document`.

[ ]:
```
#TODO - calculate the log posteriors for Madison and Hamilton using feedforward␣
↪LMs
log_posterior_madison_ffnn = ...
log_posterior_hamilton_ffnn = ...
#TODO - determine authorship
author_ffnn = ...
```

[ ]:
```
grader.check("ffnn_author")
```

[ ]:
```
print (author_ffnn)
```

Use the RNN neural language models to calculate the log posteriors for `document`.

[ ]:
```
#TODO - calculate the log posteriors for Madison and Hamilton using RNN LMs
log_posterior_madison_rnn = ...
log_posterior_hamilton_rnn = ...
#TODO - determine authorship
author_rnn = ...
```

[ ]:
```
grader.check("rnn_author")
```

[ ]:
```
print (author_rnn)
```

Now, we can use these models to determine authorship on the entire test set. Define the `ffnn_classify` and `rnn_classify` functions, which take a sequence of `tokens` and return either `'Hamilton'` or `'Madison'` depending on which of the two has a higher probability of authoring the text.

[ ]:
```
def ffnn_classify(tokens):
    """Returns the predicted author according to the FFNN model.
    Arguments:
      tokens: a list of tokens.
    Returns: 'Hamilton' or 'Madison'."""
    #TODO - implement this method
    ...

def rnn_classify(tokens):
    """Returns the predicted author according to the RNN model.
    Arguments:
      tokens: a list of tokens.
    Returns: 'Hamilton' or 'Madison'."""
    #TODO - implement this method
    ...

for ex in tqdm(testing):
    print(f"{ex['number']:2} {ffnn_classify(ex['tokens']):8}␣
 ↪{rnn_classify(ex['tokens']):8}")
```

[ ]:
```
grader.check("authorship")
```

Question: What would happen if the dataset is imbalanced, that is, if we have much more training data for one author than the other?

> Hint: With sufficient data, the model usually gets lower perplexity than with an insufficient amount of data.

Type your answer here, replacing this text.

4 Lab debrief – for consensus submission only

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of Lab 2-3

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```