SNLP assignment 6:
n-gram and RNN language models

Deadline: Aug 8, 2018 @ 10:00 CEST

This exercise set is about two families of language models, one using simple and interpolated n-grams, and another using recurrent neural networks.

We will use the essays written by the class members during assignment 0 as training data, which you can find under `train/` in your assignment repository.[1] Some of the test files in `test/` is also randomly selected from assignment 0 assays. However, there are also a few short surprise texts in this directory.

Implement both n-gram models (Exercise 2 and Exercise 3) in a single python script named `n-gram.py`, and the RNN models in Exercise 4 as another single python script `rnn.py`.[2]

Why am I doing this?

- Experiment with both the 'classic' n-gram language models and the 'neural' (RNN) language models
- Evaluate and compare different models
- A glimpse at the NLTK library

Exercise 1. *Word and sentence tokenization* (0.5 P.)

Write a function with name `tokenize()` that takes a variable number of file names and returns

- a tokenized corpus: a list whose members are sentences which in turn are lists of tokens
- a vocabulary: an ordered sequence with unique vocabulary items

Your function should convert all tokens to lowercase.

For this exercise, use the default NLTK sentence and word tokenizers to tokenize a given text file.

Exercise 2. *Ngram language model* (1.5 P.)

Implement a simple `Ngram` class with the following interface.[3]

`Ngram()` the constructor should accept one required argument, `n`, the order of the n-gram model.

`update()` updates the ngram counts for the given sentence. It should be, for example, callable like
`update(["I", "'m", "sorry", "Dave", ",", "."])`.

`prob()` returns the MLE probability of the given sequence.

`sprob()` returns the smoothed probability of a given sentence. For bigrams and higher-order n-grams, use Good-Turing discounting and for unigrams use Laplace (add-one) smoothing.[4]

`ppl()` returns the perplexity of the given list of sentences based on their smoothed probabilities as calculated by `sprob()`.

`next()` samples a word from the conditional distribution of given context. For example `next('I')` should return a random word w according to probability distribution $\mathrm{P}(w|\mathtt{I})$. Use the MLE distributions for this exercise.[5]

- Print out perplexity scores for all test files, using a unigram, bigram and trigram model.[6]
- Using each unigram, bigram and trigram models, generate 5 random sentences with the `next()` method you wrote.

[1] Essays are anonymized by removing some of the identifying information.

[2] Tokenization function (Exercise 1) can be in one of the files (and used by the other), or copied in both.

[3] Do not use any external libraries, e.g., NLTK, for this exercise. You are, however, encouraged to check your results against a well-known implementation.
This exercise overlaps with one of the last year's assignments. You can make use of two sample solutions provided in your assignment repository under `sample-code/`. However, make sure that you understand the part of the code you borrow, and **do not** include any code that is not required for the present exercise.

[4] Could you use If you wanted to use Good-Turing estimation for unigrams, how would you distribute the discounted probability mass to the unknown events?

[5] What would be the difference if you used the smoothed estimates of the distributions?

[6] What is the upper bound for the perplexity?

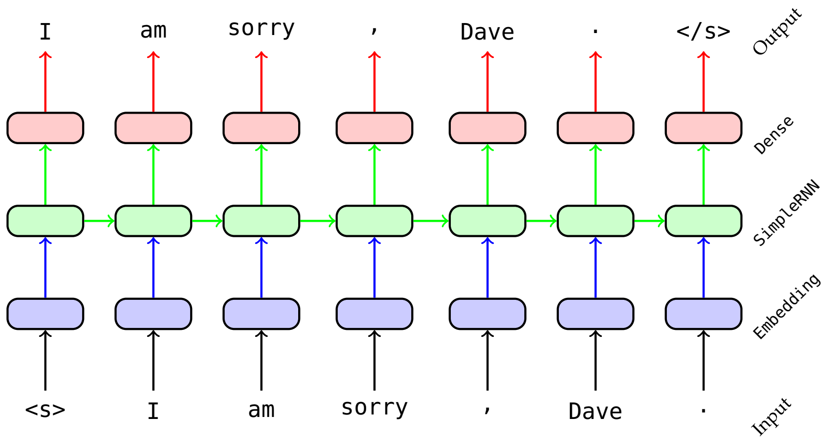
Exercise 3. *Interpolation* (1.5 P.)

- Write another Python class that implements an interpolated bigram model such that,

$$P(w_i|w_{i-1}) = (1-\lambda)P^*(w_i|w_{i-1}) + \lambda P^*(w_i)$$

  where λ is the Good-Turing discount for bigrams, $P^*(\cdot)$ is the smoothed estimations obtained by the n-gram model implemented in Exercise 2. Your class should implement the same interface as the class defined in Exercise 2.[7]
- Print out perplexity scores for all test files using your interpolated bigram model.

(optional) Using the interpolated bigram model, generate 5 random sentences.



```
1  model = Sequential((
2    Embedding(vocab_len, 32,
3              mask_zero=True,
4              input_length=max_len),
5    Dropout(0.5),
6    SimpleRNN(64,
7              return_sequences=True),
8    Dropout(0.5),
9    TimeDistributed(
10     Dense(vocab_len,
11           activation='softmax')
12   )
13 ))
```

Figure 1: A sketch of the RNN language model for Exercise 4. `vocab_len` is length of the vocabulary, including start- and end-of-sentence symbols, and a special symbol for 'unknown' words. You are also recommended not to use 0 for any of the symbols as it will be used as padding by Keras. `max_len` is the maximum sentence length in the training data. It is presumed here that all sentences are padded (or truncated) to this length. Note that for this model, the input is numeric sequences (each word is represented by an integer) but gold-standard output is one-hot vectors. The predictions of the model, at each time step, will be a categorical probability distribution over all words in the vocabulary.

Exercise 4. *An RNN language model* (2.5 P.)

- Train an RNN model that predicts next word in the sequence.

  You are free to build any model of your choice. The model and the Keras implementation outlined in Figure 1 can be used as a starting point. Note that unlike the RNN classifier you worked with in in assignment 5 where you only needed to use the final representation built for the whole sequence, here you need to use representations at each time step. The `TimeDistributed()` 'wrapper' on line 9 makes sure that the dense layer is connected to the output of each time step of the RNN layer.

  You are encouraged to build a better performing model, and tune it properly. However, it is not required for this exercise.[8] Otherwise, you can turn the code snippet in Figure 1 to a working example by processing/reshaping input properly, and (for the next step) understanding/using its output.
- Calculate and output the perplexity of the model on each of the training data sets.

(optional) Generate and output 5 random sentences from your RNN language model.

[7] The instructions here does not necessarily lead to an efficient implementation of n-gram models. However, please follow the specification here and when there is a choice between clarity and efficiency, choose clarity.

[8] How many classes do we have? What level of accuracy do you expect from the model? E.g., do you think an accuracy of 90 % achievable? Note that the achievable accuracy is related to the expected perplexity.