COMPX216 Assignment 4
Natural language processing using Markov chain n-gram models

Abstract

In this assignment, you will write functions that build Markov chain n-gram models from a corpus, use these models to predict the next token in a sequence, and compute the log likelihood of a sequence based on a model.

1 Tasks

There are five tasks you need to perform for this assignment. Download the provided `assignment4corpus.txt` file from Moodle and move it into the `aima-python` directory. This is the corpus we will use in this assignment. It is source from https://www.gutenberg.org/ebooks/15164. Download the provided `assignment4.py` file from Moodle and move it into the `aima-python` directory. This file contains the skeleton code for this assignment. You can follow the comments in this code to complete the assignment. Test code for each task is provided in the lower half of the file. After completing a task, uncomment the corresponding block of test code by removing the `'''` marks around it to test it in your run. Do not modify the names of classes, functions, or variables provided in the file. This is the file you will submit for marking. It is permitted to import additional modules required by your implementation.

Once you have written the code for a task, uncomment its test code and run the assignment code in a command-line terminal as shown below. Remember to ensure that your working directory is `aima-python`, and your virtual environment is activated.

```
python assignment4.py
```

1.1 Task 1

Your first task is to write functions that build Markov chain n-gram models as dictionaries. You will start with building a unigram model, then a bigram model, and lastly an n-gram with $n \in \mathbb{Z}^+$.

1.1.1 Task 1.1

The `build_unigram()` function takes a token list named `sequence` as input and returns a unigram model represented as a dictionary.

The returned dictionary should contain only one key-value pair, as a unigram model does not depend on a context. The key should be an empty tuple `()`, and the value should itself be a dictionary whose keys make up the vocabulary, and values are frequencies of their corresponding keys.

For example, given a sequence `['a', 'b', 'c', 'b']`, the unigram model should be represented as follows.

```
{
  ():
    {
      'a': 1,
      'b': 2,
      'c': 1
    }
}
```

1.1.2 Task 1.2

The `build_bigram()` function takes a token list named `sequence` as input and returns a bigram model represented as a dictionary.

The returned dictionary should contain a key-value pair for each observed context. Each key should be a tuple containing a context, and the corresponding value should be a dictionary whose keys are tokens that follow the context, and values are frequencies of their corresponding keys.

For example, given a sequence `['a', 'b', 'c', 'b']`, the bigram model should be represented as follows.

```
{
  ('a',):
    {
      'b': 1
    }
  ('b',):
    {
      'c': 1
    }
  ('c',):
    {
      'b': 1
    }
}
```

1.1.3 Task 1.3

The `build_n_gram()` function takes a token list named `sequence` and a natural number `n` as input and returns an n-gram model represented as a dictionary.

The returned dictionary should contain a key-value pair for each observed context. Each key should be a tuple containing a context, and the corresponding value should be a dictionary whose keys are tokens that entail the context, and values are frequencies of their corresponding keys.

For example, given a sequence `['a', 'b', 'c', 'b']` and $n = 3$, the trigram model should be represented as follows.

```
{
  ('a', 'b'):
    {
      'c': 1
    }
  ('b', 'c'):
    {
      'b': 1
    }
}
```

1.2 Task 2

Your second task is to write the `query_n_gram()` function that takes an n-gram `model` and a token tuple `sequence` as input and returns the dictionary corresponding to this context `sequence`.

If the model is unigram, the context is irrelevant, and the dictionary corresponding to the empty tuple should be returned. For example, the unigram model example above should always return the following.

```
{
  'a': 1,
  'b': 2,
  'c': 1
}
```

If $n \geq 2$, the dictionary corresponding to the context should be returned if the context exists as a key in the model. `None` should be returned if otherwise. For example, the trigram model example above should return `{'b': 1}` if the context `sequence` is `('b', 'c')`, and it should return `None` if the context `sequence` is `('a', 'c')`.

1.3 Task 3

Your third task is to write the `blend_predictions()` function that takes a list of predictions `preds` from multiple models and a blending `factor` as input and returns a dictionary as blended predictions.

Each prediction in `preds` is a dictionary such as one returned by `query_n_-gram()`. Predictions that are `None` in `preds` should be removed. The remaining predictions should first be normalised into probabilities sum up to `1`, and then they should be blended according to the given blending `factor`. The first prediction should weigh `factor`, the second prediction should weight `factor` of the remainder weight, and so on. The last prediction should weigh the remainder weight.

For example, given three predictions and a factor of 0.8, the first prediction should weigh 0.8, the second $(1 - 0.8) \cdot 0.8 = 0.16$, and the third $(1 - 0.8) \cdot (1 - 0.8) = 0.04$.

The returned blended prediction should have probabilities that sum up to `1`.

1.4 Task 4

Your fourth task is to write the `predict()` function that takes a token list named `sequence` and a list of `models` as input and returns a token sampled from the `models`' blended predictions.

This function should utilise `query_n_gram()` and `blend_predictions()` from the previous two tasks. You need to check if the context `sequence` is of sufficient size for each model before querying a model for its predictions. For example, a `sequence` of three tokens is not sufficient for a 5-gram model, so a 5-gram model should not be used. On the other hand, it is longer than what a trigram model requires, so only its last two tokens should be used to query the trigram model. After obtaining predictions from the models, the function should blend the predictions. Your code can rely on the default `factor` in `blend_predictions()`, and you do not need to use a different value for `factor`. The function should return a token randomly chosen based on the blended prediction's probabilities.

1.5 Task 5

Your fifth task is to write functions that compute log likelihood of a token sequence using n-gram models.

1.5.1 Task 5.1

You need to write the `log_likelihood_ramp_up()` function that takes a token list named `sequence` and a list of n-gram `models` as input and returns log likelihood of the `sequence` based on the `model` using the chain rule.

We assume the list of n-gram models are in decrementing order for n and end with a unigram model. The unigram model, i.e., the last in the list, is used to compute the likelihood of the first token, the bigram model, i.e., the second to last in the list, is used to compute the likelihood of the second token, and so on. This "ramp-up" process continues until we reach the first model in the list with the highest n, and from this point onward we keep using this first model.

We compute the log likelihood of each token in the `sequence` based on its context using a specific n-gram model as described above and return the sum of the log likelihoods. The log likelihood is the logarithm of the probability of the token following its $(n-1)$-token context. If a sub-sequence does not exist in the model, i.e., the $(n-1)$-token context or the token following the context do not correspond to a known combination in the model, its likelihood is computed as `0`, and the returned sum should be `-math.inf` regardless of the other sub-sequences.

1.5.2 Task 5.2

You need to write the `log_likelihood_blended()` function that takes a token list named `sequence` and a list of n-gram `models` as input and returns log likelihood of the `sequence` based on the `models` using the chain rule.

We compute the log likelihood of each token in the `sequence` based on its context and return the sum of the log likelihoods. The log likelihood is the logarithm of the blended probability of the token following its prior context. You can use `blend_predictions()` from Task 3 to compute the blended probability of a token in a sequence.