Branch: model nltk / nltk / test / model.doctest Find file Copy path

Copper-Head Unigram counts now also keep track of words in sentence-initial conte... f5b7c47 17 days ago

1 contributor

332 lines (246 sloc) 10.5 KB Raw Blame History

```
.. Copyright (C) 2001-2015 NLTK Project
.. For license information, see LICENSE.TXT

.. -*- coding: utf-8 -*-

===============
Counting Ngrams
===============

Building a Vocabulary
---------------------

    >>> from nltk.corpus import gutenberg
    >>> sents = gutenberg.sents("burgess-busterbrown.txt")
    >>> test_sents = sents[3:5]
    >>> test_words = [w for s in test_sents for w in s]
    >>> test_words[:5]
    ['Buster', 'Bear', 'yawned', 'as', 'he']

    >>> from nltk.model import build_vocabulary

The first argument to the `build_vocabulary` function is a cutoff value.

    >>> vocab = build_vocabulary(2, test_words)

This function returns an instance of NgramModelVocabulary, its characteristics
are described below.
Firstly, tokens with counts greater than or equal to the cuttoff value will
be considered part of the vocabulary.

    >>> vocab['the']
    3
    >>> 'the' in vocab
    True
    >>> vocab['he']
    2
    >>> 'he' in vocab
    True

Tokens with frequency counts less than the cutoff value will be considered not
part of the vocabulary even though their entries in the count dictionary are
preserved.

    >>> vocab['Buster']
    1
    >>> 'Buster' in vocab
    False
    >>> vocab['aliens']
    0
    >>> 'aliens' in vocab
    False

Keeping the count entries for seen words allows us to change the cutoff value
without having to recalculate the counts.

    >>> vocab.cutoff = 1
```

```
    >>> "Buster" in vocab
    True
    >>> "aliens" in vocab
    False

The cutoff value influences not only membership checking but also the result of
getting the size of the vocabulary using the built-in `len`.
Note that while the number of keys in the vocab dictionary stays the same, passing
it to `len` yields different numbers depending on the cutoff

    >>> len(vocab.keys())
    37
    >>> len(vocab)
    38
    >>> vocab.cutoff = 2
    >>> len(vocab)
    8

Note also that we add 1 to the size of the vocabulary, so even when cutoff=1
`len(vocab) > len(vocab.keys())`. This is done because in many language modeling
algorithms the size of the vocabulary is used for normalizating scores and because
it needs to account for the unknown token label.
This is definitely not an uncotroversial design choice, one that may be reverted
based on discussion and usage.

Vocabulary from multiple sources
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The `build_vocabulary` function can handle multiple text arguments, i.e.

    >>> test_words2 = sents[5]
    >>> test_words3 = sents[6]
    >>> vocab2 = build_vocabulary(2, test_words, test_words2, test_words3)
    >>> len(vocab2)
    14


Counting Ngrams
---------------

    >>> from nltk.model import count_ngrams

The first argument to `count_ngrams` is the highest ngram order to consider, the
second is an NgramModelVocabulary instance, followed by one (or more) texts.
A text is a sequence of sentences, eg. a list of lists of strings.

    >>> bigram_counts = count_ngrams(2, vocab, sents[3:7])

This returns an instance of an NgramCounter class which provides an interface
to the ngram counts.

For all non-unigram ngrams (order > 1) the counts are stored in the `ngrams`
attribute indexed by ngram order.

    >>> bigram_counts.ngrams[2]
    <ConditionalFreqDist with 9 conditions>

The keys of this ConditionalFreqDist are the contexts preceding a word.

    >>> sorted(bigram_counts.ngrams[2].conditions()) # doctest: +NORMALIZE_WHITESPACE
    [('.',), ('<UNK>',), ('<s>',), ('and',),
    ('he',), ('his',), ('the',), ('to',), ('yawned',)]

Each context has a FreqDist of tokens found following it in the text.

    >>> bigram_counts.ngrams[2][('.',)]
    FreqDist({'</s>': 4})

Accessing unigram counts works a little differently because they don't have any
preceding contexts. They are namely stored as a simple FreqDist in the `unigrams`
attribute.

    >>> print(bigram_counts.unigrams)
    <FreqDist with 10 samples and 121 outcomes>
```

```
    >>> expected_unigram_counts = {'.': 4,
    ... '</s>': 4,
    ... '<s>': 4,
    ... '<UNK>': 80,
    ... 'and': 5,
    ... 'he': 6,
    ... 'his': 5,
    ... 'the': 6,
    ... 'to': 4,
    ... 'yawned': 3}
    >>> bigram_counts.unigrams == expected_unigram_counts
    True

Tweaking the unknown label
~~~~~~~~~~~~~~~~~~~~~~~~~~

While counting ngrams all tokens that do not occur frequently enough (read: are
not "in" the NgramModelVocabulary instance) are replaced by a special "unknown word"
label. By convention this tends to be something like "<UNK>".
This is the default for the NgramCounter and it gets returned by
the `check_against_vocab` method for words that don't pass the cutoff.

    >>> bigram_counts.check_against_vocab('the')
    'the'
    >>> bigram_counts.check_against_vocab('aliens')
    '<UNK>'

This can be changed by passing a different value for the `unk_label` argument.

    >>> diff_unk = count_ngrams(2, vocab, sents[3:7], unk_label="UNKNOWN")
    >>> diff_unk.check_against_vocab('aliens')
    'UNKNOWN'


Changing the cutoff
~~~~~~~~~~~~~~~~~~~

Sometimes we want to compare the effect of different cutoff values on the ngram
counts. We can do this specifying an `unk_cutoff` argument that overrides
the vocabulary's cutoff value.

    >>> cutoff_1 = count_ngrams(2, vocab, unk_cutoff=1)
    >>> cutoff_1.vocabulary.cutoff
    1

Note that this does not affect the original vocabulary cutoff value.

    >>> vocab.cutoff
    2

Changing how ngrams are generated
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

NgramCounter uses nltk's `util.ngrams` function to break a text up into ngrams.
The behavior of `util.ngrams` is controlled by passing it keyword arguments stored
in an attribute of NgramCounter.

    >>> default_kwargs = {'left_pad_symbol': '<s>',
    ... 'pad_left': True,
    ... 'pad_right': True,
    ... 'right_pad_symbol': '</s>'}
    >>> bigram_counts.ngrams_kwargs == default_kwargs
    True

The default values of these arguments have been chosen based on what I think are
the most common practices for ngram splitting. You can easily override the settings
by passing corresponding key=value pairs to `count_ngrams`

    >>> no_padding = count_ngrams(2, vocab, sents[3:6], pad_left=False, pad_right=False)

You can always test your ngram generation by calling the `to_ngrams` method
with an example sentence.

    >>> list(no_padding.to_ngrams(sents[4]))  # doctest: +NORMALIZE_WHITESPACE
```

```
    [('Once', 'more'), ('more', 'he'), ('he', 'yawned'), ('yawned', ','),
    (',', 'and'), ('and', 'slowly'), ('slowly', 'got'), ('got', 'to'),
    ('to', 'his'), ('his', 'feet'), ('feet', 'and'), ('and', 'shook'),
    ('shook', 'himself'), ('himself', '.')]

Compare this to the output of the default counter.

    >>> list(bigram_counts.to_ngrams(sents[4]))  # doctest: +NORMALIZE_WHITESPACE
    [('<s>', 'Once'), ('Once', 'more'), ('more', 'he'), ('he', 'yawned'), ('yawned', ','),
    (',', 'and'), ('and', 'slowly'), ('slowly', 'got'), ('got', 'to'),
    ('to', 'his'), ('his', 'feet'), ('feet', 'and'), ('and', 'shook'),
    ('shook', 'himself'), ('himself', '.'), ('.', '</s>')]


Multiple (or no) sources
~~~~~~~~~~~~~~~~~~~~~~~~

Just like `build_vocab`, `count_ngrams` can handle multiple source text arguments.

    >>> multiple_texts_counter = count_ngrams(2, vocab, sents[3:6], sents[6:9])

Moreover, it is also possible to specify no texts whatsoever, which simply creates
an empty NgramCounter object so that it can be trained later.

    >>> no_initial_texts = count_ngrams(2, vocab)

In this case though, it's probably more reader-friendly to use the NgramCounter
class directly:

    >>> from nltk.model.counter import NgramCounter
    >>> no_initial_texts = NgramCounter(2, vocab)

If you choose to do this, you can call the `train_counts` method and pass it some text
to populate your counter instance with some numbers.

    >>> no_initial_texts.train_counts(sents[3:7])
    >>> expected_unigram_counts = {'.': 4,
    ... '</s>': 4,
    ... '<s>': 4,
    ... '<UNK>': 80,
    ... 'and': 5,
    ... 'he': 6,
    ... 'his': 5,
    ... 'the': 6,
    ... 'to': 4,
    ... 'yawned': 3}
    >>> no_initial_texts.unigrams == expected_unigram_counts
    True


====================
From Counts to Scores
====================

Example: MLE
------------

Once we counted up the ngrams it is trivial to turn them into a proper model
with scores (probabilities): just pass your NgramCounter object to an ngram model class.
The module currently has classes for only the basic ngram model types:
MLE, Lidstone, and Laplace.
Here's an example of how to use an MLE estimator with the counts we've defined.

    >>> from nltk.model import MLENgramModel
    >>> bigram_model = MLENgramModel(bigram_counts)

We can access the `bigram_counts` object from the `ngram_counter` attribute:

    >>> bigram_model.ngram_counter == bigram_counts
    True

The most important method of all ngram model classes is `score`, which computes
the score (aka probability) of a word given some preceding context.
In our example we are working with a bigram model, so the context consists of only
```

```
one word.

    >>> ex_score = bigram_model.score("yawned", ["he"])

Note that an MLE score is no more than the relative frequency of the word given
its context (it just looks much nicer!).

    >>> bigram_counts.ngrams[2][('he',)].freq('yawned') == ex_score
    True

Also note that both lists and tuples are supported as `context` arguments.

    >>> bigram_model.score("yawned", ("he",)) == ex_score
    True

In addition to `score` all ngram model classes provide the following methods:
- logscore
- entropy
- perplexity
Their usage is described in the following section.


Custom NgramModel Classes
-------------------------

In case you need an ngram model estimator that's not currently in the module,
it's quite easy to define your own class for that, you simply subclass the
`BaseNgramModel` class. The following section describes.

    >>> from nltk.model import BaseNgramModel
    >>> base_model = BaseNgramModel(bigram_counts)

Since it is intended only as a base class for real estimators, the `BaseNgramModel`
doesn't have an interesting `score` method, it always returns 0.5.

    >>> base_model.score("test", ('abc',))
    0.5
    >>> base_model.score("abc", ("test",))
    0.5

Children classes are expected to override this method and provide their own
logic for how to compute a score.
The advantage of having a base ngram model class is that there are plenty of methods
that do not depend of `score` in their implementations which are all defined in
the `BaseNgramModel` class and which you get "for free" by inheriting from it.
Here are examples of these methods:

    >>> base_model.logscore("abc", ("test",))
    -1.0
    >>> base_model.entropy(sents[4])
    1.0
    >>> base_model.perplexity(sents[4])
    2.0
```