Package: kgrams (via r-universe)

July 10, 2024

Title Classical k-gram Language Models

Version 0.2.0

Description Training and evaluating k-gram language models in R, supporting several probability smoothing techniques, perplexity computations, random text generation and more.

License GPL (>= 3)

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE, roclets = c (` `namespace", ` `rd"))

RoxygenNote 7.2.3

LinkingTo Rcpp, RcppProgress

Imports Rcpp, rlang, methods, utils, RcppProgress (>= 0.1), Rdpack

Depends R (>= 4.0)

Suggests testthat (>= 3.0.0), covr, knitr, rmarkdown

Config/testthat/edition 3

RdMacros Rdpack

VignetteBuilder knitr

URL https://vgherard.github.io/kgrams/, https://github.com/vgherard/kgrams

BugReports https://github.com/vgherard/kgrams/issues

Repository https://vgherard.r-universe.dev

RemoteUrl https://github.com/vgherard/kgrams

RemoteRef v0.2.0

RemoteSha 7d7e59dc6005bf01400349952d39e1d12de95158

Contents

`dictionary` *Word dictionaries*

Description

Construct or coerce to and from a dictionary.

Usage

```
dictionary(object, ...)

## S3 method for class 'kgram_freqs'
dictionary(object, size = NULL, cov = NULL, thresh = NULL, ...)

## S3 method for class 'character'
dictionary(
  object,
  .preprocess = identity,
  size = NULL,
  cov = NULL,
  thresh = NULL,
  ...
)

## S3 method for class 'connection'
dictionary(
  object,
```

```
  .preprocess = identity,
  size = NULL,
  cov = NULL,
  thresh = NULL,
  max_lines = Inf,
  batch_size = max_lines,
  ...
)

as_dictionary(object)

## S3 method for class 'kgrams_dictionary'
as_dictionary(object)

## S3 method for class 'character'
as_dictionary(object)

## S3 method for class 'kgrams_dictionary'
as.character(x, ...)
```

Arguments

`object`	object from which to extract a dictionary, or to be coerced to dictionary.
`...`	further arguments passed to or from other methods.
`size`	either `NULL` or a length one positive integer. Predefined size of the required dictionary (the top `size` most frequent words are retained).
`cov`	either `NULL` or a length one numeric between `0` and `1`. Predefined text coverage fraction of the dictionary (the most frequent words providing the required coverage are retained).
`thresh`	either `NULL` or length one a positive integer. Minimum word count threshold to include a word in the dictionary.
`.preprocess`	a function taking a character vector as input and returning a character vector as output. Optional preprocessing transformation applied to text before creating the dictionary.
`max_lines`	a length one positive integer or `Inf`. Maximum number of lines to be read from the `connection`. If `Inf`, keeps reading until the End-Of-File.
`batch_size`	a length one positive integer less than or equal to `max_lines`.Size of text batches when reading text from `connection`.
`x`	a `dictionary`.

Details

These generic functions are used to build `dictionary` objects, or to coerce from other formats to `dictionary`, and from a `dictionary` to a character vector. By now, the only non-trivial type coercible to `dictionary` is `character`, in which case each entry of the input vector is considered as a single word. Coercion from `dictionary` to `character` returns the list of words included in the dictionary as a regular character vector.

Dictionaries can be extracted from kgram_freqs objects, or *built* from text coming either directly from a character vector or a connection.

A single preprocessing transformation can be applied before processing the text for unique words. After preprocessing, *anything delimited by one or more white space characters* in the transformed text input *is counted as a word* and may be added to the dictionary modulo additional constraints.

The possible constraints for including a word in the dictionary can be of three types: (i) fixed size of dictionary, implemented by the size argument; (ii) fixed text covering fraction, as specified by the cov argument; or (iii) minimum word count threshold, thresh argument. *Only one of these constraints can be applied at a time*, so that specifying more than one of size, cov or thresh results in an error.

Value

A dictionary for dictionary() and as_dictionary(), a character vector for the as.character() method.

Author(s)

Valerio Gherardi

Examples

```
# Building a dictionary from Shakespeare's "Much Ado About Nothing"

dict <- dictionary(much_ado)
length(dict)
query(dict, "leonato") # TRUE
query(dict, c("thy", "thou")) # c(TRUE, TRUE)
query(dict, "smartphones") # FALSE

# Getting list of words as regular character vector
words <- as.character(dict)
head(words)

# Building a dictionary from a list of words
dict <- as_dictionary(c("i", "the", "a"))
```

EOS *Special Tokens*

Description

Return Begin-Of-Sentence, End-Of-Sentence and Unknown-Word special tokens.

Usage

```
EOS()

BOS()

UNK()
```

Details

These functions return the internal representation of BOS, EOS and UNK tokens respectively. Their actual returned values are irrelevant and their only purpose is to simplify queries of k-gram counts and probabilities involving the special tokens, as shown in the examples.

Value

a string representing the appropriate special token.

Author(s)

Valerio Gherardi

Examples

```
f <- kgram_freqs("a b b a b", 2)
query(f, c(BOS(), EOS(), UNK()))

m <- language_model(f, "add_k", k = 1)
probability(c("a", "b") %|% BOS(), m)
probability("a b b a" %+% EOS(), m)

# The actual values of BOS(), EOS() and UNK() are irrelevant
c(BOS(), EOS(), UNK())
```

---

kgram_freqs *k-gram Frequency Tables*

Description

Extract k-gram frequency counts from a text or a connection.

Principal methods supported by objects of class `kgram_freqs`:

- `query()`: query k-gram counts from the table. See query
- `probability()`: compute word continuation and sentence probabilities using Maximum Likelihood estimates. See probability.
- `language_model()`: build a k-gram language model using various probability smoothing techniques. See language_model.

Usage

```
kgram_freqs(object, ...)

## S3 method for class 'numeric'
kgram_freqs(
  object,
  .preprocess = identity,
  .tknz_sent = identity,
  dict = NULL,
  ...
)

## S3 method for class 'kgram_freqs'
kgram_freqs(object, ...)

## S3 method for class 'character'
kgram_freqs(
  object,
  N,
  .preprocess = identity,
  .tknz_sent = identity,
  dict = NULL,
  open_dict = is.null(dict),
  verbose = FALSE,
  ...
)

## S3 method for class 'connection'
kgram_freqs(
  object,
  N,
  .preprocess = identity,
  .tknz_sent = identity,
  dict = NULL,
  open_dict = is.null(dict),
  verbose = FALSE,
  max_lines = Inf,
  batch_size = max_lines,
  ...
)

process_sentences(
  text,
  freqs,
  .preprocess = attr(freqs, ".preprocess"),
  .tknz_sent = attr(freqs, ".tknz_sent"),
  open_dict = TRUE,
  in_place = TRUE,
```

```
  verbose = FALSE,
  ...
)

## S3 method for class 'character'
process_sentences(
  text,
  freqs,
  .preprocess = attr(freqs, ".preprocess"),
  .tknz_sent = attr(freqs, ".tknz_sent"),
  open_dict = TRUE,
  in_place = TRUE,
  verbose = FALSE,
  ...
)

## S3 method for class 'connection'
process_sentences(
  text,
  freqs,
  .preprocess = attr(freqs, ".preprocess"),
  .tknz_sent = attr(freqs, ".tknz_sent"),
  open_dict = TRUE,
  in_place = TRUE,
  verbose = FALSE,
  max_lines = Inf,
  batch_size = max_lines,
  ...
)
```

Arguments

`object`	any type allowed by the available methods. The type defines the behaviour of `kgram_freqs()` as a default constructor, a copy constructor or a constructor of a non-trivial object. See 'Details'.
`...`	further arguments passed to or from other methods.
`.preprocess`	a function taking a character vector as input and returning a character vector as output. Optional preprocessing transformation applied to text before k-gram tokenization. See 'Details'.
`.tknz_sent`	a function taking a character vector as input and returning a character vector as output. Optional sentence tokenization step applied to text after preprocessing and before k-gram tokenization. See 'Details'.
`dict`	anything coercible to class dictionary. Optional pre-specified word dictionary.
`N`	a length one integer. Maximum order of k-grams to be considered.
`open_dict`	`TRUE` or `FALSE`. If `TRUE`, any new word encountered during processing not appearing in the original dictionary is included into the dictionary. Otherwise, new words are replaced by an unknown word token. It is by default `TRUE` if `dict` is specified, `FALSE` otherwise.

verbose	Print current progress to the console.
max_lines	a length one positive integer or Inf. Maximum number of lines to be read from the connection. If Inf, keeps reading until the End-Of-File.
batch_size	a length one positive integer less than or equal to max_lines.Size of text batches when reading text from connection.
text	a character vector or a connection. Source of text from which k-gram frequencies are to be extracted.
freqs	a kgram_freqs object, to which new k-gram counts from text are to be added.
in_place	TRUE or FALSE. Should the initial kgram_freqs object be modified in place?

Details

The function kgram_freqs() is a generic constructor for objects of class kgram_freqs, i.e. k-gram frequency tables. The constructor from integer returns an empty 'kgram_freqs' of fixed order, with an optional predefined dictionary (which can be empty) and .preprocess and .tknz_sent functions to be used as defaults in other kgram_freqs methods. The constructor from kgram_freqs returns a copy of an existing object, and it is provided because, in general, kgram_freqs objects have reference semantics, as discussed below.

The following discussion focuses on process_sentences() generic, as well as on the character and connection methods of the constructor kgram_freqs(). These functions extract k-gram frequency counts from a text source, which may be either a character vector or a connection. The second option is useful if one wants to avoid loading the full text corpus in physical memory, allowing to process text from different sources such as files, compressed files or URLs.

The returned object is of class kgram_freqs (a thin wrapper around the internal C++ class where all k-gram computations take place). kgram_freqs objects have methods for querying bare k-gram frequencies (query) and maximum likelihood estimates of sentence probabilities or word continuation probabilities (see probability)) . More importantly kgram_freqs objects are used to create language_model objects, which support various probability smoothing techniques.

The function kgram_freqs() is used to *construct* a new kgram_freqs object, initializing it with the k-gram counts from the text input, whereas process_sentences() is used to add k-gram counts from a new text to an *existing* kgram_freqs object, freqs. In this second case, the initial object freqs can either be modified in place (for in_place == TRUE, the default) or by making a copy (in_place == FALSE), see the examples below. The final object is returned invisibly when modifying in place, visibly in the second case. It is worth to mention that modifying in place a kgram_freqs object freqs will also affect language_model objects created from freqs with language_model(), which will also be updated with the new information. If one wants to avoid this behaviour, one can make copies using either the kgram_freqs() copy constructor, or the in_place = FALSE argument.

The dict argument allows to provide an initial set of known words. Subsequently, one can either work with such a closed dictionary (open_dict == FALSE), or extended the dictionary with all new words encountered during k-gram processing (open_dict == TRUE) .

The .preprocess and .tknz_sent functions are applied *before* k-gram counting takes place, and are in principle arbitrary transformations of the original text. *After* preprocessing and sentence tokenization, each line of the transformed input is presented to the k-gram counting algorithm as a separate sentence (these sentences are implicitly padded with N - 1 Begin-Of-Sentence (BOS) and one End-Of-Sentence (EOS) tokens, respectively. This is illustrated in the examples). For basic

usage, this package offers the utilities preprocess and tknz_sent. Notice that, strictly speaking, there is some redundancy in these two arguments, as the processed input to the k-gram counting algorithm is `.tknz_sent(.preprocess(text))`. They appear explicitly as separate arguments for two main reasons:

- The presence of `.tknz_sent` is a reminder of the fact that sentences have to be explicitly separeted in different entries of the processed input, in order for `kgram_freqs()` to append the correct Begin-Of-Sentence and End-Of-Sentence paddings to each sentence.
- At prediction time (e.g. with probability), by default only `.preprocess` is applied when computing conditional probabilities, whereas both `.preprocess()` and `.tknz_sent()` are applied when computing sentence absolute probabilities.

Value

A `kgram_freqs` class object: k-gram frequency table storing k-gram counts from text. For `process_sentences()`, the updated `kgram_freqs` object is returned invisibly if `in_place` is `TRUE`, visibly otherwise.

Author(s)

Valerio Gherardi

See Also

query, probability language_model, dictionary

Examples

```
# Build a k-gram frequency table from a character vector

f <- kgram_freqs("a b b a a", 3)
f
summary(f)
query(f, c("a", "b")) # c(3, 2)
query(f, c("a b", "a" %+% EOS(), BOS() %+% "a b")) # c(1, 1, 1)
query(f, "a b b a") # NA (counts for k-grams of order k > 3 are not known)

process_sentences("b", f)
query(f, c("a", "b")) # c(3, 3): 'f' is updated in place

f1 <- process_sentences("b", f, in_place = FALSE)
query(f, c("a", "b")) # c(3, 3): 'f' is copied
query(f1, c("a", "b")) # c(3, 4): the new 'f1' stores the updated counts




# Build a k-gram frequency table from a file connection

## Not run:
f <- kgram_freqs(file("my_text_file.txt"), 3)
```

```
## End(Not run)


# Build a k-gram frequency table from an URL connection
## Not run:
f <- kgram_freqs(url("http://my.website/my_text_file.txt"), 3)

## End(Not run)
```

`language_model` *k-gram Language Models*

Description

Build a k-gram language model.

Principal methods supported by objects of class `language_model`:

- `probability()`: compute word continuation and sentence probabilities. See probability.
- `sample_sentences()`: generate random text by sampling from the language model probability distribution at arbitary temperature. See sample_sentences.
- `perplexity()`: Compute the language model perplexity on a test corpus. See perplexity.

Usage

```
language_model(object, ...)

## S3 method for class 'language_model'
language_model(object, ...)

## S3 method for class 'kgram_freqs'
language_model(object, smoother = "ml", N = param(object, "N"), ...)
```

Arguments

`object`	an object which stores the information required to build the k-gram model. At present, necessarily a `kgram_freqs` object, or a `language_model` object of which a copy is desired (see Details).
`...`	possible additional parameters required by the smoother.
`smoother`	a length one character vector. Indicates the smoothing technique to be applied to compute k-gram continuation probabilities. A list of available smoothers can be obtained with `smoothers()`, and further information on a particular smoother through `info()`.
`N`	a length one integer. Maximum order of k-grams to use in the language model. This muss be less than or equal to the order of the underlying `kgram_freqs` object.

Details

These generics are used to construct objects of class `language_model`. The `language_model` method is only needed to create copies of `language_model` objects (that is to say, new copies which are not altered by methods which modify the original object in place, see e.g. parameters). The discussion below focuses on language models and the `kgram_freqs` method.

kgrams supports several k-gram language models, including Interpolated Kneser-Ney, Stupid Backoff and others (see smoothers). The objects created by `language_models()` have methods for computing word continuation and sentence probabilities (see probability), random text generation (see sample_sentences) and other type of language modeling tasks such as computing perplexities and word prediction accuracies.

Smoothers have often tuning parameters, which need to be specified by (exact) name through the `...` arguments; otherwise, `language_model()` will use default values and, once per session, throw a warning. `info(smoother)` lists all parameters needed by a specific smoother, together with their allowed parameter space.

The run-time of `language_model()` may vary substantially for different smoothing methods, depending on whether or not a method requires the computation of additional quantities (that is to say, beyond k-gram counts) for its operativity (this is, for instance, the case for the Kneser-Ney smoother).

Value

A `language_model` object.

Author(s)

Valerio Gherardi

Examples

```
# Create an interpolated Kneser-Ney 2-gram language model

freqs <- kgram_freqs("a a b a a b a b a b a b", 2)
model <- language_model(freqs, "kn", D = 0.5)
model
summary(model)
probability("a" %|% "b", model)

# For more examples, see ?probability, ?sample_sentences and ?perplexity.
```

`midsummer` *A Midsummer Night's Dream*

Description

The entire play "A Midsummer Night's Dream" from William Shakespeare.

Usage

```
midsummer
```

Format

A length one character vector, containing the entire text of "A Midsummer Night's Dream" from William Shakespeare. The script used for generating this file is available here

Source

https://www.folger.edu/

See Also

much_ado

Examples

```
midsummer[840]
```

---

much_ado *Much Ado About Nothing*

Description

The entire play "Much Ado About Nothing" from William Shakespeare.

Usage

```
much_ado
```

Format

A length one character vector, containing the entire text of "Much Ado About Nothing" from William Shakespeare. The script used for generating this file is available here

Source

https://www.folger.edu/

See Also

midsummer

Examples

```
much_ado[840]
```

parameters *Language Model Parameters*

Description

Get and set parameters of a language model.

Usage

```
param(object, which)

## S3 method for class 'kgram_freqs'
param(object, which)

param(object, which) <- value

parameters(object)
```

Arguments

object	a language_model or kgram_freqs class object.
which	a string. Name of the parameter to get or set.
value	new value for the parameter specified by which. Typically a length one numeric.

Details

These functions are used to retrieve or modify the parameters of a language_model or a kgram_freqs object. Any object of, or inheriting from, any of these two classes has at least two parameters:

- N: higher order of k-grams considered in the model for language_model, or stored in memory for kgram_freqs.
- V: size of the dictionary (excluding the special tokens BOS(), EOS(), UNK()).

For an object of class kgram_freqs, these are the only parameters, and they are read-only. language_models allow to set N less than or equal to the order of the underlying kgram_freqs object.

In addition to these, language_models can have additional parameters, e.g. discount values or interpolation constants, depending on the particular smoother employed by the model. A list of parameters available for a given smoother can be obtained through info() (see smoothers).

It is important to mention that setting a parameter is an in-place operation. This implies that if, say, object m is a language_model object, the code m1 <- m ; param(m1, which) <- value will set the parameter which to value both for m1 *and* m. The reason for this is that, behind the scenes, both m and m1 are pointers to the same C++ object. In order to create a true copy, one can use the copy constructor language_model(), see language_model.

Value

a list for parameters(), a single value, typically numeric, for param() and param()<- (the new value, in this last case).

Examples

```
# Get and set k-gram model parameters

f <- kgram_freqs("a a b a b", 3)
param(f, "N")
parameters(f)

m <- language_model(f, "sbo", lambda = 0.2)
param(m, "V")
param(m, "lambda")
param(m, "N") <- 2
param(m, "lambda") <- 0.4

if (FALSE) {
        param(m, "V") <- 5 # Error: dictionary size cannot be set.
}

if (FALSE) {
        param(f, "N") <- 4 # Error: parameters of 'kgram_freqs' cannot be set
}

m1 <- m
param(m1, "lambda") <- 0.5
param(m, "lambda") # 0.5 ; param() modifies 'm' by reference!

m2 <- language_model(m) # This creates a true copy
param(m2, "lambda") <- 0.6
param(m, "lambda") # 0.5
```

---

perplexity *Language Model Perplexities*

Description

Compute language model perplexities on a test corpus.

Usage

```
perplexity(
  text,
  model,
  .preprocess = attr(model, ".preprocess"),
  .tknz_sent = attr(model, ".tknz_sent"),
  exp = TRUE,
  ...
)

## S3 method for class 'character'
```

```
perplexity(
  text,
  model,
  .preprocess = attr(model, ".preprocess"),
  .tknz_sent = attr(model, ".tknz_sent"),
  exp = TRUE,
  detailed = FALSE,
  ...
)

## S3 method for class 'connection'
perplexity(
  text,
  model,
  .preprocess = attr(model, ".preprocess"),
  .tknz_sent = attr(model, ".tknz_sent"),
  exp = TRUE,
  batch_size = Inf,
  ...
)
```

Arguments

`text`	a character vector or connection. Test corpus from which language model perplexity is computed.
`model`	an object of class `language_model`.
`.preprocess`	a function taking a character vector as input and returning a character vector as output. Preprocessing transformation applied to input before computing perplexity.
`.tknz_sent`	a function taking a character vector as input and returning a character vector as output. Optional sentence tokenization step applied before computing perplexity.
`exp`	`TRUE` or `FALSE`. If `TRUE`, returns the actual perplexity - exponential of cross-entropy per token - otherwise returns its natural logarithm.
`...`	further arguments passed to or from other methods.
`detailed`	`TRUE` or `FALSE`. If `TRUE`, the output has a `"details"` attribute, which is a data-frame containing the cross-entropy of each individual sentence tokenized from `text`.
`batch_size`	a length one positive integer or `Inf`. Size of text batches when reading text from a `connection`. If `Inf`, all input text is processed in a single batch.

Details

These generic functions are used to compute a `language_model` perplexity on a test corpus, which may be either a plain character vector of text, or a connection from which text can be read in batches. The second option is useful if one wants to avoid loading the full text in physical memory, and allows to process text from different sources such as files, compressed files or URLs.

"Perplexity" is defined here, following Ref. (Chen and Goodman 1999), as the exponential of the normalized language model cross-entropy with the test corpus. Cross-entropy is normalized by the total number of words in the corpus, where we include the End-Of-Sentence tokens, but not the Begin-Of-Sentence tokens, in the word count.

The custom .preprocess and .tknz_sent arguments allow to apply transformations to the text corpus before the perplexity computation takes place. By default, the same functions used during model building are employed, c.f. kgram_freqs and language_model.

A note of caution is in order. Perplexity is not defined for all language models available in kgrams. For instance, smoother `"sbo"` (i.e. Stupid Backoff (Brants et al. 2007)) does not produce normalized probabilities, and this is signaled by a warning (shown once per session) if the user attempts to compute the perplexity for such a model. In these cases, when possible, perplexity computations are performed anyway case, as the results might still be useful (e.g. to tune the model's parameters), even if their probabilistic interpretation does no longer hold.

Value

a number. Perplexity of the language model on the test corpus.

Author(s)

Valerio Gherardi

References

Brants T, Popat AC, Xu P, Och FJ, Dean J (2007). "Large Language Models in Machine Translation." In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 858–867. https://aclanthology.org/D07-1090/.

Chen SF, Goodman J (1999). "An empirical study of smoothing techniques for language modeling." *Computer Speech & Language*, **13**(4), 359–394.

Examples

```
# Train 4-, 6-, and 8-gram models on Shakespeare's "Much Ado About Nothing",
# compute their perplexities on the training and test corpora.
# We use Shakespeare's "A Midsummer Night's Dream" as test.


train <- much_ado
test <- midsummer

tknz <- function(text) tknz_sent(text, keep_first = TRUE)
f <- kgram_freqs(train, 8, .tknz_sent = tknz)
m <- language_model(f, "kn", D = 0.75)

# Compute perplexities for 4-, 6-, and 8-gram models
FUN <- function(N) {
        param(m, "N") <- N
        c(train = perplexity(train, m), test = perplexity(test, m))
```

```
        }
sapply(c("N = 4" = 4, "N = 6" = 6, "N = 8" = 8), FUN)
```

`preprocess` *Text preprocessing*

Description

A minimal text preprocessing utility.

Usage

```
preprocess(input, erase = "[^.?!:;'[:alnum:][:space:]]", lower_case = TRUE)
```

Arguments

`input`	a character vector.
`erase`	a length one character vector. Regular expression matching parts of text to be *erased* from input. The default removes anything not alphanumeric (`[A-z0-9]`), space (white space, tab, vertical tab, newline, form feed, carriage return), apostrophes or punctuation characters (`"[.?!:;]"`).
`lower_case`	a length one logical vector. If TRUE, puts everything to lower case.

Details

The expressions `preprocess(x, erase = pattern, lower_case = TRUE)` and `preprocess(x, erase = pattern, lower_case = FALSE)` are roughly equivalent to `tolower(gsub(pattern, "", x))` and `gsub(pattern, "", x)`, respectively, provided that the regular expression 'pattern' is correctly recognized by R.

Note. This function, as well as tknz_sent, are included in the library for illustrative purposes only, and are not optimized for performance. Furthermore (for performance reasons) the function has a separate implementation for Windows and UNIX OS types, respectively, so that results obtained in the two cases may differ slightly. In contexts that require full reproducibility, users are encouraged to define their own preprocessing and tokenization custom functions - or to work with externally processed data.

Value

a character vector containing the processed output.

Author(s)

Valerio Gherardi

Examples

```
preprocess("#This Is An Example@-@!#")
```

`probability` *Language Model Probabilities*

Description

Compute sentence probabilities and word continuation conditional probabilities from a language model

Usage

```
probability(object, model, .preprocess = attr(model, ".preprocess"), ...)

## S3 method for class 'kgrams_word_context'
probability(object, model, .preprocess = attr(model, ".preprocess"), ...)

## S3 method for class 'character'
probability(
  object,
  model,
  .preprocess = attr(model, ".preprocess"),
  .tknz_sent = attr(model, ".tknz_sent"),
  ...
)
```

Arguments

`object`	a character vector for sentence probabilities, a word-context conditional expression created with the conditional operator `%|%` (see word_context). for word continuation probabilities.
`model`	an object of class `language_model`.
`.preprocess`	a function taking a character vector as input and returning a character vector as output. Preprocessing transformation applied to input before computing probabilities
`...`	further arguments passed to or from other methods.
`.tknz_sent`	a function taking a character vector as input and returning a character vector as output. Optional sentence tokenization step applied before computing sentence probabilities.

Details

The generic function `probability()` is used to obtain both sentence unconditional probabilities (such as Prob("I was starting to feel drunk")) and word continuation conditional probabilities (such as Prob("you" | "i love")). In plain words, these probabilities answer the following related but conceptually different questions:

- Sentence probability Prob(s): what is the probability that extracting a single sentence (from a corpus of text, say) we will obtain exactly 's'?

- Continuation probability Prob(w|c): what is the probability that a given context 'c' will be followed exactly by the word 'w'?

In order to compute continuation probabilities (i.e. Prob(w|c)), one must create conditional expressions with the infix operator `%|%`, as shown in the examples below. Both `probability` and `%|%` are vectorized with respect to words (left hand side of `%|%`), but the context must be a length one character (right hand side of `%|%`).

The input is treated as in query for what concerns word tokenization: anything delimited by (one or more) white space(s) is tokenized as a word. For sentence probabilities, Begin-Of-Sentence and End-Of-Sentence paddings are implicitly added to the input, but specifying them explicitly does not produce wrong results as BOS and EOS tokens are ignored by `probability()` (see the examples below). For continuation probabilities, any context of more than `N - 1` words (where `N` is the k-gram order the language model) is truncated to the last `N - 1` words.

By default, the same `.preprocess()` and `.tknz_sent()` functions used during model building are applied to the input, but this can be overriden with arbitrary functions. Notice that the `.tknz_sent` can be useful (for sentence probabilities) if e.g. the input is a length one unprocessed character vector.

Value

a numeric vector. Probabilities of the sentences or word continuations.

Author(s)

Valerio Gherardi

Examples

```
# Usage of probability()

f <- kgram_freqs("a b b a b a b", 2)

m <- language_model(f, "add_k", k = 1)
probability(c("a", "b", EOS(), UNK()) %|% BOS(), m) # c(0.4, 0.2, 0.2, 0.2)
probability("a" %|% UNK(), m) # not NA
```

---

query *Query k-gram frequency tables or dictionaries*

Description

Return the frequency count of k-grams in a k-gram frequency table, or whether words are contained in a dictionary.

Usage

```
query(object, x)

## S3 method for class 'kgram_freqs'
query(object, x)

## S3 method for class 'kgrams_dictionary'
query(object, x)
```

Arguments

object	a kgram_freqs or dictionary class object.
x	a character vector. A list of k-grams if object is of class kgram_freqs, a list of words if object is a dictionary.

Details

This generic has slightly different behaviors when querying for the presence of words in a dictionary and for k-gram counts in a frequency table respectively. For words, query() looks for exact matches between the input and the dictionary entries. Queries of Begin-Of-Sentence (BOS()) and End-Of-Sentence (EOS()) tokens always return TRUE, and queries of the Unknown-Word token return FALSE (see special_tokens).

On the other hand, queries of k-gram counts first perform a word level tokenization, so that anything separated by one or more space characters in the input is considered as a single word (thus, for instance queries of strings such as "i love you", " i love you"), or "i love you ") all produce the same outcome). Moreover, querying for any word outside the underlying dictionary returns the counts corresponding to the Unknown-Word token (UNK()) (e.g., if the word "prcsrn" is outside the dictionary, querying "i love prcsrn" is the same as querying paste("i love", UNK())). Queries from k-grams of order k > N will return NA.

A subsetting equivalent of query, with synthax object[x] is available (see the examples). query(object, x). The query of the empty string "" returns the total count of words, including the EOS and UNK tokens, but not the BOS token.

See also the examples below.

Value

an integer vector, containing k-gram counts of x, if object is a kgram_freqs class object, a logical vector if object is a dictionary. Vectorized over x.

Author(s)

Valerio Gherardi

Examples

```
# Querying a k-gram frequency table
f <- kgram_freqs("a a b a b b a b", N = 2)
query(f, c("a", "b")) # query single words
```

```
query(f, c("a b")) # query a 2-gram
identical(query(f, "c"), query(f, "d"))  # TRUE, both "c" and "d" are <UNK>
identical(query(f, UNK()), query(f, "c")) # TRUE
query(f, EOS()) # 1, since text is a single sentence
f[c("b b", "b")] # query with subsetting synthax
f[""] # 9 (includes the EOS token)

# Querying a dictionary
d <- as_dictionary(c("a", "b"))
query(d, c("a", "b", "c")) # query some words
query(d, c(BOS(), EOS(), UNK())) # c(TRUE, TRUE, FALSE)
d["a"] # query with subsetting synthax
```

`sample_sentences` *Random Text Generation*

Description

Sample sentences from a language model's probability distribution.

Usage

```
sample_sentences(model, n, max_length, t = 1)
```

Arguments

`model`	an object of class `language_model`.
`n`	an integer. Number of sentences to sample.
`max_length`	an integer. Maximum length of sampled sentences.
`t`	a positive number. Sampling temperature (optional); see Details.

Details

This function samples sentences according the prescribed language model's probability distribution, with an optional temperature parameter. The temperature transform of a probability distribution is defined by `p(t) = exp(log(p) / t) / Z(t)` where `Z(t)` is the partition function, fixed by the normalization condition `sum(p(t)) = 1`.

Sampling is performed word by word, using the already sampled string as context, starting from the Begin-Of-Sentence context (i.e. `N - 1` BOS tokens). Sampling stops either when an End-Of-Sentence token is encountered, or when the string exceeds `max_length`, in which case a truncated output is returned.

Some language models may give a non-zero probability to the the Unknown word token, but this is never produced in text generated by `sample_sentences()`: when randomly sampled, it is simply ignored.

Finally, a word of caution on some special smoothers: `"sbo"` smoother (Stupid Backoff), does not produce normalized continuation probabilities, but rather continuation *scores*. Sampling is here

performed by assuming that Stupid Backoff scores are *proportional* to actual probabilities. 'ml' smoother (Maximum Likelihood) does not assign probabilities when the k-gram count of the context is zero. When this happens, the next word is chosen uniformly at random from the model's dictionary.

Value

a character vector of length `n`. Random sentences generated from the language model's distribution.

Author(s)

Valerio Gherardi

Examples

```
# Sample sentences from 8-gram Kneser-Ney model trained on Shakespeare's
# "Much Ado About Nothing"



### Prepare the model and set seed
freqs <- kgram_freqs(much_ado, 8, .tknz_sent = tknz_sent)
model <- language_model(freqs, "kn", D = 0.75)
set.seed(840)

sample_sentences(model, n = 3, max_length = 10)

### Sampling at high temperature
sample_sentences(model, n = 3, max_length = 10, t = 100)

### Sampling at low temperature
sample_sentences(model, n = 3, max_length = 10, t = 0.01)
```

`smoothers` *k-gram Probability Smoothers*

Description

Information on available k-gram continuation probability smoothers.

List of smoothers currently supported by `kgrams`:

- `"ml"`: Maximum Likelihood estimate (Markov 1913).
- `"add_k"`: Add-k smoothing (Dale and Laplace 1995; Lidstone 1920; Johnson 1932; Jeffreys 1998).
- `"abs"`: Absolute discounting (Ney and Essen 1991).
- `"wb"`: Witten-Bell smoothing (Bell et al. 1990; Witten and Bell 1991)

- "kn": Interpolated Kneser-Ney. (Kneser and Ney 1995; Chen and Goodman 1999).
- "mkn": Interpolated modified Kneser-Ney. (Chen and Goodman 1999).
- "sbo": Stupid Backoff (Brants et al. 2007).

Usage

```
smoothers()

info(smoother)
```

Arguments

smoother	a string. Code name of probability smoother.

Value

`smoothers()` returns a character vector, the list of code names of probability smoothers available in kgrams. `info(smoother)` returns `NULL` (invisibly) and prints some information on the selected smoothing technique.

Author(s)

Valerio Gherardi

References

Bell TC, Cleary JG, Witten IH (1990). *Text compression*. Prentice-Hall, Inc.

Brants T, Popat AC, Xu P, Och FJ, Dean J (2007). "Large Language Models in Machine Translation." In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 858–867. https://aclanthology.org/D07-1090/.

Chen SF, Goodman J (1999). "An empirical study of smoothing techniques for language modeling." *Computer Speech & Language*, **13**(4), 359–394.

Dale AI, Laplace P (1995). *Philosophical essay on probabilities*. Springer.

Jeffreys H (1998). *The theory of probability*. OUP Oxford.

Johnson WE (1932). "Probability: The deductive and inductive problems." *Mind*, **41**(164), 409–423.

Kneser R, Ney H (1995). "Improved backing-off for M-gram language modeling." *1995 International Conference on Acoustics, Speech, and Signal Processing*, **1**, 181-184 vol.1.

Lidstone GJ (1920). "Note on the general case of the Bayes-Laplace formula for inductive or a posteriori probabilities." *Transactions of the Faculty of Actuaries*, **8**(182-192), 13.

Markov AA (1913). "Essai d'une Recherche Statistique Sur le Texte du Roman Eugene Oneguine." *Bull. Acad. Imper. Sci. St. Petersburg*, **7**.

Ney H, Essen U (1991). "On smoothing techniques for bigram-based natural language modelling." In *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 825–828. IEEE Computer Society.

Witten IH, Bell TC (1991). "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression." *Ieee transactions on information theory*, **37**(4), 1085–1094.

Examples

```
# List available smoothers
smoothers()

# Get information on smoother "kn", i.e. Interpolated Kneser-Ney
info("kn")
```

`tknz_sent` *Sentence tokenizer*

Description

Extract sentences from a batch of text lines.

Usage

```
tknz_sent(input, EOS = "[.?!:;]+", keep_first = FALSE)
```

Arguments

`input`	a character vector.
`EOS`	a regular expression matching an End-Of-Sentence delimiter.
`keep_first`	TRUE or FALSE? Should the first character of the matches be appended to the returned sentences (with a space)?

Details

`tknz_sent()` splits text into sentences, where sentence delimiters are specified by a regular expression through the `EOS` argument. Specifically, when an EOS token is found, the next sentence begins at the first position in the input string not containing any of the EOS tokens *or white space* (so that entries like `"Hi there!!!"` or `"Hello . . ."` are both recognized as a single sentence).

If `keep_first` is `FALSE`, the delimiters are stripped off from the returned sequences. Otherwise, the first character of the substrings matching the `EOS` regular expressions are appended to the corresponding sentences, preceded by a white space.

In the absence of any EOS delimiter, tknz_sent() returns the input as is, since parts of text corresponding to different entries of the input vector x are understood as parts of separate sentences.

Note. This function, as well as preprocess, are included in the library for illustrative purposes only, and are not optimized for performance. Furthermore (for performance reasons) the function has a separate implementation for Windows and UNIX OS types, respectively, so that results obtained in the two cases may differ slightly. In contexts that require full reproducibility, users are encouraged to define their own preprocessing and tokenization custom functions - or to work with externally processed data.

Value

a character vector, each entry of which corresponds to a single sentence.

Author(s)

Valerio Gherardi

Examples

```
tknz_sent("Hi there! I'm using kgrams.")
```

---

word_context *Word-context conditional expression*

Description

Create word-context conditional expression with the %|% operator.

Usage

```
word %|% context
```

Arguments

word	a character vector. Word or words to include as the variable part of the conditional expression.
context	a character vector of length one. The fixed (or "given") part of the conditional expression.

Details

The intuitive meaning of the operator %|% is that of the mathematical symbol | (given). This operator is used to create conditional expressions representing the occurrence of some word after a given context (for instance, the expression "you" %|% "i love" would represent the occurrence of the word "you" after the string "i love"). The purpose of %|% is to create objects which can be given as input to probability() (see probability for further examples).

Value

a word_context class object.

Author(s)

Valerio Gherardi

Examples

```
f <- kgram_freqs(much_ado, 2, .tknz_sent = tknz_sent)
m <- language_model(f, "kn", D = 0.5)
probability("leonato" %|% "enter", m)
```

`%+%` *String concatenation*

Description

String concatenation

Usage

```
lhs %+% rhs
```

Arguments

lhs	a string or vector of strings.
rhs	a string or vector of strings.

Details

The expression `lhs %+% rhs` is equivalent to `paste(lhs, rhs, sep = " ", collapse = NULL, recycle0 = FALSE)`. See paste for more details.

Value

a string or vector of strings.

Author(s)

Valerio Gherardi

Brief synthax for string concatenation.

See Also

paste

Examples

```
"i love" %+% c("cats", "jazz", "you")
```

Index