Package ‘rcttext’

January 2, 2025

Title Tools for impact analysis in randomized trials with text outcomes

Version 0.0.0.9000

Description A flexible and user-friendly toolkit for performing impact analysis in randomized trials with outcomes generated through human, machine, and/or hybrid scoring of text data. Provides functionality for feature extraction and aggregation, applying supervised and unsupervised machine learning models for semi-automated text scoring, estimating model-assisted treatment impacts with respect to text outcomes under various randomized designs, visually representing found impacts on text outcomes, and additional functionality for performing text analysis using existing frameworks, especially quanteda.

Encoding UTF-8

Imports quanteda,
textreg,
sampling,
tm,
dplyr,
caret,
quanteda.textstats,
plotrix,
randomizr,
stringi,
Rdpack

RdMacros Rdpack

RoxygenNote 7.3.2

Depends R (>= 3.5)

LazyData true

Suggests knitr,
rmarkdown,
testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Contents

`apply_hunspell` *Force spell-correct*

Description

Try to spell correct unrecognized words as best as able using the hunspell package. This will take the first suggestion of hunspell and just replace the text with the suggestion, word by word.

Usage

```
apply_hunspell(
  text,
  additional_words = NULL,
  skip_prefix = NULL,
  threshold = 1,
  to_lower = FALSE,
  verbose = FALSE
)
```

Arguments

`text`	The text to spell check (list of character vectors)
`additional_words`	List of words that should be considered as spelled correctly.
`skip_prefix`	List of prefixes that if a word has we should skip over
`threshold`	Don't try to correct anything that only occurs threshold or less times in corpus.
`to_lower`	If TRUE will keep everything lowercase, regardless of spelling suggestions.

Details

All words of 2 or 1 character are skipped.

Value

Vector of text, spell-corrected we hope.

Examples

```
## Example 1: Single string correction

# Text to be checked for spelling errors
txt = "This function seplaces the textt withh the suggestion"

# Check for spelling errors and apply replacements
txt_rep = apply_hunspell( txt, verbose=TRUE, threshold = 0 )
txt_rep

## Example 2: Multiple texts correction

# Load example dataframe with multiple texts
data("toy_reads")

# Extract texts from the dataframe
txts = toy_reads$text
length(txts)

# Check for spelling errors and apply replacements
txts_rep = apply_hunspell( txts, verbose=TRUE, threshold = 0 )
which( txts_rep != txts )

# Display original and corrected texts
txts[7]
txts_rep[7]
```

```
txts[20]
txts_rep[20]

# Check for spelling errors and apply replacements with additional accepted words
txts_rep2 = apply_hunspell( txts,
                            verbose=TRUE,
                            additional_words = c( "dont", "rainforest" ),
                            threshold = 0 )
                            which( txts_rep2 != txts )

# Display original and corrected texts with additional accepted words
txts[7]
txts_rep2[7]

txts[20]
txts_rep2[20]
```

`best_ML` *best_ML Function*

Description

This function evaluates multiple machine learning models using a specified resampling methods and returns the best performing model based on highest out-of-fold R-squared.

Usage

```
best_ML(
  features,
  outcome,
  num = 3,
  por = 0.8,
 method = c("boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV",
    "LGOCV", "none", "oob", "adaptive_cv", "adaptive_boot", "adaptive_LGOCV"),
  mlList = c("rf", "xgbTree")
)
```

Arguments

`features`	A data frame of feature variables.
`outcome`	A vector or data frame containing the outcome variable.
`num`	Either the number of folds (for cross-validation) or the number of resampling iterations.
`por`	The proportion of data to be used for training, with the remainder used for testing (default is 0.8).
`method`	The resampling method to be used. Options include "boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV", "LGOCV", "none", "oob", "adaptive_cv", "adaptive_boot", and "adaptive_LGOCV". Default is "cv".
`mlList`	A vector of machine learning methods to be used. Some options include "rf" (random forest) and "xgbTree" (XGBoost tree). Default is c("rf", "xgbTree").

Value

A list containing:

`mean_performance`	A data frame of average performance metrics (e.g., R-squared) for each model.
`best_model`	The name of the best-performing model based on the highest R-squared value.

Examples

```
# Load necessary libraries

# Simulate example data
set.seed(123)
features <- data.frame(
  x1 = rnorm(100),
  x2 = rnorm(100),
  x3 = rnorm(100)
)
outcome <- rnorm(100)

# Apply the best_ML function
result <- best_ML(features = features, outcome = outcome, num = 3, por = 0.8,
                  method = "cv", mlList = c("rf", "xgbTree"))

# View the results
print(result)
```

`ccs_list_table` *Given a list of CCS models, produce a table of the models*

Description

This is a wrapper of the textreg make.list.table() method with some extra cleanup.

Usage

```
ccs_list_table(model_list)
```

Arguments

`model_list`	A list of CCS models, or the result of a ccs_list_table() call.

`ccs_result_table` *Given a list of CCS model, produce a summary table of the results*

Description

Given a list of CCS model, produce a summary table of the results

Usage

```
ccs_result_table(model_list, corpus, Z)
```

Arguments

`model_list`	A list of CCS models, or the result of a ccs_list_table() call.
`corpus`	The corpus (as a list of strings)
`Z`	The binary treatment variable

Value

A data frame containing summary of the results.

`ccs_tuned_textreg` *Find threshold C and run textreg with that tuning parameter.*

Description

Find threshold C and run textreg with that tuning parameter.

Usage

```
ccs_tuned_textreg(corpus, Z, R = 20, beta = 0.8, ...)
```

Arguments

`R`	The number of randomizations to use for the null distribution.
`beta`	The quantile of the null distribution to use for regularization.
`...`	Additional arguments to pass to textreg. See textreg::textreg for more information.

Value

Textreg result object with the passed tuning parameter.

clean_features *Clean and simplify feature set*

Description

Given set of generated features, simplify set of features using the carat package.

Usage

```
clean_features(
  meta,
  ignore = NULL,
  remove.lc = TRUE,
  uniqueCut = 1,
  freqCut = 99,
  cor = 0.95,
  verbose = FALSE
)
```

Arguments

`meta`	A matrix or dataframe containing the set of only numeric features to be simplified.
`ignore`	List of column names to ignore when simplifying (e.g., ID column and other columns such as special features that should be preserved).
`remove.lc`	TRUE means remove collinear combinations of features.
`uniqueCut`	Param passed to carat's nearZeroVar
`freqCut`	Param passed to carat's nearZeroVar
`cor`	Cutoff of how correlated features should be before dropping one.
`verbose`	Print out progress to console.

Value

Updated 'meta' in matrix form with fewer columns of the preserved features.

Examples

```
## Example 1
# Create a 10x20 matrix with random numbers between 0 and 1
random_matrix <- matrix(runif(200), nrow = 10, ncol = 20)

# Create a 10x20 dataframe with random numbers between 0 and 1
random_df <- data.frame(matrix(runif(200), nrow = 10, ncol = 20))

# Apply the clean_features function to the matrix
cleaned_matrix <- clean_features(random_matrix)

# Apply the clean_features function to the dataframe and convert back to dataframe
cleaned_df <- data.frame(clean_features(random_df))

## Example 2
```

```
# Apply the clean_features function with a correlation cutoff (0.10)
cleaned_matrix <- clean_features(random_matrix, cor = 0.10)

# Apply the clean_features function to the dataframe with a correlation cutoff (0.90)
cleaned_df <- data.frame(clean_features(random_df, cor = 0.50))
```

`clean_text` *Generic text cleaning function*

Description

Quick and easy text cleaning. Take given copus and remove punctuation, remove whitespace, convert everything to lowercase. This function is used for pre-processing text within the generate_features function.

Usage

```
clean_text(x, split_hyphens = TRUE)
```

Arguments

`x`	Character vector or corpus object.
`split_hyphens`	A logical indicating whether hyphenated words should be treated as two tokens (split at the hyphen).

Details

Strings can be collapsed to an empy string. E.g., " . " will change to "".

Examples

```
## Typo correction: given corpus x -> given corpus

# Texts to be cleaned
txts = c( "THIS FUNCTION CONVERTS EVERYTHING TO LOWERCASE.",
          "This function removes punctuation........",
          "This function removes     whitespace.",
          "This-function-splits-hyphens",
          "The main use for clean_text is to check which elements of
           your text are converted to empty strings.",
          " " )

## Example 1: Convert to Lowercase
txts[1]
clean_text( txts[1] )

## Example 2: Remove Punctuation
txts[2]
clean_text( txts[2] )

## Example 3: Remove White spaces
txts[3]
clean_text( txts[3] )
```

```
## Example 4: Split Hyphens
txts[4]
clean_text( txts[4] )
clean_text( txts[4], split_hyphens = FALSE )

## Example 5: Clean Entire Vector of Texts)
clean_text( txts )
```

`essay_grader` *Grading Essays or Generating Responses with ChatGPT*

Description

This function grades essays or generates responses based on given prompts and texts using a specified model. To run this function, you need to have your own ChatGPT API key.

This function interacts with the ChatGPT API to generate responses based on a given prompt. The function allows for a global fallback mechanism for the API key if not provided explicitly.

Usage

```
essay_grader(text, prompt, model)

ChatGPT(prompt, model = "gpt-3.5-turbo", my_API = NULL, temp = 0)
```

Arguments

`text`	A vector of text strings to be graded or used as input for generating responses.
`prompt`	A string containing the prompt to be sent to the ChatGPT model.
`model`	A string specifying the model to be used, with "gpt-3.5-turbo" as the default.
`my_API`	A string containing the ChatGPT API key. If NULL, the function will attempt to use an API key stored in the global environment.
`temp`	A numeric value between 0 and 1 that controls the randomness of the model's output, with 0 as the default for deterministic results.

Value

A data frame with generated responses or graded results for each input text.

A string containing the generated response from the ChatGPT model.

Examples

```
# Assign your own ChatGPT API key
my_API <- "your-api-key-here"

# Define example texts and prompt
texts <- c("This is the first essay.", "Here is another essay.")
prompt <- "Evaluate the quality of this essay."

# Specify the model to use
```

```
model <- "gpt-3.5-turbo"

# Run the essay_grader function
graded_essays <- essay_grader(texts, prompt, model)

# View the results
print(graded_essays)
# Example usage with explicit API key
response <- ChatGPT("What is the capital of France?", model = "gpt-3.5-turbo", my_API = "your-api-key-here")

# Example usage with global API key
my_API <- "your-api-key-here"
response <- ChatGPT("What is the capital of France?", model = "gpt-3.5-turbo")

# Example usage with different temperature
response <- ChatGPT("Generate a creative story", model = "gpt-3.5-turbo", temp = 0.7)
```

---

estimate_impacts *Estimate treatment impacts for hybrid-scored text outcomes*

Description

Given text from a randomized trial with a binary treatment, where a subset of the documents have been human-scored, this function computes model-assisted estimates for the average treatment effect with respect to the human-coded outcome.

Usage

```
estimate_impacts(
  y.obs,
  yhat,
  Z,
  wts = NULL,
  design = c("crd", "multi", "cluster", "rcbd"),
  siteID = NULL,
  clusterID = NULL,
  data,
  adjust = NULL
)
```

Arguments

`y.obs`	A vector of human-coded scores (with NAs for unscored documents).
`yhat`	A vector of predicted scores estimated via `predict_scores`.
`Z`	Indicator for treatment assignment.
`wts`	Sampling weights for which documents were human scored. Assumed uniform if null.
`design`	Type of design used for random assignment (complete randomization, multisite randomized, cluster randomized, and blocked and cluster randomized).
`siteID`	Vector of IDs for site, for multi-site randomized experiments.

`clusterID`	Vector of IDs for cluster, for cluster-randomized experiments.
`data`	A `data.frame` of subject-level identifiers, demographic variables, group membership, and/or other pre-treatment covariates.
`adjust`	(optional) character vector or named list of variables in the data matrix to adjust for when estimating treatment impacts.

Value

A model object for estimating treatment impact across an array of features.

`eval_metrics` *Evaluate Metrics for Machine Learning Iterations*

Description

This function calculates evaluation metrics for the results of machine learning iterations. Depending on the type of outcome variable (continuous or categorical), the function computes different sets of metrics and returns them in a summarized format.

Usage

```
eval_metrics(result = NULL, outcome = NULL)
```

Arguments

`result`	A list of results from ‘ML_iteration‘ or ‘ML_iterations‘. Each element of the list should be a list containing the results of a single iteration, including the calculated metrics.
`outcome`	Type of outcome variable: either "continuous" for continuous outcomes or "categorical" for categorical outcomes.

Value

A data frame containing the evaluation metrics. For continuous outcomes, the metrics include: - ‘rmse‘: Root Mean Squared Error - ‘r_squared‘: R-squared value - ‘mae‘: Mean Absolute Error

For categorical outcomes, the metrics include: - ‘accuracy‘: Accuracy - ‘accuracy_ll‘: Lower limit of the accuracy confidence interval - ‘accuracy_ul‘: Upper limit of the accuracy confidence interval - ‘uwk‘: Unweighted kappa - ‘qwk‘: Quadratic weighted kappa

Examples

```
# Load texts
data("toy_reads")

# Generate text features
feats = generate_features( toy_reads$text, meta=toy_reads,
                           sent = TRUE,
                           clean_features = TRUE,
                           read = c("Flesch","Flesch.Kincaid", "ARI"),
                           ld=c("TTR","R","K"),
                           ignore=c("ID"),
                          verbose = TRUE )

# Preprocess the feature space to remove collinear features
# and features with near-zero variance
X_all = dplyr::select( feats,
                       -ID, -Q1, -Q2, -text, -more )
X_all = predict(caret::preProcess( X_all, method = c("nzv","corr"),
                                   uniqueCut=2, cutoff=0.95), X_all )
caret::findLinearCombos(X_all) # sanity check to make sure no redundant features

# Transform all variables as numeric variables
X_all[] <- lapply(X_all,
                  function(x) if(is.character(x)) as.numeric(as.factor(x)) else x)

# Extract outcome variables
all_Scores <- toy_reads$Q1

## Set parameters
X <- X_all
Y <- all_Scores
n_iter <- 2
n_tune <- 2
control <- caret::trainControl(method = 'cv')
preProc <- 'zv'
outcome <- 'continuous'
best_mod <- 'rf'

## Define the percentages for training portions
percentages <- c(.20, 0.40, 0.60, 0.80)

## Loop through each percentage
random_forest_Scores = ML_iterations (x = X, y = Y,
                                      n_iteration = n_iter,
                                      training_portions = percentages,
                                      trCon = control,
                                      preProc = preProc,
                                      n.tune = n_tune,
                                      model = best_mod,
                                      outcome = outcome)
## Evaluate the metrics
eval_metrics(random_forest_Scores, outcome = outcome)
```

eval_model — *Evaluate In-Sample and Out-of-Sample Performance of a Machine Learning Model*

Description

This function calculates and records various performance measures for a machine learning model, including in-sample (training) and out-of-sample (testing) metrics.

Usage

```
eval_model(yhat, Yobs, coded)
```

Arguments

`yhat`	A numeric vector of predicted outcomes for the entire dataset.
`Yobs`	A numeric vector of observed outcomes (ground truth) corresponding to the predictions in 'yhat'.
`coded`	A binary vector indicating whether a document was used for training ('1') or testing ('0').

Value

A data frame with the following performance measures for both in-sample and out-of-sample data:

- `ybar`: The mean of the predicted outcomes.
- `var.yhat`: The variance of the predicted outcomes.
- `R2`: The coefficient of determination (R-squared), indicating how well the predictions match the observed outcomes.
- `mse`: The mean squared error, representing the average squared difference between the predicted and observed outcomes.

Examples

```
# Example usage with hypothetical data
yhat <- rnorm(100)
Yobs <- rnorm(100)
coded <- sample(c(0, 1), 100, replace = TRUE)
performance <- eval_model(yhat, Yobs, coded)
print(performance)
```

`extract_liwc` *Functions for processing and appending output from Linguistic Inquiry Word Count (LIWC) software*

Description

Functions for processing and appending output from Linguistic Inquiry Word Count (LIWC) software

Usage

```
extract_liwc(file, meta = NULL, ID.liwc = 1, ID.meta = NULL, clean = TRUE)
```

Arguments

`file`	character path to LIWC output file
`meta`	optional data frame to attach LIWC output to
`ID.liwc`	ID column (either name or column number) of document IDs in the liwc file.
`ID.meta`	If meta is specified, character vector of the document ID column to use for merging, corresponding with IDs in ID.liwc.
`clean`	should LIWC output be cleaned prior to appending (e.g., remove linear combinations, repeat variables, etc.)

Examples

```
# This function extracts features from a LIWC-generated-generated data
# (CSV or dataframe).
# The txt_data should be a dataframe
# and have the same column name(s) as "reads_liwc".

data("reads_liwc")
data("example_meta")

reads_liwc = reads_liwc
txt_data = meta

all.feats = extract_liwc( file = "reads_liwc",
                          meta = txt_data,
                          ID.liwc = c( "ID" ),
                          ID.meta = c( "ID" ),
                          clean = FALSE )
```

extract_taaco *Import and merge text features generated using TAACO*

Description

This method helps support feature extraction using TAACO. You will have to use the external TACCO program to generate these features; these methods just help move back and forth from R to TACCO.

Usage

```
extract_taaco(
  file,
  meta = NULL,
  ID.meta = NULL,
  drop_para = FALSE,
  drop_sent = TRUE
)
```

Arguments

file	Filename where TAACO results are stored
meta	Optional `data.frame` with additional document-level variables to include in output.
ID.meta	If `meta` is specified, character vector with name of variables used for merging.
drop_para	Drop paragraph level measures of cohesion from features (TRUE/FALSE).
drop_sent	Drop adjacent overlap between sentences (TRUE/FALSE).

Details

See `prep_external()` for generating text files that would be ready for TAACO analysis. Call this on the corpus to make files that can be read in and processed easily.

Once external processing is complete, `extract_taaco()` reads output and log files produced by the TAACO program and returns a `data.frame` that can be merged with other feature sets.

The "Filename" column in the read file should be the IDs (with a '.txt' suffix that will be dropped). The results can then be merged with 'meta', if passed,

Value

Returns a `data.frame` of text features.

Examples

```
# This function extracts features from a TAACO-generated data
# (CSV or dataframe).
# The txt_data should be a dataframe
# and have the same column name(s) as "reads_tacco"

data("reads_taaco")
data("example_meta")

reads_tacco = reads_taaco
txt_data = meta

all.feats = extract_taaco( "reads_taaco",
                            meta = txt_data,
                            ID.meta = "ID" )
```

---

extract_w2v — *Compute document-level feature vectors from a pre-trained embedding model.*

Description

This function generates a vector embedding for each word in a string using a set of pre-trained word vectors such as GloVe (Pennington et al. 2014) and returns the mean vector projection across all words in a document.

Usage

```
extract_w2v(x, meta = NULL, model = NULL)
```

Arguments

`x`	A corpus object or character vector of text documents.
`meta`	Dataframe corresponding to the corpus x. If passed, and non-NULL, all generated features will be added to this dataframe. If NULL, a dataframe of just the features will be returned.

model	User-specified model object pointing to a custom pre-trained embedding model, represented as a matrix or data frame where the first column is the word/token and the following columns are numeric vectors. If NULL, use default "mini_glove" embeddings on 1000 common words (not recommended).

Value

A list of data frames containing the Word2Vec projections of the corpus

References

references Mikolov T, Chen K, Corrado G, Dean J (2013). "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781.* Pennington J, Socher R, Manning C (2014). "Glove: Global vectors for word representation." In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Examples

```
# The txt_data should be a dataframe

library(textdata)
data("example_meta")

txt_data = meta
glove.50d = embedding_glove6b(dimensions = 50)

all.feats = extract_w2v( clean_text( text$text ),
                         meta = txt_data,
                         model = glove.50d )
```

`generate_distance_features` *Generate distances to a set of reference documents*

Description

This is a helper function for the pairwise_distances method.

Usage

```
generate_distance_features(
  x,
  meta = NULL,
  reference_docs,
  verbose = FALSE,
  method = c("cosine", "w2v"),
  model = NULL,
  use_names = TRUE,
  names_prefix = "doc_"
)
```

Arguments

`x`	A corpus object or character vector of text documents.
`meta`	Dataframe corresponding to the corpus x. If passed, and non-NULL, all generated features will be added to this dataframe. If NULL, a dataframe of just the features will be returned.
`reference_docs`	A list of reference documents to compare corpus against. Will end up with one distance feature per reference doc.
`model`	When using w2v, a user-specified model object pointing to a pre-trained embedding model, represented as a data frame where the first column is the word/token and the following columns are numeric vectors.

See Also

`pairwise_distances`

`generate_features` *Generate an array of text features*

Description

Generates a rich feature representation for documents provided as a character vector or quanteda::corpus() object by applying an array of linguistic and syntactic indices, available text analysis dictionaries, and pre-trained embedding models to all documents.

Usage

```
generate_features(
  x,
  meta = NULL,
  lex = TRUE,
  sent = TRUE,
  ld = "all",
  clean_features = TRUE,
  read = c("ARI", "Coleman", "DRP", "ELF", "Flesch", "Flesch.Kincaid",
    "meanWordSyllables"),
  terms = NULL,
  preProc = list(uniqueCut = 1, freqCut = 99, cor = 0.95, remove.lc = TRUE),
  verbose = FALSE,
  ignore = NULL,
  ...
)
```

Arguments

`x`	A corpus object or character vector of text documents.
`meta`	Dataframe corresponding to the corpus x. If passed, and non-NULL, all generated features will be added to this dataframe. If NULL, a dataframe of just the features will be returned.
`lex`	Logical, indicating whether to compute lexical indices including measures of lexical diversity, readability, and entropy

`sent`	Logical, indicating whether to compute sentiment analysis features from available dictionaries
`ld`	character vector defining lexical diversity measures to compute; see quanteda.textstats::textstat_lexdiv
`clean_features`	TRUE means implement cleaning step where features with no variation and collinear features are dropped. (This happens before any term generation features are added.)
`read`	character vector defining readability measures to compute; see quanteda.textstats::textstat_readability
`terms`	character vector of terms to evaluate as standalone features based on document-level frequency (case-insensitive). Not cleaned by clean_features.
`preProc`	Named list of arguments passed to `clean_features()` for applying pre-processing transformations across the set of text features (e.g., removing collinear features).
`ignore`	List of column names (features) to ignore when simplifying (e.g., ID column and other columns that should be preserved). All columns in meta, if meta is non-null, will be added to this list automatically.
`...`	(optional) additional arguments passed to quanteda::tokens() for text pre-processing.

Details

Note: This function does not work with a dataframe with one row. The 'meta' argument requires a dataframe with at least two rows.

Value

A data.frame of available text features, one row per document, one column per feature (with additional columns from meta, if meta is not NULL).

Examples

```
## Example 1: Basic Feature Generation

# Create a small dataframe with 2 texts (objects).
df = data.frame(
 text = c( "This function generates an array of text features",
          "This function generates a rich feature representation as a character
           vector or quanteda::corpus() object by applying an array of linguistic
           and syntactic indices" ))

# Generated text features without simplifying the set of features
feats1 = generate_features( df$text,
                            meta = df,
                            clean_features = FALSE )


## Example 2: Feature Generation with Example Data and Customization

# Load example dataframe with multiple texts
data( "toy_reads" )

# Generate text features without simplifying the set of features
feats2 = generate_features( toy_reads$text,
                            meta=toy_reads,
                            clean_features = FALSE,
                            ignore = "ID" )

# Generate preliminary text features,
# simplifying the set of features and specifying sent, read, ld
feats3 = generate_features( toy_reads$text, meta=toy_reads,
                            sent = TRUE,
                            clean_features = TRUE,
                            read = c("Flesch","Flesch.Kincaid", "ARI"),
                            ld=c("TTR","R","K"),
                            ignore=c("ID"),
                            verbose = TRUE )
```

generate_predictions *Add predictions to a meta dataframe*

Description

Given a set of models (as a list) add predictions from model to a passed dataframe

Usage

```
generate_predictions(models, features, meta = NULL, prefix = "mod_")
```

Arguments

models	List of models from, e.g., train_models
features	A dataframe of features that correspond to the features the models were trained on (but possibly for different data).
meta	Dataframe to add predictions to. If NULL, just return a dataframe of the predictions.

get.ests *Calculate Synthetic and Model-Assisted Estimators*

Description

This function calculates synthetic and model-assisted estimators, along with their variances, based on the provided data.

Usage

```
get.ests(yhat, coded, data)
```

Arguments

yhat	A numeric vector of predicted outcomes for the entire dataset.
coded	A binary vector indicating which documents have been hand-coded (1 for coded, 0 for not coded).
data	A data frame containing the full dataset, including observed outcomes (‘Yobs‘), predicted outcomes (‘yhat‘), treatment indicator (‘Z‘), and propensity scores (‘pi.hat‘).

Value

A data frame with the following components:

- `est`: The effect estimate using the model-assisted estimator.
- `SE`: The standard error of the true variance of the estimate.
- `SEhat`: The estimated standard error of the estimate based on partial coding.

Examples

```
# Example usage with hypothetical data
data <- data.frame(Yobs = rnorm(100), Z = sample(c(0, 1), 100, replace = TRUE), pi.hat = runif(100))
yhat <- rnorm(100)
coded <- sample(c(0, 1), 100, replace = TRUE)
results <- get.ests(yhat, coded, data)
print(results)
```

`get.sub` *Calculate Subset Estimator*

Description

This function calculates the subset estimator for a given dataset. It estimates the treatment effect and its variance based on a subset of the data.

Usage

```
get.sub(coded, data)
```

Arguments

`coded`	A binary vector indicating which samples are part of the subset (1) and which are not (0).
`data`	A data frame containing the observed data. It must include the columns ‘Z‘ (treatment indicator), ‘Yobs‘ (observed outcome), and ‘pi.hat‘ (sampling probabilities).

Value

A data frame containing the estimated treatment effect (‘est‘), the standard error (‘SE‘), and the estimated standard error (‘SEhat‘).

Examples

```
# Example usage with hypothetical data
coded <- c(1, 0, 1, 0, 1)
data <- data.frame(Z = c(1, 0, 1, 0, 1),
                   Yobs = c(5, 3, 6, 2, 7),
                   pi.hat = c(0.8, 0.5, 0.9, 0.6, 0.7))
result <- get.sub(coded, data)
print(result)
```

`get_ML_est` *Get All Estimates: Nominal (Oracle), Subset, Synthetic, and Model-Assisted*

Description

This function calculates and returns four types of estimates: nominal (oracle), subset, synthetic, and model-assisted.

Usage

```
get_ML_est(dat, coded, yhat)
```

Arguments

`dat`	A data frame containing the full dataset, including both observed and predicted outcomes.
`coded`	A binary vector indicating which documents have been hand-coded (1 for coded, 0 for not coded).
`yhat`	A numeric vector of predicted outcomes for the full dataset.

Value

A data frame containing the following estimates:

- `nominal.est`, `nominal.SE`, `nominal.SEhat`: Estimates based on the full hand-coded dataset.
- `subset.est`, `subset.SE`, `subset.SEhat`: Estimates based on the sample of hand-coded documents alone.
- `synth.est`, `synth.SE`, `synth.SEhat`: Oracle estimates based on predictions alone.
- `ML.est`, `ML.SE`, `ML.SEhat`: Model-assisted estimates that combine observed and predicted outcomes.

Examples

```
# Example usage with hypothetical data
dat <- data.frame(Yobs = rnorm(100), Z = sample(c(0, 1), 100, replace = TRUE))
coded <- sample(c(0, 1), 100, replace = TRUE)
yhat <- rnorm(100)
results <- get_ML_est(dat, coded, yhat)
print(results)
```

impacts_on_features *For each column of x, conduct an analysis of impact of MORE intervention on feature represented by that column.*

Description

Adjust all tests with FDR at end. For an example function for analysis_function see simple_RCT_analysis.

Usage

```
impacts_on_features(
  data,
  meta = NULL,
  ignore = NULL,
  analysis_function = simple_RCT_analysis,
  mcp = "fdr",
  planned_features = NULL,
  standardize = TRUE,
  alpha = 0.05,
  ...
)
```

Arguments

`data`	Dataframe of the meta information and features. Will calculate impacts on all columns of data that are not specified by ignore.
`meta`	Extra information that are not text features to be analyzed (e.g. treatment assignment indicator) that should be given to each analysis for each feature (this is a dataframe)
`ignore`	Character vector of column names to ignore in the analysis. Or list of column indices.
`analysis_function`	Function to apply to each column of data. Should return a one-row tibble with columns such as Estimate, SE, t, p, CI_l, CI_h. Function is of form f(feature, data, ...) with feature being a vector of values, with length equal to number of rows of data. Function results MUST return a column 'p.value' of the p.value of the test.
`mcp`	character string specifying the correction method to be applied to adjust for multiple comparisons. Defaults to fdr adjustments. See p.adjust for available adjustment methods. NULL means no adjustment.
`planned_features`	Character vector of features that are planned for assessment. They will never be screened out, regardless of their p-value. Multiple comparison is done within group of planned and unplanned features.
`standardize`	Logical. If TRUE, standardize the effect sizes in new columns of output with suffix "_std". If "pool", pool the standard deviations of the two treatment groups to get the standardize metric.
`alpha`	Significance level for hypothesis tests.
`...`	Additional arguments to pass to analysis_function.

See Also

simple_RCT_analysis

`mini_glove` *Mini glove dataset*

Description

glove embeddings (50 dimensional) for 1000 common words beyond those words listed in several stopword lists provided by quanteda.

Usage

```
mini_glove
```

Format

A matrix with 1000 rows and 50 columns

Details

Original glove embeddings downloaded from Stanford CITE.

1000 by 50 matrix, each row is an embedding. Rownames are words.

`ML_iteration` *Iterative Machine Learning with Performance Tracking*

Description

Trains a machine learning model multiple times, splitting the data into training and testing sets for each iteration. Evaluates and stores performance metrics for each run, allowing for assessment of model stability and generalization.

Trains and evaluates a specified machine learning model iteratively, using different proportions of the dataset for training in each iteration. This function facilitates the exploration of how model performance varies with changes in the amount of training data. Hyperparameter tuning can be optionally incorporated to optimize the model within each iteration.

Usage

```
ML_iteration(
  x,
  y,
  n_iteration = 1,
  training_portion = 0.8,
  model = NULL,
  outcome = NULL,
  ...
)
```

```
ML_iterations(
  x,
  y,
  n_iteration = 1,
  training_portions = 0.8,
  model = NULL,
  outcome = NULL,
  ...
)
```

Arguments

`x`	Matrix or data frame of predictor variables.
`y`	Vector of outcome variable (continuous or categorical).
`n_iteration`	Number of times to repeat model training and evaluation for each training set size (defaults to 1).
`training_portion`	Proportion of data used for training (defaults to 0.8).
`model`	Machine learning model to use (e.g., "rf", "xgbTree").
`outcome`	Type of outcome variable: "continuous" or "categorical".
`...`	Extra arguments to be passed to carat.
`training_portions`	A vector of proportions (between 0 and 1) indicating the fraction of data to be used for training in each iteration.

Details

This function provides a standardized way to train and evaluate models within a larger analysis, facilitating comparison across different models, hyperparameters, or repeated runs. It handles both regression (continuous outcome) and classification (categorical outcome) problems, providing appropriate evaluation metrics for each.

This function helps you understand how your chosen model performs with varying amounts of training data, which is crucial for assessing its potential in real-world scenarios with limited data. The inclusion of hyperparameter tuning can further optimize the model's performance for each training set size.

Value

A list containing results for each iteration, including: * The trained model (‘ML_model‘) * Predicted values on the test set (‘predicted_values‘) * Values of predictor variables on the test set (‘test_values‘) * Performance metrics (RMSE, MAE, correlation, R-squared for continuous; confusion matrix, accuracy, kappa values for categorical) * Execution time for each iteration (‘execution_time‘)

A nested list, where the first level corresponds to different training set sizes and the second level contains results for each iteration within that training set size. Each iteration's results include: * The trained model (‘ML_model‘) * Predicted values on the test set (‘predicted_values‘) * Values of predictor variables on the test set (‘test_values‘) * Performance metrics (RMSE, MAE, correlation, R-squared for continuous; confusion matrix, accuracy, kappa values for categorical) * Execution time for each iteration (‘execution_time‘)

See Also

* ‘ML_iterations‘: A wrapper function for performing multiple iterations of ‘ML_iteration‘ with different training proportions. * ‘caret::train‘: The underlying function used for model training.

* ‘ML_iteration‘: The core function performing a single iteration of model training and evaluation. * ‘caret::train‘: The underlying function used for model training.

Examples

```
# Load texts
data("toy_reads")

# Generate text features
feats = generate_features( toy_reads$text, meta=toy_reads,
                           sent = TRUE,
                           clean_features = TRUE,
                           read = c("Flesch","Flesch.Kincaid", "ARI"),
                           ld=c("TTR","R","K"),
                           ignore=c("ID"),
                           verbose = TRUE )

# Preprocess the feature space to remove collinear features
# and features with near-zero variance
X_all = dplyr::select( feats,
                       -ID, -Q1, -Q2, -text, -more )
X_all = predict(caret::preProcess( X_all, method = c("nzv","corr"),
                                   uniqueCut=2, cutoff=0.95), X_all )
caret::findLinearCombos(X_all) # sanity check to make sure no redundant features

# Transform all variables as numeric variables
X_all[] <- lapply(X_all,
                  function(x) if(is.character(x)) as.numeric(as.factor(x)) else x)

# Extract outcome variables
all_Scores <- toy_reads$Q1

## Set parameters
X <- X_all
Y <- all_Scores
n_iter <- 2
n_tune <- 2
control <- caret::trainControl(method = 'cv')
preProc <- 'zv'
outcome <- 'continuous'
portion <- 0.7
best_mod <- 'rf'

## Run ML_iteration
random_forest_Score = ML_iteration (x = X, y = Y,
                                    n_iteration = n_iter,
                                    training_portion = portion,
                                    trCon = control,
                                    preProc = preProc,
                                    n.tune = n_tune,
                                    model = best_mod,
                                    outcome = outcome)
```

```
# Load texts
data("toy_reads")

# Generate text features
feats = generate_features( toy_reads$text, meta=toy_reads,
                           sent = TRUE,
                           clean_features = TRUE,
                           read = c("Flesch","Flesch.Kincaid", "ARI"),
                           ld=c("TTR","R","K"),
                           ignore=c("ID"),
                           verbose = TRUE )
# Preprocess the feature space to remove collinear features
# and features with near-zero variance
X_all = dplyr::select( feats,
                       -ID, -Q1, -Q2, -text, -more )
X_all = predict(caret::preProcess( X_all, method = c("nzv","corr"),
                                   uniqueCut=2, cutoff=0.95), X_all )
caret::findLinearCombos(X_all) # sanity check to make sure no redundant features

# Transform all variables as numeric variables
X_all[] <- lapply(X_all,
                  function(x) if(is.character(x)) as.numeric(as.factor(x)) else x)

# Extract outcome variables
all_Scores <- toy_reads$Q1

## Set parameters
X <- X_all
Y <- all_Scores
n_iter <- 2
n_tune <- 2
control <- caret::trainControl(method = 'cv')
preProc <- 'zv'
outcome <- 'continuous'
best_mod <- 'rf'

## Define the percentages for training portions
percentages <- c(.20, 0.40, 0.60, 0.80)

## Loop through each percentage
random_forest_Scores = ML_iterations (x = X, y = Y,
                                      n_iteration = n_iter,
                                      training_portions = percentages,
                                      trCon = control,
                                      preProc = preProc,
                                      n.tune = n_tune,
                                      model = best_mod,
                                      outcome = outcome)
```

ML_plot *Create a Line Plot with Points using ggplot2*

Description

This function creates a line plot with points using the ggplot2 package. It allows for extensive customization of aesthetics and layout.

Usage

```
ML_plot(
  data = NULL,
  x = NULL,
  y = NULL,
  color = NULL,
  group = NULL,
  ylim_min = NULL,
  ylim_max = NULL,
  x_lable = NULL,
  y_lable = NULL,
  fig_title = NULL,
  legend_title = "",
  legend.position = "bottom"
)
```

Arguments

`data`	A dataframe containing the data to be plotted.
`x`	A string representing the column name for the x-axis.
`y`	A string representing the column name for the y-axis.
`color`	A string representing the column name to be used for coloring the lines and points.
`group`	A string representing the column name to be used for grouping the lines.
`ylim_min`	A numeric value representing the minimum limit for the y-axis.
`ylim_max`	A numeric value representing the maximum limit for the y-axis.
`x_lable`	A string for the label of the x-axis.
`y_lable`	A string for the label of the y-axis.
`fig_title`	A string for the title of the plot.
`legend_title`	A string for the title of the legend.
`legend.position`	A string representing the position of the legend in the plot (default is "bottom").

Value

A ggplot object representing the created plot.

Examples

```
# Load texts
data("toy_reads")

# Generate text features
feats = generate_features( toy_reads$text, meta=toy_reads,
                           sent = TRUE,
                           clean_features = TRUE,
                           read = c("Flesch","Flesch.Kincaid", "ARI"),
                           ld=c("TTR","R","K"),
                           ignore=c("ID"),
                           verbose = TRUE )

# Preprocess the feature space to remove collinear features
# and features with near-zero variance
X_all = dplyr::select( feats,
                       -ID, -Q1, -Q2, -text, -more )
X_all = predict(caret::preProcess( X_all, method = c("nzv","corr"),
                                   uniqueCut=2, cutoff=0.95), X_all )
caret::findLinearCombos(X_all) # sanity check to make sure no redundant features

# Transform all variables as numeric variables
X_all[] <- lapply(X_all,
                  function(x) if(is.character(x)) as.numeric(as.factor(x)) else x)

# Extract outcome variables
all_Scores <- toy_reads$Q1

## Set parameters
X <- X_all
Y <- all_Scores
n_iter <- 2
n_tune <- 2
control <- caret::trainControl(method = 'cv')
preProc <- 'zv'
outcome <- 'continuous'

## Define the percentages for training portions
percentages <- c(.20, 0.40, 0.60, 0.80)

## Loop through each percentage
best_mod <- 'rf'
random_forest_Scores = ML_iterations ( x = X, y = Y,
                                       n_iteration = n_iter,
                                       training_portions = percentages,
                                       trCon = control,
                                       preProc = preProc,
                                       n.tune = n_tune,
                                       model = best_mod,
                                       outcome = outcome )

best_mod <- 'rrf'
regularized_rf_Scores = ML_iterations ( x = X, y = Y,
                                        n_iteration = n_iter,
                                        training_portions = percentages,
                                        trCon = control,
                                        preProc = preProc,
                                        n.tune = n_tune,
                                        model = best_mod,
                                        outcome = outcome )

## Evaluate the metrics
eval_random_forest_Scores = eval_metrics( random_forest_Scores,
                                          outcome = outcome )
```

```
eval_regularized_rf_Scores = eval_metrics( regularized_rf_Scores,
                                           outcome = outcome )
## Create data frames for data visualization
all_Scores = rbind( eval_random_forest_Scores,
                    eval_regularized_rf_Scores )


## Create a line plot (Continuous outcomes)
ML_plot( data = all_Scores,
         x = "training", y = "rmse", color = "model", group = "model",
         ylim_min= 0.5, ylim_max= 0.7,
         x_lable = "Training Set (%)", y_lable = "RMSE",
         fig_title = "RMSE by Training set % for Different Models",
         legend_title = "", legend.position = "bottom" )
```

`pairwise_distances` *Calculate all pairwise distances between documents in two corpus.*

Description

Calculate all pairwise distances between documents in two corpus.

Usage

```
pairwise_distances(
  corpus1,
  corpus0,
  method = c("cosine", "w2v"),
  wide = TRUE,
  model = NULL,
  use_names = TRUE,
  names_prefix = "doc_"
)
```

Arguments

`wide`	If TRUE return distances with each column corresponding to a document in corpus0, and each row a document in corpus1. If FALSE return dataframe of indices for each pair, with the third column being distance.
`model`	When using w2v, a user-specified model object pointing to a pre-trained embedding model, represented as a data frame where the first column is the word/token and the following columns are numeric vectors.

Value

Dataframe of either length(corpus1) rows or length(corpus1)*length(corpus0) rows, depending on wide.

plot_ccs — *Plot the results from a CCS run*

Description

This function provides a visualization of the set of words and phrases found to differ systematically between treatment and control groups

Usage

```
plot_ccs(
  out,
  xlim = NULL,
  xadj = c(-0.025, 0.025),
  xlab = "Difference in Usage Rates",
  sub = "(Treatment - Control)",
  ...
)
```

Arguments

`out`	a textreg.result() object
`xlim`	limits for x-axis
`xadj`	adjustments to the lower and upper limits on the x-axis of the plot
`...`	additional arguments passed to plot

plot_textfx — *Plot the results from an impact analysis with text outcomes*

Description

This function provides a visualization of the set of textual features found to differ systematically between treatment and control groups.

Usage

```
plot_textfx(out, alpha = 0.05, cols = FALSE, group = TRUE, xlim = NULL, ...)
```

Arguments

`out`	a model object output from `estimate_impacts()`
`alpha`	the threshold for determining statistical significance
`cols`	should effects be colored by direction (red for negative impacts, blue for positive impacts)
`group`	(optional) should effects be grouped by category (e.g., higher-level summary measures, linguistic features, etc.)
`xlim`	(optional) bounds for x-axis
`...`	additional arguments passed to plot

predict_scores *Extract predictions from a fitted text scoring model.*

Description

This function calculates predicted scores for all documents based on fitted models. It handles both in-sample and out-of-sample predictions for different groups.

Usage

```
predict_scores(fit, X, Z)
```

Arguments

`fit`	A list containing fitted models for two groups (typically treatment and control). The list should include ‘mod0‘ for group 0 and ‘mod1‘ for group 1, along with a binary vector ‘coded‘ indicating which documents are part of the coded subset.
`X`	A data frame or matrix of predictor variables used for making predictions.
`Z`	A binary vector indicating group membership (e.g., 0 for control, 1 for treatment).

Value

A numeric vector of predicted scores for all documents.

Examples

```
# Example usage with hypothetical data
fit <- list(mod0 = lm(Y ~ X1 + X2, data = data0), mod1 = lm(Y ~ X1 + X2, data = data1), coded = c(1, 0, 1, 0, 1))
X <- data.frame(X1 = rnorm(100), X2 = rnorm(100))
Z <- sample(c(0, 1), 100, replace = TRUE)
yhat <- predict_scores(fit, X, Z)
print(yhat)
```

prep_external *Prepare text documents for analysis using external programs*

Description

Text pre-processing and corpus management functions to provide compatibility with external text analysis programs and standalone software packages such as Linguistic Inquiry Word Count (LIWC), the Tool for Automated Analysis of Cohesion (TAACO) and the Sentiment Analysis and Social Cognition Engine (SEANCE).

Usage

```
prep_external(x, dir, docnames = NULL, preProc = NULL)
```

Arguments

`x`	A corpus object or character vector of text documents.
`dir`	Name of directory where the generated intermediate text files should be stored.
`docnames`	Optional character string specifying file names for each document in `x`.
`preProc`	Optional text pre-processing function(s) (e.g., stemming) to apply prior to writing text files for analysis in external programs.

References

Pennebaker JW, Booth RJ, Boyd RL, Francis ME (2015). "Linguistic Inquiry and Word Count: LIWC 2015." www.liwc.net. Crossley SA, Kyle K, McNamara DS (2016). "The tool for the automatic analysis of text cohesion (TAACO): Automatic assessment of local, global, and text cohesion." *Behavior research methods*, **48**(4), 1227–1237. Crossley SA, Kyle K, McNamara DS (2017). "Sentiment Analysis and Social Cognition Engine (SEANCE): An automatic tool for sentiment, social cognition, and social-order analysis." *Behavior research methods*, **49**(3), 803–821.

`rcttext` *The 'rcttext' package*

Description

The 'rcttext' package provides a set of functions for working with text data in R, when that text is an outcome in a randomized controlled trial. The package includes functions for cleaning and preprocessing text data, as well as for applying text mining techniques such as sentiment analysis and topic modeling.

Author(s)

Maintainer: Reagan Mozer <`rmozer@g.harvard.edu`> [copyright holder]

Authors:

- Luke Miratrix <`lmiratrix@g.harvard.edu`> [copyright holder]
- Shireen Al-Adeimi <`aladeimi@msu.edu`> [data contributor]

`repair_spelling` *Replace all words in dictionary with alternates*

Description

Given text as a list of character strings, and a dictionary as a two-column dataframe with the first column being misspelled words and the second being corect spelling, swap all misspelled words with the correct spellings.

Usage

```
repair_spelling(text, dictionary, to_words = NULL)
```

Arguments

`text`	Character vector
`dictionary`	Either a dataframe with two columns of text, or a character vector of words.
`to_words`	If dictionary is a list of words, this is list of corresponding words.

Value

Character vector as same length as text. The revised version of text.

Examples

```
## Example 1: Basic Word Replacement
# Texts to be repaired
txt = "This function replaces alll wordss in dictionary with alternates"

# Repair spelling errors by replacing specified words with their correct forms
txt_rep = repair_spelling( txt,
                           c( "alll", "wordss" ),
                           c( "all", "word" ))
txt_rep

## Example 2: Spelling Correction on a Data Frame Column
data( "toy_reads" )

# Repair spelling errors in the text column of the dataframe

toy_reads$text_rep = repair_spelling( toy_reads$text,
                                      c( "No", "the", "what","My" ),
                                      c( "NO.", "THE", "WHAT", "MY" ) )

# View the original and repaired texts
view(data.frame(toy_reads$text, toy_reads$text_rep))
```

`results.tab` *Make results table for grid CCS run*

Description

Make results table for grid CCS run

Usage

```
results.tab(result, corp, Z)
```

Arguments

`result`	a textreg.result() object
`corp`	a corpus or character vector to calculate term frequencies across
`Z`	an indicator for treatment assignment
`clusterID`	optional vector of cluster ID's
`...`	additional arguments passed to textreg().

Value

a textreg.result() object.

---

run_ccs	*Perform Concise Comparative Summarization across a grid of tuning parameters*

Description

Wrapper for textreg::textreg().

Determine the penalty C that will zero out the textreg model for a series of randomly permuted labelings with random assignment dictated by a blocked and cluster-randomized experiment.

Usage

```
run_ccs(x, Z, clusterID = NULL)

## S3 method for class 'threshold.C'
cluster(
  x,
  z,
  design = c("crd", "multi", "cluster", "rcbd"),
  clusterID = NULL,
  siteID = NULL,
  R,
  ...
)
```

Arguments

x	a corpus, character vector of text documents, or set of text features.
Z	an indicator for treatment assignment
clusterID	vector of cluster ID's
design	Type of design used for random assignment (complete randomization, multisite randomized, cluster randomized, and blocked and cluster randomized).
siteID	vector of block ID's
R	Number of times to scramble treatment assignment labels
...	additional arguments passed to textreg().

Details

Method repeatedly generates +1/-1 vectors within the given blocking structure with blocks of +1/-1 within the clustering vector, and then finds a threshold C for each permutation.

Value

a textreg.result() object.

List of numbers. First is the threshold C for the passed labeling. Remainder are the reference distribution based on the permutations.

`simple_RCT_analysis` *Default analysis_function for impacts_on_features*

Description

This is the default analysis function to use in impacts_on_features.

Usage

```
simple_RCT_analysis(feature, formula, data, ...)
```

Arguments

`feature`	Vector of values for the feature of interest.
`formula`	Formula to use in the analysis of form ~ Z + X1, where the first argument has to be the treatment indicator.
`data`	Dataframe with the feature and other variables.
`...`	Additional arguments to pass to lm_robust.

See Also

impacts_on_features

`student_essays` *MORE study science essays*

Description

Dataset consisting of a sample of 40 student essays and corresponding human-coded quality scores (sampled from MORE study pilot data)

Usage

```
student_essays
```

Format

A data.frame with 40 observations of 4 variables:

studentid Student identification number

text Student generated essay text

treatment MORE intervention treatment indicator (0=Control, 1=Treatment)

score Human-coded writing quality score

`textfx`	*Given text from a randomized trial with a binary treatment, this function computes estimates for the average treatment effect with respect to an array of text-based outcomes*

Description

Given text from a randomized trial with a binary treatment, this function computes estimates for the average treatment effect with respect to an array of text-based outcomes

Usage

```
textfx(
  x,
  Z,
  adj = NULL,
  data,
  wts = NULL,
  design = list(siteID = NULL, clusterID = NULL)
)
```

Arguments

`x`	A character vector of text documents or a feature matrix
`Z`	Indicator for treatment assignment.
`adj`	(optional) character vector or named list of variables in the data matrix to adjust for when estimating treatment impacts.
`data`	A `data.frame` of subject-level identifiers, demographic variables, group membership, and/or other pre-treatment covariates.
`wts`	Sampling weights for documents. Assumed uniform if null.
`design`	For multi-site and cluster randomized experiments, a named list of vectors containing site IDs and/or cluster IDs.

Value

A model object for estimating treatment impact across an array of features.

`textfx_terms`	*Estimates the average treatment effect on the frequency and prevalence of a list of specified terms and phrases.*

Description

Estimates the average treatment effect on the frequency and prevalence of a list of specified terms and phrases.

Usage

```
textfx_terms(x, Z, terms, ...)
```

Arguments

`x`	A character vector of text documents or a feature matrix
`Z`	Indicator for treatment assignment.
`terms`	Terms and phrases to evaluate
`...`	optional parameters passed to quanteda::tokens()

Value

A vector showing the frequency and prevalence of the specified terms within each treatment group and results of the hypothesis test comparing prevalence across groups.

`textML` *Model-assisted impact analysis through hybrid human/machine text scoring*

Description

A wrapper function for the multiple steps of generating features, training a scoring model on the human-coded data, predicting scores, and comparing human v. machine estimates.

Usage

```
textML(
  x,
  y,
  z = NULL,
  wts = NULL,
  design = c("crd", "multi", "cluster", "rcbd"),
  siteID = NULL,
  clusterID = NULL,
  max.features = NULL,
  ...
)
```

Arguments

`x`	a corpus or character vector of text documents.
`y`	a vector of human-coded scores. Set elements to 'NA' for documents not previously scored.
`z`	optional indicator for treatment assignment. If specified, separate ensembles will be trained for each treatment group;
`wts`	Sampling weights for which documents were human scored. Assumed uniform if null.
`design`	Type of design used for random assignment (complete randomization, multisite randomized, cluster randomized, and blocked and cluster randomized).
`siteID`	Vector of IDs for site, for multi-site randomized experiments.
`clusterID`	Vector of IDs for cluster, for cluster-randomized experiments.
`max.features`	maximum number of text features to use for model training. Defaults to 'NULL' (no strict limit)
`...`	additional arguments passed to train.

Details

This function takes in a corpus of text documents (or a set of computed text features) along with a sample of human-coded outcome values, and trains an ensemble of machine learning models to predict the outcome as a function of the machine measures of text.

Value

a `textML` model object

`textsamp` *Select a random sample of documents*

Description

Functions to select random samples of documents using different sampling schemes and/or along different design criteria.

Proportions will be calculated as round(size * n_k / n), where n is the number of documents total.

(In particular, this method will sample $round(size / K)$ documents from each cluster, where K is the total number of clusters.) - sounds like equal_size_samples within stratified sampling - n

Usage

```
textsamp(
  x,
  size = NULL,
  prob = NULL,
  wt.fn = NULL,
  scheme = NULL,
  sampling = NULL,
  sampling_control = NULL,
  method = c("srswor", "srswr", "systematic", "poisson"),
  return.data = TRUE,
  ...
)

textsamp_strata(x, size, by = NULL, equal_size_samples = FALSE, ...)

textsamp_cluster(x, by = NULL, n_clusters, cluster_size = NULL, ...)
```

Arguments

`x`	A corpus object or character vector of text documents or dataframe.
`size`	The number of documents to sample across all clusters. If NULL, all documents will be sampled from each selected cluster.
`prob`	a vector of probability weights for each document.
`wt.fn`	a function for generating probability weights; ignored when `prob` is used. See Details.
`scheme`	optional sampling scheme to implement. NOT YET IMPLEMENTED.
`sampling`	the type of sampling to perform. Options include 'strata' and 'cluster'.

`sampling_control`	A list containing control parameters for the specified sampling method (see details)
`method`	the following methods are implemented: simple random sampling without replacement (‘srswor‘), simple random sampling with replacement (‘srswr‘), Poisson sampling (‘poisson‘), systematic sampling (‘systematic‘); if `method` is missing, the default method is `srswor`.
`return.data`	logical; if `TRUE`, the function returns the subset of x that are sampled. FALSE returns a vector of row numbers corresponding to the sampled documents.
`...`	additional arguments passed on to ‘textsamp‘. Cannot include ‘scheme‘.
`by`	a `data.frame` with document-level grouping variable(s) or character vector with names of variables in ‘docvars(x)‘
`equal_size_samples`	TRUE means sample same number of documents from each strata. FALSE means sample proportional to strata size, as described above.
`n_clusters`	The number of clusters to sample. If not provided, an error will be thrown.
`cluster_size`	The number of documents to sample from each selected cluster. If NULL, all documents within each cluster will be sampled.
`n_cluster`	the number of clusters to sample from data

Details

Select a random sample of documents

Functions to select random samples of documents using different sampling schemes and/or along different design criteria.

For stratified sampling (‘sampling = "strata"‘), ‘sampling_control‘ should be a list with: * ‘by‘: A character vector of column names in ‘x‘ to define the strata. * ‘equal_size_samples‘: Logical. If TRUE, sample equal numbers from each stratum.

For cluster sampling (‘sampling = "cluster"‘), ‘sampling_control‘ should be a list with: * ‘by‘: A character vector of column names in ‘x‘ to define the clusters. * ‘n_clusters‘: The number of clusters to sample. * ‘cluster_size‘: (Optional) The number of documents to sample from each cluster.

If multiple clustering variables are passed, this will make clusters as all the _unique_ combinations of these variables. E.g., if A has values of 1, 2, 3 and B has values of 1, 2, then there could be up to six clusters.

Value

Returns either the sampled data or a vector of rownumbers of sampled documents.

Examples

```
# Load example dataframe
data("toy_reads")

## Example 1:Sample 4 documents using the default method
##           (simple random sampling without replacement)
textamp_df = textsamp(toy_reads, size = 4)

## Example 2: Sample 8 documents using Poisson sampling
```

```
textamp_df2 = textsamp(toy_reads, size = 8, method = "poisson")

## Example 3: Sample 8 documents using systematic sampling,
##            but only return the row numbers
textamp_df3 = textsamp(toy_reads, size = 8,
                       method = "systematic", return.data = FALSE)

# Load example dataframe
data("toy_reads")

## Example 1: Stratified Sampling by a Single Variable
stratified_df1 <- textsamp_strata(toy_reads, size = 6, by = "more")

Example 2: Stratified Sampling by Multiple Variables
stratified_df2 <- textsamp_strata(toy_reads, size = 10,
                                  by = c("Q1", "more"))

Example 3: Stratified Sampling with Unequal Sample Sizes per Stratum
stratified_df3 <- textsamp_strata(toy_reads, size = 10,
                                  by = "more", equal_size_samples = FALSE)
```

`toy_reads` *Dataset with 20 essays from READS pilot data*

Description

Used for testing and illustation of rcttext functions.

Usage

```
toy_reads
```

Format

A data.frame with 5 columns and 20 rows

Details

5 column data.frame, ID is the id of subject, Q1, Q2, more are meta information on scoring, and text contains character string of the text of the essay.

`train_ensemble` *Train an ensemble learner for semi-supervised text scoring*

Description

This function takes in a corpus of text documents or a set of computed text features, along with a sample of human-coded outcome values and trains an ensemble of machine learning models to predict the outcome as a function of machine measures of text.

Usage

```
train_ensemble(
  X,
  Z = NULL,
  Yobs,
  coded,
  n.tune = 3,
  bounds = NULL,
  preProc = "zv",
  model = "rf",
  seeds = NA,
  ...
)
```

Arguments

`X`	a `data.frame` or matrix of numeric text features.
`Z`	optional indicator for treatment assignment. If specified, separate ensembles will be trained for each treatment group;
`Yobs`	a vector of human-coded scores for the outcome of interest.
`coded`	A binary vector indicating which documents have been human coded.
`n.tune`	an integer denoting the amount of granularity in the tuning parameter grid. By default, this argument is the number of levels for each tuning parameters that should be generated by train.
`bounds`	a vector (y1, y2) specifying the lower and upper limits for prediction
`...`	additional arguments passed to trainControl.
`cvf`	number of folds for cross validation
`return.all`	should all component models be returned? If 'FALSE', returns only the fitted ensemble(s).

Value

a fitted model object

`train_models` *Train machine learners on a single (unlabeled) dataset.*

Description

Train a bunch of machine learners, along with an ensemble made of those learners, and return the list of fit models.

Usage

```
train_models(
  x,
  y,
  n.tune = 3,
  preProc = NULL,
  bounds = NULL,
  methods = "small",
  include_BART = TRUE,
  include_ensemble = TRUE,
  verbose = TRUE
)
```

Arguments

`x`	A matrix of features
`y`	A vector of outcomes
`n.tune`	The number of tuning parameters to search over
`preProc`	A pre-processing list
`bounds`	The bounds of the outcome variable
`methods`	A character vector of methods to use. Special values of "small" will use a small, hand-picked list of defaults. "full" will use a larger list of defaults.
`include_BART`	Should we include BART in the ensemble?
`include_ensemble`	Should we include the ensemble in the list of models returned? If TRUE, will return vanilla list, not a caretList, of models.
`verbose`	Should we print out progress?

Value

A list of caret 'train' objects (or ensamble equivilents) that can be used to predict on new data.

Index