Package ‘bartMachine’

November 10, 2013

Type Package

Title Bayesian Additive Regression Trees

Version 1.0

Date 2013-07-16

Author Adam Kapelner and Justin Bleich

Maintainer Adam Kapelner <kapelner@wharton.upenn.edu>

Description An advanced implementation of Bayesian Additive Regression Trees (BART, Chipman et al, 2010)

License GPL2

Depends R (>= 2.15.0), rJava (>= 0.8-4), car, randomForest

SystemRequirements Java (>= 1.7.0)

R topics documented:

`bart_machine_get_posterior` *Get Posterior*

Usage

```
bart_machine_get_posterior(bart_machine, new_data)
```

Arguments

`bart_machine`

`new_data`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, new_data)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (class(new_data) != "matrix" && class(new_data) != "data.frame") {
      stop("X needs to be a matrix or data frame with the same column names as the training data.")
    }
    if (!bart_machine$use_missing_data) {
        nrow_before = nrow(new_data)
        new_data = na.omit(new_data)
        if (nrow_before > nrow(new_data)) {
            cat(nrow_before - nrow(new_data), "rows omitted due to missing data\n")
        }
    }
    if (nrow(new_data) == 0) {
        stop("No rows to predict.\n")
    }
    java_bart_machine = bart_machine$java_bart_machine
```

```
    num_iterations_after_burn_in = bart_machine$num_iterations_after_burn_in
    n = nrow(new_data)
    new_data = pre_process_new_data(new_data, bart_machine)
    if (!bart_machine$use_missing_data) {
        M = matrix(0, nrow = nrow(new_data), ncol = ncol(new_data))
        for (i in 1:nrow(new_data)) {
            for (j in 1:ncol(new_data)) {
                if (is.missing(new_data[i, j])) {
                  M[i, j] = 1
                }
            }
        }
        if (sum(M) > 0) {
          cat("WARNING: missing data found in test data and BART was not built with missing data feature!\n")
        }
    }
    y_hat_posterior_samples = t(sapply(.jcall(bart_machine$java_bart_machine,
        "[[D", "getGibbsSamplesForPrediction", .jarray(new_data,
            dispatch = TRUE), as.integer(BART_NUM_CORES)), .jevalArray))
    y_hat = rowMeans(y_hat_posterior_samples)
    list(y_hat = y_hat, X = new_data, y_hat_posterior_samples = y_hat_posterior_samples)
  }
```

---

bart_machine_num_cores

Return number of cores being used

Usage

```
bart_machine_num_cores()
```

Details

Returns the number of cores currently being used by parallelized BART functions

Value

BART_NUM_CORES	number of cores currently being used by parallelized BART functions

Author(s)

Adam Kapelner and Justin Bleich

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function ()
{
    BART_NUM_CORES
  }
```

bart_predict_for_test_data

Predict for test data

Usage

```
bart_predict_for_test_data(bart_machine, Xtest, ytest)
```

Arguments

bart_machine

Xtest

ytest

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, Xtest, ytest)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    ytest_hat = predict(bart_machine, Xtest)
    if (bart_machine$pred_type == "regression") {
        n = nrow(Xtest)
        L2_err = sum((ytest - ytest_hat)^2)
        list(y_hat = ytest_hat, L1_err = sum(abs(ytest - ytest_hat)),
            L2_err = L2_err, rmse = sqrt(L2_err/n), e = ytest -
                ytest_hat)
    }
    else {
        confusion_matrix = as.data.frame(matrix(NA, nrow = 3,
            ncol = 3))
        rownames(confusion_matrix) = c(paste("actual", bart_machine$y_levels),
            "use errors")
        colnames(confusion_matrix) = c(paste("predicted", bart_machine$y_levels),
            "model errors")
        confusion_matrix[1:2, 1:2] = as.integer(table(ytest,
            ytest_hat))
        confusion_matrix[3, 1] = round(confusion_matrix[2, 1]/(confusion_matrix[1,
            1] + confusion_matrix[2, 1]), 3)
        confusion_matrix[3, 2] = round(confusion_matrix[1, 2]/(confusion_matrix[1,
            2] + confusion_matrix[2, 2]), 3)
        confusion_matrix[1, 3] = round(confusion_matrix[1, 2]/(confusion_matrix[1,
            1] + confusion_matrix[1, 2]), 3)
        confusion_matrix[2, 3] = round(confusion_matrix[2, 1]/(confusion_matrix[2,
            1] + confusion_matrix[2, 2]), 3)
        confusion_matrix[3, 3] = round((confusion_matrix[1, 2] +
            confusion_matrix[2, 1])/sum(confusion_matrix[1:2,
```

```
            1:2]), 3)
        list(y_hat = ytest_hat, confusion_matrix = confusion_matrix)
    }
  }
```

build_bart_machine *Build a BART Model*

Description

Builds a BART model for regression or classification

Usage

```
build_bart_machine(X = NULL, y = NULL, Xy = NULL, num_trees = 50, num_burn_in = 250, num_iterations
```

Arguments

- `X` — Dataframe or matrix of predictors. Factors are automatically converted to dummies interally.
- `y` — Vector of response variable. If `y` is `numeric` or `integer`, a BART model for regression is built. If `y` is a factor with two levels, a BART model for classification is built.
- `Xy` — A dataframe or matrix of predictors and the response. The response column must be named "y".
- `num_trees` — The number of trees to be grown in the sum-of-trees model.
- `num_burn_in` — Number of MCMC samples to be discarded as "burn-in".
- `num_iterations_after_burn_in` — Number of MCMC samples to draw from the posterior distribution of $\hat{f}(x)$.
- `alpha` — Base hyperparameter in tree prior for whether a node is nonterminal or not.
- `beta` — Power hyperparameter in tree prior for whether a node is nonterminal or not.
- `k` — For regression, `k` determines the prior probability that $E(Y|X)$ is contained in the interval (y_{min}, y_{max}), based on a normal distribution. For example, when $k = 2$, the prior probability is 95%. For classification, `k` determines the prior probability that $E(Y|X)$ is between $(-3, 3)$. Note that a larger value of `k` results in more shrinkage and a more conservative fit.
- `q` — Quantile of the prior on the error variance at which the data-based estimate is placed. Note that the larger the value of `q`, the more aggressive the fit as you are placing more prior weight on values lower than the data-based estimate. Not used for classification.
- `nu` — Degrees of freedom for the inverse $\chi - squared$ prior. Not used for classification.
- `prob_rule_class` — Threshold for classification. Any observation with a conditional probability greater than `prob_class_rule` is assigned the "positive" outcome. Note that the first level of the response is treated as the "negative" outcome and the second is treated as the "positive" outcome.
- `mh_prob_steps` — Vector of prior probabilities for proposing changes to the tree structures: (GROW, PRUNE, CHANGE)

debug_log
: If TRUE, additional information about the model construction are printed to a file in the working directory.

run_in_sample
: If TRUE, in-sample statistics such as $\hat{f}(x)$, Pseudo-R^2, and RMSE are computed. Setting this to FALSE when not needed can decrease computation time.

s_sq_y
: If "mse", a data-based estimated of the error variance is computed as the MSE from ordinary least squares regression. If "var"., the data-based estimate is computed as the variance of the response. Not used in classification.

cov_prior_vec
: Vector assigning relative weights to how often a particular variable should be proposed as a candidate for a split. The vector is internally normalized so that the weights sum to 1. Note that the length of this vector must equal the length of the design matrix after dummification and augmentation of indicators of missingness (if used). See Bleich et al. (2013) for more details on when to use this feature.

use_missing_data
: If TRUE, the missing data feature is used to automatically handle missing data without imputation. See Kapelner and Bleich (2013) for details.

covariates_to_permute
: ??

num_rand_samps_in_library
: Before building a BART model, samples from the Standard Normal and $\chi - squared(\nu)$ are drawn to be used in the MCMC steps. This parameter determines the number of samples to be taken.

use_missing_data_dummies_as_covars
: If TRUE, additional indicator variables for whether or not an observation in a particular column is missing are included. See Kapelner and Bleich (2013) for details.

replace_missing_data_with_x_j_bar
: If TRUE,missing entries in X are imputed with average value or modal category.

impute_missingness_with_rf_impute
: If TRUE, missing entries are filled in using the rf.impute() function from the randomForest library.

impute_missingness_with_x_j_bar_for_lm
: If TRUE, when computing the data-based estimate of σ^2, missing entries are imputed with average value or modal category.

mem_cache_for_speed
: ??

verbose
: Prints information about progress of the algorithm to the screen.

Details

Returns an object of class "bart_machine". Note that this object persists in the Java heap until codedestroy_bart_machine is called or the R session is terminated.

Note

This function is parallelized and each core will create an independent MCMC chain of size num_burn_in + num_iterations_after_burn_in/bart_machine_num_cores.

Author(s)

Adam Kapelner and Justin Bleich

See Also

link{destroy_bart_machine} link{bart_machine_cv}

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.
```

---

build_bart_machine_cv *Build BART-CV*

Usage

```
build_bart_machine_cv(X = NULL, y = NULL, Xy = NULL, num_tree_cvs = c(50, 200), k_cvs = c(2, 3, 5),
```

Arguments

X

y

Xy

num_tree_cvs

k_cvs

nu_q_cvs

k_folds

...

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X = NULL, y = NULL, Xy = NULL, num_tree_cvs = c(50,
    200), k_cvs = c(2, 3, 5), nu_q_cvs = list(c(3, 0.9), c(3,
    0.99), c(10, 0.75)), k_folds = 5, ...)
{
    y_levels = levels(y)
    if (class(y) == "numeric" || class(y) == "integer") {
        pred_type = "regression"
    }
    else if (class(y) == "factor" & length(y_levels) == 2) {
        pred_type = "classification"
    }
    if (pred_type == "classification") {
        nu_q_cvs = list(c(3, 0.9))
    }
    if ((is.null(X) && is.null(Xy)) || is.null(y) && is.null(Xy)) {
```

```
        stop("You need to give BART a training set either by specifying X and y or by specifying a matrix Xy whi
    }
    else if (is.null(X) && is.null(y)) {
        y = Xy$y
        Xy$y = NULL
        X = Xy
    }
    min_rmse_num_tree = NULL
    min_rmse_k = NULL
    min_rmse_nu_q = NULL
    min_oos_rmse = Inf
    min_oos_misclassification_error = Inf
    for (k in k_cvs) {
        for (nu_q in nu_q_cvs) {
            for (num_trees in num_tree_cvs) {
                cat(paste("  BART CV try: k:", k, "nu, q:", paste(as.numeric(nu_q),
                  collapse = ", "), "m:", num_trees, "\n"))
                k_fold_results = k_fold_cv(X, y, k_folds = k_folds,
                  num_trees = num_trees, k = k, nu = nu_q[1],
                  q = nu_q[2], ...)
                if (pred_type == "regression" && k_fold_results$rmse <
                  min_oos_rmse) {
                  min_oos_rmse = k_fold_results$rmse
                  min_rmse_k = k
                  min_rmse_nu_q = nu_q
                  min_rmse_num_tree = num_trees
                }
             else if (pred_type == "classification" && k_fold_results$misclassification_error <
                  min_oos_misclassification_error) {
                min_oos_misclassification_error = k_fold_results$misclassification_error
                  min_rmse_k = k
                  min_rmse_nu_q = nu_q
                  min_rmse_num_tree = num_trees
                }
            }
        }
    }
   cat(paste("  BART CV win: k:", min_rmse_k, "nu, q:", paste(as.numeric(min_rmse_nu_q),
        collapse = ", "), "m:", min_rmse_num_tree, "\n"))
    build_bart_machine(X, y, num_trees = min_rmse_num_tree, k = min_rmse_k,
        nu = min_rmse_nu_q[1], q = min_rmse_nu_q[2], ...)
}
```

`calc_credible_intervals` *Calculate Credible Intervals*

Usage

```
calc_credible_intervals(bart_machine, new_data, ci_conf = 0.95)
```

Arguments

`bart_machine`

new_data

ci_conf

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, new_data, ci_conf = 0.95)
{
    new_data = pre_process_new_data(new_data, bart_machine)
    n_test = nrow(new_data)
    ci_lower_bd = array(NA, n_test)
    ci_upper_bd = array(NA, n_test)
    y_hat_posterior_samples = t(sapply(.jcall(bart_machine$java_bart_machine,
        "[[D", "getGibbsSamplesForPrediction", .jarray(new_data,
            dispatch = TRUE), as.integer(BART_NUM_CORES)), .jevalArray))
    y_hat = rowMeans(y_hat_posterior_samples)
    for (i in 1:n_test) {
        ci_lower_bd[i] = quantile(sort(y_hat_posterior_samples[i,
            ]), (1 - ci_conf)/2)
        ci_upper_bd[i] = quantile(sort(y_hat_posterior_samples[i,
            ]), (1 + ci_conf)/2)
    }
    cbind(ci_lower_bd, ci_upper_bd)
  }
```

---

calc_prediction_intervals

Calculate Prediction Intervals

Usage

```
calc_prediction_intervals(bart_machine, new_data, pi_conf = 0.95, normal_samples_per_gibbs_sample
```

Arguments

bart_machine

new_data

pi_conf

normal_samples_per_gibbs_sample

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, new_data, pi_conf = 0.95, normal_samples_per_gibbs_sample = 100)
```

```
{
    new_data = pre_process_new_data(new_data, bart_machine)
    n_test = nrow(new_data)
    pi_lower_bd = array(NA, n_test)
    pi_upper_bd = array(NA, n_test)
    y_hat_posterior_samples = t(sapply(.jcall(bart_machine$java_bart_machine,
        "[[D", "getGibbsSamplesForPrediction", .jarray(new_data,
            dispatch = TRUE), as.integer(BART_NUM_CORES)), .jevalArray))
    sigsqs = .jcall(bart_machine$java_bart_machine, "[D", "getGibbsSamplesSigsqs")
   all_prediction_samples = array(NA, c(n_test, bart_machine$num_iterations_after_burn_in,
        normal_samples_per_gibbs_sample))
    for (i in 1:n_test) {
        for (n_g in 1:bart_machine$num_iterations_after_burn_in) {
            y_hat_draw = y_hat_posterior_samples[i, n_g]
            sigsq_draw = sigsqs[n_g]
           all_prediction_samples[i, n_g, ] = rnorm(n = normal_samples_per_gibbs_sample,
                mean = y_hat_draw, sd = sqrt(sigsq_draw))
        }
    }
    for (i in 1:n_test) {
        pi_lower_bd[i] = quantile(c(all_prediction_samples[i,
            , ]), (1 - pi_conf)/2)
        pi_upper_bd[i] = quantile(c(all_prediction_samples[i,
            , ]), (1 + pi_conf)/2)
    }
    cbind(pi_lower_bd, pi_upper_bd)
  }
```

---

check_bart_error_assumptions

Check BART Error Assumptions

Usage

```
check_bart_error_assumptions(bart_machine, alpha_normal_test = 0.05, alpha_hetero_test = 0.05, he
```

Arguments

bart_machine

alpha_normal_test

alpha_hetero_test

hetero_plot

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, alpha_normal_test = 0.05, alpha_hetero_test = 0.05,
```

```
    hetero_plot = "yhats")
{
    if (!(hetero_plot %in% c("ys", "yhats"))) {
        stop("You must specify the parameter \"hetero_plot\" as \"ys\" or \"yhats\"")
    }
    if (bart_machine$pred_type == "classification") {
        stop("There are no convergence diagnostics for classification.")
    }
    graphics.off()
    par(mfrow = c(2, 1))
    es = bart_machine$residuals
    y_hat = bart_machine$y_hat
    normal_p_val = shapiro.test(es)$p.value
    qqp(es, col = "blue", main = paste("Assessment of Normality\n",
        "p-val for shapiro-wilk test of normality of residuals:",
      round(normal_p_val, 3)), xlab = "Normal Q-Q plot for in-sample residuals\n(Theoretical Quantiles)")
    if (hetero_plot == "yhats") {
      plot(y_hat, es, main = paste("Assessment of Heteroskedasticity\nFitted vs residuals"),
           xlab = "Fitted Values", ylab = "Residuals", col = "blue")
    }
    else if (hetero_plot == "ys") {
      plot(bart_machine$y, es, main = paste("Assessment of Heteroskedasticity\nFitted vs residuals"),
           xlab = "Actual Values", ylab = "Residuals", col = "blue")
    }
    abline(h = 0, col = "black")
    par(mfrow = c(1, 1))
  }
```

---

cov_importance_test *Covariance Importance Test*

Usage

```
cov_importance_test(bart_machine, covariates = NULL, num_permutations = 100, num_trees = NULL, plo
```

Arguments

bart_machine

covariates

num_permutations

num_trees

plot

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, covariates = NULL, num_permutations = 100,
    num_trees = NULL, plot = TRUE)
```

```
{
    if (is.null(covariates)) {
        title = "BART omnibus test for covariate importance\n"
    }
    else if (length(covariates) <= 3) {
        if (class(covariates[1]) == "numeric") {
          cov_names = paste(bart_machine$training_data_features_with_missing_features[covariates],
                collapse = ", ")
        }
        else {
            cov_names = paste(covariates, collapse = ", ")
        }
        title = paste("BART test for importance of covariate(s):",
            cov_names, "\n")
    }
    else {
        title = paste("BART test for importance of", length(covariates),
            "covariates", "\n")
    }
    cat(title)
    sd_y = sd(bart_machine$y)
    if (is.null(num_trees)) {
        num_trees = bart_machine$num_trees
        observed_error_estimate = ifelse(bart_machine$pred_type ==
            "regression", bart_machine$PseudoRsq, bart_machine$misclassification_error)
    }
    else {
        bart_machine_copy = build_bart_machine(X = bart_machine$X,
            y = bart_machine$y, use_missing_data = bart_machine$use_missing_data,
          use_missing_data_dummies_as_covars = bart_machine$use_missing_data_dummies_as_covars,
            num_trees = num_trees, verbose = FALSE)
        observed_error_estimate = ifelse(bart_machine$pred_type ==
            "regression", bart_machine$PseudoRsq, bart_machine$misclassification_error)
        destroy_bart_machine(bart_machine_copy)
    }
    permutation_samples_of_error = array(NA, num_permutations)
    for (nsim in 1:num_permutations) {
        cat(".")
        if (nsim%%50 == 0) {
            cat("\n")
        }
        if (is.null(covariates)) {
            bart_machine_samp = build_bart_machine(X = bart_machine$X,
             y = sample(bart_machine$y), use_missing_data = bart_machine$use_missing_data,
             use_missing_data_dummies_as_covars = bart_machine$use_missing_data_dummies_as_covars,
                num_trees = num_trees, verbose = FALSE)
        }
        else {
            X_samp = bart_machine$X
            bart_machine_samp = build_bart_machine(X = X_samp,
                y = bart_machine$y, covariates_to_permute = covariates,
                use_missing_data = bart_machine$use_missing_data,
             use_missing_data_dummies_as_covars = bart_machine$use_missing_data_dummies_as_covars,
                num_trees = num_trees, verbose = FALSE)
        }
        permutation_samples_of_error[nsim] = ifelse(bart_machine$pred_type ==
          "regression", bart_machine_samp$PseudoRsq, bart_machine_samp$misclassification_error)
```

```
        destroy_bart_machine(bart_machine_samp)
    }
    cat("\n")
    pval = ifelse(bart_machine$pred_type == "regression", sum(observed_error_estimate <
        permutation_samples_of_error), sum(observed_error_estimate >
        permutation_samples_of_error))/num_permutations
    if (plot) {
        hist(permutation_samples_of_error, xlim = c(min(permutation_samples_of_error,
            0.99 * observed_error_estimate), max(permutation_samples_of_error,
          1.01 * observed_error_estimate)), xlab = paste("permutation samples\n pval = ",
            pval), br = num_permutations/10, main = paste(title,
            "Null Samples of", ifelse(bart_machine$pred_type ==
                "regression", "Pseudo-R^2's", "Misclassification Errors")))
        abline(v = observed_error_estimate, col = "blue", lwd = 3)
    }
    cat("p_val = ", pval, "\n")
    invisible(list(scaled_rmse_perm_samples = permutation_samples_of_error,
        scaled_rmse_obs = observed_error_estimate, pval = pval))
  }
```

destroy_bart_machine *Destroy BART Model*

Description

This function destroys a BART model by setting all Java pointers to null. (ADAM: CORRECT?)

Usage

```
destroy_bart_machine(bart_machine)
```

Arguments

bart_machine

Details

Removing a "bart_machine" object from R does not free heap space from Java. Since BART objects can consume a large amount of RAM, it is important to remove these objects by calling this function if they are no longer needed or many BART objects are being created.

Author(s)

Adam Kapelner and Justin Bleich

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine)
{
```

```
    .jcall(bart_machine$java_bart_machine, "V", "destroy")
    bart_machine$bart_destroyed = TRUE
    .jcall("java/lang/System", "V", "gc")
  }
```

dummify_data — *Dummify Data*

Usage

```
dummify_data(data, ...)
```

Arguments

data

...

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (data, ...)
{
    as.data.frame(pre_process_training_data(data, ...))
  }
```

get_sigsqs — *Get Posterior Error Variance Estimates*

Usage

```
get_sigsqs(bart_machine, after_burn_in = TRUE)
```

Arguments

bart_machine

after_burn_in

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, after_burn_in = TRUE)
{
    if (bart_machine$bart_destroyed) {
```

```
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (bart_machine$pred_type == "classification") {
        stop("There are no sigsq's for classification.")
    }
    sigsqs = .jcall(bart_machine$java_bart_machine, "[D", "getGibbsSamplesSigsqs")
    if (after_burn_in) {
        num_iterations_after_burn_in = bart_machine$num_iterations_after_burn_in
        num_burn_in = bart_machine$num_burn_in
        num_gibbs = bart_machine$num_gibbs
        num_trees = bart_machine$num_trees
        sigsqs[(length(sigsqs) - num_iterations_after_burn_in):length(sigsqs)]
    }
    else {
        sigsqs
    }
  }
```

`get_tree_depths` *Get the Posterior Tree Depths*

Usage

```
get_tree_depths(bart_machine)
```

Arguments

bart_machine

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine)
{
    tree_depths_after_burn_in = NULL
    for (c in 1:bart_machine$num_cores) {
        tree_depths_after_burn_in_core = t(sapply(.jcall(bart_machine$java_bart_machine,
            "[[I", "getDepthsForTreesInGibbsSampAfterBurnIn",
            as.integer(c)), .jevalArray))
        tree_depths_after_burn_in = rbind(tree_depths_after_burn_in,
            tree_depths_after_burn_in_core)
    }
    tree_depths_after_burn_in
  }
```

get_var_counts_over_chain

Get the Variable Inclusion Counts

Usage

```
get_var_counts_over_chain(bart_machine, type = "splits")
```

Arguments

bart_machine

type

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, type = "splits")
{
    C = t(sapply(.jcall(bart_machine$java_bart_machine, "[[I",
        "getCountsForAllAttribute", as.integer(BART_NUM_CORES),
        type), .jevalArray))
    colnames(C) = colnames(bart_machine$model_matrix_training_data)[1:bart_machine$p]
    C
  }
```

get_var_props_over_chain

Get the Variable Inclusion Proportions

Usage

```
get_var_props_over_chain(bart_machine, type = "splits")
```

Arguments

bart_machine

type

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, type = "splits")
{
```

```
    attribute_props = .jcall(bart_machine$java_bart_machine,
        "[D", "getAttributeProps", as.integer(BART_NUM_CORES),
        type)
   names(attribute_props) = colnames(bart_machine$model_matrix_training_data)[1:bart_machine$p]
    attribute_props
  }
```

---

hist_sigsqs *Create a Histogram of the Posterior Error Variance Estimates*

Usage

```
hist_sigsqs(bart_machine, extra_text = NULL, data_title = "data_model", save_plot = FALSE)
```

Arguments

bart_machine

extra_text

data_title

save_plot

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, extra_text = NULL, data_title = "data_model",
    save_plot = FALSE)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (bart_machine$pred_type == "classification") {
        stop("There are no sigsq's for classification.")
    }
    sigsqs = .jcall(bart_machine$java_bart_machine, "[D", "getGibbsSamplesSigsqs")
    num_iterations_after_burn_in = bart_machine$num_iterations_after_burn_in
    num_burn_in = bart_machine$num_burn_in
    num_gibbs = bart_machine$num_gibbs
    num_trees = bart_machine$num_trees
   sigsqs_after_burnin = sigsqs[(length(sigsqs) - num_iterations_after_burn_in):length(sigsqs)]
    assign("sigsqs_after_burnin", sigsqs_after_burnin, .GlobalEnv)
    avg_sigsqs = mean(sigsqs_after_burnin, na.rm = TRUE)
    if (save_plot) {
        save_plot_function(bart_machine, "sigsqs_hist", data_title)
    }
    else {
        dev.new()
    }
    ppi_a = quantile(sigsqs_after_burnin, 0.025)
    ppi_b = quantile(sigsqs_after_burnin, 0.975)
```

```
    hist(sigsqs_after_burnin, br = 100, main = "Histogram of sigsqs after burn-in",
        xlab = paste("sigsq  avg = ", round(avg_sigsqs, 1), ",  95% PPI = [",
            round(ppi_a, 1), ", ", round(ppi_b, 1), "]", sep = ""))
    abline(v = avg_sigsqs, col = "blue")
    abline(v = ppi_a, col = "yellow")
    abline(v = ppi_b, col = "yellow")
    if (save_plot) {
        dev.off()
    }
    invisible(sigsqs_after_burnin)
  }
```

`init_java_for_bart_machine_with_mem_in_mb`

Initialize a JVM with a specified heap size

Usage

```
init_java_for_bart_machine_with_mem_in_mb(bart_max_mem)
```

Arguments

`bart_max_mem`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_max_mem)
{
    jinit_params = paste("-Xmx", bart_max_mem, "m", sep = "")
    .jinit(parameters = jinit_params)
    for (dependency in JAR_DEPENDENCIES) {
        .jaddClassPath(paste(find.package("bartMachine"), "/java/",
            dependency, sep = ""))
    }
  }
```

`interaction_investigator`

Explore Interactions within BART Model

Usage

```
interaction_investigator(bart_machine, plot = TRUE, num_replicates_for_avg = 5, num_trees_bottlen
```

Arguments

bart_machine

plot

num_replicates_for_avg

num_trees_bottleneck

num_var_plot

cut_bottom

bottom_margin

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, plot = TRUE, num_replicates_for_avg = 5,
    num_trees_bottleneck = 20, num_var_plot = 50, cut_bottom = NULL,
    bottom_margin = 10)
{
    interaction_counts = array(NA, c(bart_machine$p, bart_machine$p,
        num_replicates_for_avg))
    for (r in 1:num_replicates_for_avg) {
        if (r == 1 & num_trees_bottleneck == bart_machine$num_trees) {
            interaction_counts[, , r] = sapply(.jcall(bart_machine$java_bart_machine,
                "[[I", "getInteractionCounts", as.integer(BART_NUM_CORES)),
                .jevalArray)
        }
        else {
            bart_machine_dup = bart_machine_duplicate(bart_machine,
                num_trees = num_trees_bottleneck)
           interaction_counts[, , r] = sapply(.jcall(bart_machine_dup$java_bart_machine,
                "[[I", "getInteractionCounts", as.integer(BART_NUM_CORES)),
                .jevalArray)
            destroy_bart_machine(bart_machine_dup)
            cat(".")
            if (r%%40 == 0) {
                cat("\n")
            }
        }
    }
    cat("\n")
    interaction_counts_avg = apply(interaction_counts, 1:2, mean)
    if (bart_machine$use_missing_data == T) {
      rownames(interaction_counts_avg) = bart_machine$training_data_features_with_missing_features
      colnames(interaction_counts_avg) = bart_machine$training_data_features_with_missing_features
    }
    else {
        rownames(interaction_counts_avg) = bart_machine$training_data_features
        colnames(interaction_counts_avg) = bart_machine$training_data_features
    }
    interaction_counts_sd = apply(interaction_counts, 1:2, sd)
```

```
interaction_counts_s_w_test = apply(interaction_counts, 1:2,
    shapiro_wilk_p_val)
avg_counts = array(NA, bart_machine$p * (bart_machine$p -
    1)/2)
sd_counts = array(NA, bart_machine$p * (bart_machine$p -
    1)/2)
iter = 1
for (i in 1:bart_machine$p) {
    for (j in 1:bart_machine$p) {
        if (j <= i) {
            avg_counts[iter] = interaction_counts_avg[i,
              j]
            sd_counts[iter] = interaction_counts_sd[i, j]
            names(avg_counts)[iter] = paste(rownames(interaction_counts_avg)[i],
              "x", rownames(interaction_counts_avg)[j])
            iter = iter + 1
        }
    }
}
num_total_interactions = bart_machine$p * (bart_machine$p +
    1)/2
if (num_var_plot == Inf || num_var_plot > num_total_interactions) {
    num_var_plot = num_total_interactions
}
avg_counts_sorted_indices = sort(avg_counts, decreasing = TRUE,
    index.return = TRUE)$ix
avg_counts = avg_counts[avg_counts_sorted_indices][1:num_var_plot]
sd_counts = sd_counts[avg_counts_sorted_indices][1:num_var_plot]
if (is.null(cut_bottom)) {
    ylim_bottom = 0
}
else {
    ylim_bottom = cut_bottom * min(avg_counts)
}
if (plot) {
    par(mar = c(bottom_margin, 6, 3, 0))
    if (is.na(sd_counts[1])) {
        moe = 0
    }
    else {
        moe = 1.96 * sd_counts/sqrt(num_replicates_for_avg)
    }
    bars = barplot(avg_counts, names.arg = names(avg_counts),
        las = 2, ylab = "Relative Importance", col = "gray",
        ylim = c(ylim_bottom, max(avg_counts + moe)), xpd = FALSE)
    if (!is.na(sd_counts[1])) {
        conf_upper = avg_counts + 1.96 * sd_counts/sqrt(num_replicates_for_avg)
        conf_lower = avg_counts - 1.96 * sd_counts/sqrt(num_replicates_for_avg)
        segments(bars, avg_counts, bars, conf_upper, col = rgb(0.59,
            0.39, 0.39), lwd = 3)
        segments(bars, avg_counts, bars, conf_lower, col = rgb(0.59,
            0.39, 0.39), lwd = 3)
    }
    par(mar = c(5.1, 4.1, 4.1, 2.1))
}
invisible(list(interaction_counts_avg = interaction_counts_avg,
  interaction_counts_sd = interaction_counts_sd, interaction_counts_s_w_test = interaction_counts_s_w_t
```

```
}
```

investigate_var_importance *Explore Variable Inclusion Proportions in BART Model*

Usage

```
investigate_var_importance(bart_machine, type = "splits", plot = TRUE, num_replicates_for_avg = 5,
```

Arguments

bart_machine

type

plot

num_replicates_for_avg

num_trees_bottleneck

num_var_plot

bottom_margin

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, type = "splits", plot = TRUE, num_replicates_for_avg = 5,
    num_trees_bottleneck = 20, num_var_plot = Inf, bottom_margin = 10)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    var_props = array(0, c(num_replicates_for_avg, bart_machine$p))
    for (i in 1:num_replicates_for_avg) {
        if (i == 1 & num_trees_bottleneck == bart_machine$num_trees) {
            var_props[i, ] = get_var_props_over_chain(bart_machine,
                type)
        }
        else {
            bart_machine_dup = build_bart_machine(bart_machine$X,
                bart_machine$y, num_trees = num_trees_bottleneck,
             num_burn_in = bart_machine$num_burn_in, num_iterations_after_burn_in = bart_machine$num_iterati
                cov_prior_vec = bart_machine$cov_prior_vec, run_in_sample = FALSE,
                use_missing_data = bart_machine$use_missing_data,
             use_missing_data_dummies_as_covars = bart_machine$use_missing_data_dummies_as_covars,
                num_rand_samps_in_library = bart_machine$num_rand_samps_in_library,
             replace_missing_data_with_x_j_bar = bart_machine$replace_missing_data_with_x_j_bar,
             impute_missingness_with_rf_impute = bart_machine$impute_missingness_with_rf_impute,
             impute_missingness_with_x_j_bar_for_lm = bart_machine$impute_missingness_with_x_j_bar_for_lm,
```

```
                verbose = FALSE)
            var_props[i, ] = get_var_props_over_chain(bart_machine_dup,
                type)
            destroy_bart_machine(bart_machine_dup)
        }
        cat(".")
    }
    cat("\n")
    avg_var_props = colMeans(var_props)
    names(avg_var_props) = bart_machine$training_data_features_with_missing_features
    sd_var_props = apply(var_props, 2, sd)
    names(sd_var_props) = bart_machine$training_data_features_with_missing_features
    if (num_var_plot == Inf) {
        num_var_plot = bart_machine$p
    }
    avg_var_props_sorted_indices = sort(avg_var_props, decreasing = TRUE,
        index.return = TRUE)$ix
    avg_var_props = avg_var_props[avg_var_props_sorted_indices][1:num_var_plot]
    sd_var_props = sd_var_props[avg_var_props_sorted_indices][1:num_var_plot]
    if (plot) {
        par(mar = c(bottom_margin, 6, 3, 0))
        if (is.na(sd_var_props[1])) {
            moe = 0
        }
        else {
            moe = 1.96 * sd_var_props/sqrt(num_replicates_for_avg)
        }
        bars = barplot(avg_var_props, names.arg = names(avg_var_props),
            las = 2, ylab = "Inclusion Proportion", col = "gray",
            ylim = c(0, max(avg_var_props + moe)))
        conf_upper = avg_var_props + 1.96 * sd_var_props/sqrt(num_replicates_for_avg)
        conf_lower = avg_var_props - 1.96 * sd_var_props/sqrt(num_replicates_for_avg)
        segments(bars, avg_var_props, bars, conf_upper, col = rgb(0.59,
            0.39, 0.39), lwd = 3)
        segments(bars, avg_var_props, bars, conf_lower, col = rgb(0.59,
            0.39, 0.39), lwd = 3)
        par(mar = c(5.1, 4.1, 4.1, 2.1))
    }
    invisible(list(avg_var_props = avg_var_props, sd_var_props = sd_var_props))
  }
```

---

k_fold_cv *Estimate Out-of-sample Error with K-fold Cross validation*

Usage

```
k_fold_cv(X, y, k_folds = 5, ...)
```

Arguments

X

y

k_folds

...

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (X, y, k_folds = 5, ...)
{
    y_levels = levels(y)
    if (class(y) == "numeric" || class(y) == "integer") {
        pred_type = "regression"
    }
    else if (class(y) == "factor" & length(y_levels) == 2) {
        pred_type = "classification"
    }
    n = nrow(X)
    Xpreprocess = pre_process_training_data(X)
    p = ncol(Xpreprocess)
    if (k_folds <= 1 || k_folds > n) {
      stop("The number of folds must be at least 2 and less than or equal to n, use \"Inf\" for leave one out"
    }
    if (k_folds == Inf) {
        k_folds = n
    }
    holdout_size = round(n/k_folds)
    split_points = seq(from = 1, to = n, by = holdout_size)[1:k_folds]
    if (pred_type == "regression") {
        L1_err = 0
        L2_err = 0
    }
    else {
        confusion_matrix = matrix(0, nrow = 3, ncol = 3)
        rownames(confusion_matrix) = c(paste("actual", y_levels),
            "use errors")
        colnames(confusion_matrix) = c(paste("predicted", y_levels),
            "model errors")
    }
    Xy = data.frame(Xpreprocess, y)
    for (k in 1:k_folds) {
        cat(".")
        holdout_index_i = split_points[k]
        holdout_index_f = ifelse(k == k_folds, n, split_points[k +
            1] - 1)
        test_data_k = Xy[holdout_index_i:holdout_index_f, ]
        training_data_k = Xy[-c(holdout_index_i:holdout_index_f),
            ]
        bart_machine_cv = build_bart_machine(training_data_k[,
            1:p], training_data_k[, (p + 1)], run_in_sample = FALSE,
            ...)
        predict_obj = bart_predict_for_test_data(bart_machine_cv,
            test_data_k[, 1:p], test_data_k[, (p + 1)])
        destroy_bart_machine(bart_machine_cv)
        if (pred_type == "regression") {
            L1_err = L1_err + predict_obj$L1_err
            L2_err = L2_err + predict_obj$L2_err
        }
```

```
        else {
            confusion_matrix[1:2, 1:2] = confusion_matrix[1:2,
                1:2] + table(test_data_k$y, predict_obj$y_hat)
        }
    }
    cat("\n")
    if (pred_type == "regression") {
        list(L1_err = L1_err, L2_err = L2_err, rmse = sqrt(L2_err/n),
            PseudoRsq = 1 - L2_err/sum((y - mean(y))^2))
    }
    else {
        confusion_matrix[3, 1] = round(confusion_matrix[2, 1]/(confusion_matrix[1,
            1] + confusion_matrix[2, 1]), 3)
        confusion_matrix[3, 2] = round(confusion_matrix[1, 2]/(confusion_matrix[1,
            2] + confusion_matrix[2, 2]), 3)
        confusion_matrix[1, 3] = round(confusion_matrix[1, 2]/(confusion_matrix[1,
            1] + confusion_matrix[1, 2]), 3)
        confusion_matrix[2, 3] = round(confusion_matrix[2, 1]/(confusion_matrix[2,
            1] + confusion_matrix[2, 2]), 3)
        confusion_matrix[3, 3] = round((confusion_matrix[1, 2] +
            confusion_matrix[2, 1])/sum(confusion_matrix[1:2,
            1:2]), 3)
       list(confusion_matrix = confusion_matrix, misclassification_error = confusion_matrix[3,
            3])
    }
  }
```

pd_plot *Partial Dependence Plot*

Usage

```
pd_plot(bart_machine, j, levs = c(0.05, seq(from = 0.1, to = 0.9, by = 0.1), 0.95), lower_ci = 0.02
```

Arguments

bart_machine

j

levs

lower_ci

upper_ci

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, j, levs = c(0.05, seq(from = 0.1, to = 0.9,
    by = 0.1), 0.95), lower_ci = 0.025, upper_ci = 0.975)
{
    if (bart_machine$bart_destroyed) {
```

```
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (class(j) == "numeric" && (j < 1 || j > bart_machine$p)) {
        stop(paste("You must set j to a number between 1 and p =",
            bart_machine$p))
    }
    else if (class(j) == "character" && !(j %in% bart_machine$training_data_features)) {
      stop("j must be the name of one of the training features (see \"<bart_model>$training_data_features\")
    }
    else if (!(class(j) == "numeric" || class(j) == "character")) {
        stop("j must be a column number or column name")
    }
    x_j = bart_machine$model_matrix_training_data[, j]
    x_j_quants = quantile(x_j, levs)
    bart_predictions_by_quantile = array(NA, c(length(levs),
        bart_machine$n, bart_machine$num_iterations_after_burn_in))
    for (q in 1:length(levs)) {
        x_j_quant = x_j_quants[q]
        test_data = bart_machine$X
        test_data[, j] = rep(x_j_quant, bart_machine$n)
        bart_predictions_by_quantile[q, , ] = bart_machine_get_posterior(bart_machine,
            test_data)$y_hat_posterior_samples
        cat(".")
    }
    cat("\n")
    if (bart_machine$pred_type == "classification") {
        bart_predictions_by_quantile = qnorm(bart_predictions_by_quantile)
    }
    bart_avg_predictions_by_quantile_by_gibbs = array(NA, c(length(levs),
        bart_machine$num_iterations_after_burn_in))
    for (q in 1:length(levs)) {
        for (g in 1:bart_machine$num_iterations_after_burn_in) {
          bart_avg_predictions_by_quantile_by_gibbs[q, g] = mean(bart_predictions_by_quantile[q,
                , g])
        }
    }
    bart_avg_predictions_by_quantile = apply(bart_avg_predictions_by_quantile_by_gibbs,
        1, mean)
    bart_avg_predictions_lower = apply(bart_avg_predictions_by_quantile_by_gibbs,
        1, quantile, probs = lower_ci)
    bart_avg_predictions_upper = apply(bart_avg_predictions_by_quantile_by_gibbs,
        1, quantile, probs = upper_ci)
   var_name = ifelse(class(j) == "character", j, bart_machine$training_data_features[j])
    ylab_name = ifelse(bart_machine$pred_type == "classification",
        "Partial Effect (Probits)", "Partial Effect")
    plot(x_j_quants, bart_avg_predictions_by_quantile, type = "o",
        main = "Partial Dependence Plot", ylim = c(min(bart_avg_predictions_lower,
            bart_avg_predictions_upper), max(bart_avg_predictions_lower,
            bart_avg_predictions_upper)), ylab = ylab_name, xlab = paste(var_name,
            "plotted at specified quantiles"))
    lines(x_j_quants, bart_avg_predictions_lower, type = "o",
        col = "blue")
    lines(x_j_quants, bart_avg_predictions_upper, type = "o",
        col = "blue")
  invisible(list(x_j_quants = x_j_quants, bart_avg_predictions_by_quantile = bart_avg_predictions_by_quant
}
```

plot_convergence_diagnostics
Plot Convergence Diagnostics

Usage

```
plot_convergence_diagnostics(bart_machine)
```

Arguments

bart_machine

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine)
{
    par(mfrow = c(2, 2))
    if (bart_machine$pred_type == "regression") {
        plot_sigsqs_convergence_diagnostics(bart_machine)
    }
    plot_mh_acceptance_reject(bart_machine)
    plot_tree_num_nodes(bart_machine)
    plot_tree_depths(bart_machine)
    par(mfrow = c(1, 1))
  }
```

plot_mh_acceptance_reject
Plot the proportion of Metropolis-Hastings Acceptances

Usage

```
plot_mh_acceptance_reject(bart_machine)
```

Arguments

bart_machine

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine)
```

```
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    mh_acceptance_reject = get_mh_acceptance_reject(bart_machine)
    a_r_before_burn_in = mh_acceptance_reject[["a_r_before_burn_in"]]
   a_r_before_burn_in_avg_over_trees = rowSums(a_r_before_burn_in)/bart_machine$num_trees
    a_r_after_burn_in_avgs_over_trees = list()
    for (c in 1:bart_machine$num_cores) {
        a_r_after_burn_in = mh_acceptance_reject[["a_r_after_burn_in"]][[c]]
      a_r_after_burn_in_avgs_over_trees[[c]] = rowSums(a_r_after_burn_in)/bart_machine$num_trees
    }
    num_after_burn_in_per_core = length(a_r_after_burn_in_avgs_over_trees[[1]])
    num_gibbs_per_core = bart_machine$num_burn_in + num_after_burn_in_per_core
    plot(1:num_gibbs_per_core, rep(0, num_gibbs_per_core), ylim = c(0,
        1), type = "n", main = "Percent Acceptance by Gibbs Sample",
        xlab = "Gibbs Sample", ylab = "% of Trees Accepting")
    abline(v = bart_machine$num_burn_in, col = "grey")
    points(1:bart_machine$num_burn_in, a_r_before_burn_in_avg_over_trees,
        col = "grey")
   tryCatch(lines(loess.smooth(1:bart_machine$num_burn_in, a_r_before_burn_in_avg_over_trees),
        col = "black", lwd = 4), error = function(e) {
        e
    })
    for (c in 1:bart_machine$num_cores) {
        points((bart_machine$num_burn_in + 1):num_gibbs_per_core,
            a_r_after_burn_in_avgs_over_trees[[c]], col = COLORS[c])
        tryCatch(lines(loess.smooth((bart_machine$num_burn_in +
            1):num_gibbs_per_core, a_r_after_burn_in_avgs_over_trees[[c]]),
            col = COLORS[c], lwd = 4), error = function(e) {
            e
        })
    }
  }
```

`plot_sigsqs_convergence_diagnostics` *Plot Error Variance Estimates to Assess Convergence*

Usage

```
plot_sigsqs_convergence_diagnostics(bart_machine)
```

Arguments

`bart_machine`

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
```

```
function (bart_machine)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (bart_machine$pred_type == "classification") {
        stop("There are no convergence diagnostics for classification.")
    }
    sigsqs = get_sigsqs(bart_machine, after_burn_in = FALSE)
    num_iterations_after_burn_in = bart_machine$num_iterations_after_burn_in
    num_burn_in = bart_machine$num_burn_in
    num_gibbs = bart_machine$num_gibbs
    num_trees = bart_machine$num_trees
   sigsqs_after_burnin = sigsqs[(length(sigsqs) - num_iterations_after_burn_in):length(sigsqs)]
    avg_sigsqs_after_burn_in = mean(sigsqs_after_burnin, na.rm = TRUE)
    plot(sigsqs, main = paste("Sigsq Estimates over Gibbs Samples"),
        xlab = "Gibbs sample (yellow lines: after burn-in 95% PPI)",
        ylab = paste("Sigsq by iteration, avg after burn-in =",
            round(avg_sigsqs_after_burn_in, 3)), ylim = c(quantile(sigsqs,
            0.01), quantile(sigsqs, 0.99)), pch = ".", cex = 3,
        col = "gray")
    points(sigsqs, pch = ".", col = "red")
    ppi_sigsqs = quantile(sigsqs[num_burn_in:length(sigsqs)],
        c(0.025, 0.975))
    abline(a = ppi_sigsqs[1], b = 0, col = "yellow")
    abline(a = ppi_sigsqs[2], b = 0, col = "yellow")
    abline(a = avg_sigsqs_after_burn_in, b = 0, col = "blue")
    abline(v = num_burn_in, col = "gray")
    if (bart_machine$num_cores > 1) {
        for (c in 2:bart_machine$num_cores) {
          abline(v = num_burn_in + (c - 1) * bart_machine$num_iterations_after_burn_in/bart_machine$num_cores
                col = "gray")
        }
    }
  }
```

---

plot_tree_depths *Plot Tree Depths*

Usage

```
plot_tree_depths(bart_machine)
```

Arguments

bart_machine

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
```

```
function (bart_machine)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    tree_depths_after_burn_in = get_tree_depths(bart_machine)
    num_after_burn_in_per_core = nrow(tree_depths_after_burn_in)
    plot(1:num_after_burn_in_per_core, rep(0, num_after_burn_in_per_core),
        type = "n", main = "Tree Depth by Gibbs Sample After Burn-in",
        xlab = "Gibbs Sample", ylab = paste("Tree Depth for all cores"),
        ylim = c(0, max(tree_depths_after_burn_in)))
    for (t in 1:ncol(tree_depths_after_burn_in)) {
        lines(1:num_after_burn_in_per_core, tree_depths_after_burn_in[,
            t], col = rgb(0.9, 0.9, 0.9))
    }
    lines(1:num_after_burn_in_per_core, apply(tree_depths_after_burn_in,
        1, mean), col = "blue", lwd = 4)
    lines(1:num_after_burn_in_per_core, apply(tree_depths_after_burn_in,
        1, min), col = "black")
    lines(1:num_after_burn_in_per_core, apply(tree_depths_after_burn_in,
        1, max), col = "black")
    if (bart_machine$num_cores > 1) {
        for (c in 2:bart_machine$num_cores) {
          abline(v = (c - 1) * bart_machine$num_iterations_after_burn_in/bart_machine$num_cores,
                col = "gray")
        }
    }
  }
```

`plot_tree_num_nodes` *Plot the Average Number of Nodes in the Trees*

Usage

```
plot_tree_num_nodes(bart_machine)
```

Arguments

bart_machine

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
   tree_num_nodes_and_leaves_after_burn_in = get_tree_num_nodes_and_leaves(bart_machine)
    num_after_burn_in_per_core = nrow(tree_num_nodes_and_leaves_after_burn_in)
```

```
    plot(1:num_after_burn_in_per_core, rep(0, num_after_burn_in_per_core),
        type = "n", main = "Tree Num Nodes And Leaves by Gibbs Sample After Burn-in",
        xlab = "Gibbs Sample", ylab = paste("Tree Num Nodes and Leaves for all cores"),
        ylim = c(0, max(tree_num_nodes_and_leaves_after_burn_in)))
    for (t in 1:ncol(tree_num_nodes_and_leaves_after_burn_in)) {
        lines(1:num_after_burn_in_per_core, tree_num_nodes_and_leaves_after_burn_in[,
            t], col = rgb(0.9, 0.9, 0.9))
    }
    lines(1:num_after_burn_in_per_core, apply(tree_num_nodes_and_leaves_after_burn_in,
        1, mean), col = "blue", lwd = 4)
    lines(1:num_after_burn_in_per_core, apply(tree_num_nodes_and_leaves_after_burn_in,
        1, min), col = "black")
    lines(1:num_after_burn_in_per_core, apply(tree_num_nodes_and_leaves_after_burn_in,
        1, max), col = "black")
    if (bart_machine$num_cores > 1) {
        for (c in 2:bart_machine$num_cores) {
          abline(v = (c - 1) * bart_machine$num_iterations_after_burn_in/bart_machine$num_cores,
                col = "gray")
        }
    }
  }
```

`plot_y_vs_yhat` *Plot the fitted Versus Actual Response*

Usage

```
plot_y_vs_yhat(bart_machine, X = NULL, y = NULL, credible_intervals = FALSE, prediction_intervals
```

Arguments

bart_machine

X

y

credible_intervals

prediction_intervals

interval_confidence_level

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, X = NULL, y = NULL, credible_intervals = FALSE,
    prediction_intervals = FALSE, interval_confidence_level = 0.95)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
```

```
}
if (credible_intervals && prediction_intervals) {
  stop("You cannot plot both credibility intervals and prediction intervals simultaneously.")
}
if (bart_machine$pred_type == "classification") {
    stop("You cannot plot y vs y_hat for classification.")
}
if (is.null(X) & is.null(y)) {
    X = bart_machine$X
    y = bart_machine$y
    y_hat = bart_machine$y_hat_train
    in_sample = TRUE
}
else {
    predict_obj = bart_predict_for_test_data(bart_machine,
        X, y)
    y_hat = predict_obj$y_hat
    in_sample = FALSE
}
if (credible_intervals) {
    credible_intervals = calc_credible_intervals(bart_machine,
        X, interval_confidence_level)
    ci_a = credible_intervals[, 1]
    ci_b = credible_intervals[, 2]
    y_in_ppi = y >= ci_a & y <= ci_b
    prop_ys_in_ppi = sum(y_in_ppi)/length(y_in_ppi)
    plot(y, y_hat, main = paste(ifelse(in_sample, "In-Sample",
        "Out-of-Sample"), " Fitted vs. Actual Values\nwith ",
        round(interval_confidence_level * 100), "% Cred. Int.'s (",
        round(prop_ys_in_ppi * 100, 2), "% coverage)", sep = ""),
        xlab = paste("Actual Values", sep = ""), ylab = "Fitted Values",
        xlim = c(min(min(y), min(y_hat)), max(max(y), max(y_hat))),
        ylim = c(min(min(y), min(y_hat)), max(max(y), max(y_hat))),
        cex = 0)
    for (i in 1:bart_machine$n) {
        segments(y[i], ci_a[i], y[i], ci_b[i], col = "grey",
            lwd = 0.1)
    }
    for (i in 1:bart_machine$n) {
        points(y[i], y_hat[i], col = ifelse(y_in_ppi[i],
            "darkgreen", "red"), cex = 0.6, pch = 16)
    }
}
else if (prediction_intervals) {
    credible_intervals = calc_prediction_intervals(bart_machine,
        X, interval_confidence_level)
    ci_a = credible_intervals[, 1]
    ci_b = credible_intervals[, 2]
    y_in_ppi = y >= ci_a & y <= ci_b
    prop_ys_in_ppi = sum(y_in_ppi)/length(y_in_ppi)
    plot(y, y_hat, main = paste(ifelse(in_sample, "In-Sample",
        "Out-of-Sample"), " Fitted vs. Actual Values\nwith ",
        round(interval_confidence_level * 100), "% Pred. Int.'s (",
        round(prop_ys_in_ppi * 100, 2), "% coverage)", sep = ""),
        xlab = paste("Actual Values", sep = ""), ylab = "Fitted Values",
        xlim = c(min(min(y), min(y_hat)), max(max(y), max(y_hat))),
        ylim = c(min(min(y), min(y_hat)), max(max(y), max(y_hat))),
```

```
            cex = 0)
        for (i in 1:bart_machine$n) {
            segments(y[i], ci_a[i], y[i], ci_b[i], col = "grey",
                lwd = 0.1)
        }
        for (i in 1:bart_machine$n) {
            points(y[i], y_hat[i], col = ifelse(y_in_ppi[i],
                "darkgreen", "red"), cex = 0.6, pch = 16)
        }
    }
    else {
        plot(y, y_hat, main = "Fitted vs. Actual Values", xlab = "Actual Values",
            ylab = "Fitted Values", col = "blue", xlim = c(min(min(y),
                min(y_hat)), max(max(y), max(y_hat))), ylim = c(min(min(y),
                min(y_hat)), max(max(y), max(y_hat))), )
    }
    abline(a = 0, b = 1, lty = 2)
  }
```

`rmse_by_num_trees` *Plot the Out-of-sample RMSE by Number of Trees*

Usage

```
rmse_by_num_trees(bart_machine, tree_list = c(5, seq(10, 50, 10), 100, 150, 200), in_sample = FALS
```

Arguments

bart_machine

tree_list

in_sample

plot

holdout_pctg

num_replicates

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, tree_list = c(5, seq(10, 50, 10), 100,
    150, 200), in_sample = FALSE, plot = TRUE, holdout_pctg = 0.3,
    num_replicates = 4)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    if (bart_machine$pred_type == "classification") {
        stop("This function does not work for classification.")
    }
```

```
  X = bart_machine$X
  y = bart_machine$y
  n = bart_machine$n
  rmses = array(NA, c(num_replicates, length(tree_list)))
  cat("num_trees = ")
  for (t in 1:length(tree_list)) {
      for (r in 1:num_replicates) {
          if (in_sample) {
              bart_machine_dup = bart_machine_duplicate(bart_machine,
                num_trees = tree_list[t], run_in_sample = TRUE)
              rmses[r, t] = bart_machine_dup$rmse_train
          }
          else {
              holdout_indicies = sample(1:n, holdout_pctg *
                n)
              Xtrain = X[setdiff(1:n, holdout_indicies), ]
              ytrain = y[setdiff(1:n, holdout_indicies)]
              Xtest = X[holdout_indicies, ]
              ytest = y[holdout_indicies]
              bart_machine_dup = bart_machine_duplicate(bart_machine,
                Xtrain, ytrain, num_trees = tree_list[t])
              predict_obj = bart_predict_for_test_data(bart_machine_dup,
                Xtest, ytest)
              rmses[r, t] = predict_obj$rmse
          }
          destroy_bart_machine(bart_machine_dup)
          cat("..")
          cat(tree_list[t])
      }
  }
  cat("\n")
  rmse_means = colMeans(rmses)
  if (plot) {
      rmse_sds = apply(rmses, 2, sd)
      y_mins = rmse_means - 2 * rmse_sds
      y_maxs = rmse_means + 2 * rmse_sds
      plot(tree_list, rmse_means, type = "o", xlab = "Number of Trees",
          ylab = paste(ifelse(in_sample, "In-Sample", "Out-Of-Sample"),
              "RMSE"), main = paste(ifelse(in_sample, "In-Sample",
              "Out-Of-Sample"), "RMSE by Number of Trees"),
          ylim = c(min(y_mins), max(y_maxs)))
      if (num_replicates > 1) {
          for (t in 1:length(tree_list)) {
              lowers = rmse_means[t] - 1.96 * rmse_sds[t]/sqrt(num_replicates)
              uppers = rmse_means[t] + 1.96 * rmse_sds[t]/sqrt(num_replicates)
              segments(tree_list[t], lowers, tree_list[t],
                uppers, col = "grey", lwd = 0.1)
          }
      }
  }
  invisible(rmse_means)
}
```

---

set_bart_machine_num_cores

Set the number of cores for BART

Description

Sets the number of cores to be used for all parallelized BART functions.

Usage

```
set_bart_machine_num_cores(num_cores)
```

Arguments

num_cores	Number of cores to use

Author(s)

Adam Kapelner and Justin Bleich

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (num_cores)
{
    assign("BART_NUM_CORES", num_cores, ".GlobalEnv")
  }
```

---

var_selection_by_permute_response_cv

Perform Variable Selection Using Cross-validation Method

Usage

```
var_selection_by_permute_response_cv(bart_machine, k_folds = 5, num_reps_for_avg = 5, num_permute
```

Arguments

bart_machine

k_folds

num_reps_for_avg

num_permute_samples

num_trees_for_permute

alpha

num_trees_pred_cv

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, k_folds = 5, num_reps_for_avg = 5, num_permute_samples = 100,
    num_trees_for_permute = 20, alpha = 0.05, num_trees_pred_cv = 200)
{
    if (k_folds <= 1 || k_folds > bart_machine$n) {
      stop("The number of folds must be at least 2 and less than or equal to n, use \"Inf\" for leave one out")
    }
    if (k_folds == Inf) {
        k_folds = bart_machine$n
    }
    holdout_size = round(bart_machine$n/k_folds)
    split_points = seq(from = 1, to = bart_machine$n, by = holdout_size)[1:k_folds]
    L2_err_mat = matrix(NA, nrow = k_folds, ncol = 3)
   colnames(L2_err_mat) = c("important_vars_local_names", "important_vars_global_max_names",
        "important_vars_global _se_names")
    for (k in 1:k_folds) {
        cat("cv #", k, "\n", sep = "")
        holdout_index_i = split_points[k]
        holdout_index_f = ifelse(k == k_folds, bart_machine$n,
            split_points[k + 1] - 1)
      training_X_k = bart_machine$model_matrix_training_data[-c(holdout_index_i:holdout_index_f),
            ]
        training_y_k = y[-c(holdout_index_i:holdout_index_f)]
        bart_machine_temp = build_bart_machine(as.data.frame(training_X_k),
            training_y_k, num_trees = bart_machine$num_trees,
          num_burn_in = bart_machine$num_burn_in, cov_prior_vec = bart_machine$cov_prior_vec,
            run_in_sample = FALSE, use_missing_data = bart_machine$use_missing_data,
          use_missing_data_dummies_as_covars = bart_machine$use_missing_data_dummies_as_covars,
            num_rand_samps_in_library = bart_machine$num_rand_samps_in_library,
          replace_missing_data_with_x_j_bar = bart_machine$replace_missing_data_with_x_j_bar,
          impute_missingness_with_rf_impute = bart_machine$impute_missingness_with_rf_impute,
          impute_missingness_with_x_j_bar_for_lm = bart_machine$impute_missingness_with_x_j_bar_for_lm,
            verbose = FALSE)
      bart_variables_select_obj_k = var_selection_by_permute_response_three_methods(bart_machine_temp,
          num_permute_samples = num_permute_samples, num_trees_for_permute = num_trees_for_permute,
            alpha = alpha, plot = FALSE)
        destroy_bart_machine(bart_machine_temp)
      test_X_k = bart_machine$model_matrix_training_data[holdout_index_i:holdout_index_f,
            ]
        text_y_k = y[holdout_index_i:holdout_index_f]
        cat("method")
        for (method in colnames(L2_err_mat)) {
            cat(".")
            vars_selected_by_method = bart_variables_select_obj_k[[method]]
            if (length(vars_selected_by_method) == 0) {
                ybar_est = mean(training_y_k)
                L2_err_mat[k, method] = sum((text_y_k - ybar_est)^2)
            }
            else {
                training_X_k_red_by_vars_picked_by_method = as.data.frame(training_X_k[,
                  vars_selected_by_method])
```

```
            bart_machine_temp = build_bart_machine(training_X_k_red_by_vars_picked_by_method,
                 training_y_k, num_burn_in = bart_machine$num_burn_in,
              num_iterations_after_burn_in = bart_machine$num_iterations_after_burn_in,
                 num_trees = num_trees_pred_cv, run_in_sample = FALSE,
                 verbose = FALSE)
               test_X_k_red_by_vars_picked_by_method = as.data.frame(test_X_k[,
                 bart_variables_select_obj_k[[method]]])
               predict_obj = bart_predict_for_test_data(bart_machine_temp,
                 test_X_k_red_by_vars_picked_by_method, text_y_k)
               destroy_bart_machine(bart_machine_temp)
               L2_err_mat[k, method] = predict_obj$L2_err
           }
      }
      cat("\n")
 }
 L2_err_by_method = colSums(L2_err_mat)
 min_var_selection_method = colnames(L2_err_mat)[which(L2_err_by_method ==
      min(L2_err_by_method))]
 min_var_selection_method = min_var_selection_method[1]
 cat("final", "\n")
bart_variables_select_obj = var_selection_by_permute_response_three_methods(bart_machine,
    num_permute_samples = num_permute_samples, num_trees_for_permute = num_trees_for_permute,
      alpha = alpha, plot = FALSE)
list(best_method = min_var_selection_method, important_vars_cv = sort(bart_variables_select_obj[[min_var
}
```

---

var_selection_by_permute_response_three_methods
Perform Variable Selection

Usage

```
var_selection_by_permute_response_three_methods(bart_machine, num_reps_for_avg = 10, num_permute_
```

Arguments

bart_machine

num_reps_for_avg

num_permute_samples

num_trees_for_permute

alpha

plot

num_var_plot

bottom_margin

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function (bart_machine, num_reps_for_avg = 10, num_permute_samples = 100,
    num_trees_for_permute = 20, alpha = 0.05, plot = TRUE, num_var_plot = Inf,
    bottom_margin = 10)
{
    if (bart_machine$bart_destroyed) {
        stop("This BART machine has been destroyed. Please recreate.")
    }
    permute_mat = matrix(NA, nrow = num_permute_samples, ncol = bart_machine$p)
    colnames(permute_mat) = bart_machine$training_data_features_with_missing_features
    cat("avg")
    var_true_props_avg = get_averaged_true_var_props(bart_machine,
        num_reps_for_avg, num_trees_for_permute)
    var_true_props_avg = sort(var_true_props_avg, decreasing = TRUE)
    cat("null")
    for (b in 1:num_permute_samples) {
        permute_mat[b, ] = get_null_permute_var_importances(bart_machine,
            num_trees_for_permute)
    }
    cat("\n")
    permute_mat = permute_mat[, names(var_true_props_avg)]
    pointwise_cutoffs = apply(permute_mat, 2, quantile, probs = 1 -
        alpha)
    important_vars_pointwise_names = names(var_true_props_avg[var_true_props_avg >
        pointwise_cutoffs & var_true_props_avg > 0])
    important_vars_pointwise_col_nums = sapply(1:length(important_vars_pointwise_names),
        function(x) {
          which(important_vars_pointwise_names[x] == bart_machine$training_data_features_with_missing_featu
        })
    max_cut = quantile(apply(permute_mat, 1, max), 1 - alpha)
    important_vars_simul_max_names = names(var_true_props_avg[var_true_props_avg >=
        max_cut & var_true_props_avg > 0])
    important_vars_simul_max_col_nums = sapply(1:length(important_vars_simul_max_names),
        function(x) {
          which(important_vars_simul_max_names[x] == bart_machine$training_data_features_with_missing_featu
        })
    perm_se = apply(permute_mat, 2, sd)
    perm_mean = apply(permute_mat, 2, mean)
    cover_constant = bisectK(tol = 0.01, coverage = 1 - alpha,
        permute_mat = permute_mat, x_left = 1, x_right = 20,
        countLimit = 100, perm_mean = perm_mean, perm_se = perm_se)
    important_vars_simul_se_names = names(var_true_props_avg[which(var_true_props_avg >=
        perm_mean + cover_constant * perm_se & var_true_props_avg >
        0)])
    important_vars_simul_se_col_nums = sapply(1:length(important_vars_simul_se_names),
        function(x) {
          which(important_vars_simul_se_names[x] == bart_machine$training_data_features_with_missing_featur
        })
    if (plot) {
        par(mar = c(bottom_margin, 6, 3, 0))
        if (num_var_plot == Inf | num_var_plot > bart_machine$p) {
```

```
            num_var_plot = bart_machine$p
        }
        par(mfrow = c(2, 1))
        non_zero_idx = which(var_true_props_avg > 0)[1:min(num_var_plot,
            length(which(var_true_props_avg > 0)))]
        plot_n = length(non_zero_idx)
        if (length(non_zero_idx) < length(var_true_props_avg))
            warning(paste(length(which(var_true_props_avg ==
                0)), "covariates with inclusion proportions of 0 omitted from plots."))
        plot(1:plot_n, var_true_props_avg[non_zero_idx], type = "n",
            xlab = NA, xaxt = "n", ylim = c(0, max(max(var_true_props_avg),
                max_cut * 1.1)), main = "Local Procedure", ylab = "proportion included")
        axis(1, at = 1:plot_n, labels = names(var_true_props_avg[non_zero_idx]),
            las = 2)
        for (j in non_zero_idx) {
            points(j, var_true_props_avg[j], pch = ifelse(var_true_props_avg[j] <=
                quantile(permute_mat[, j], 1 - alpha), 1, 16))
        }
        sapply(non_zero_idx, function(s) {
            segments(s, 0, x1 = s, quantile(permute_mat[, s],
                1 - alpha), col = "forestgreen")
        })
        plot(1:plot_n, var_true_props_avg[non_zero_idx], type = "n",
            xlab = NA, xaxt = "n", ylim = c(0, max(max(var_true_props_avg),
                max_cut * 1.1)), main = "Simul. Max and SE Procedures",
            ylab = "proportion included")
        axis(1, at = 1:plot_n, labels = names(var_true_props_avg[non_zero_idx]),
            las = 2)
        abline(h = max_cut, col = "red")
        for (j in non_zero_idx) {
            points(j, var_true_props_avg[j], pch = ifelse(var_true_props_avg[j] <
                max_cut, ifelse(var_true_props_avg[j] > perm_mean[j] +
                cover_constant * perm_se[j], 8, 1), 16))
        }
        sapply(non_zero_idx, function(s) {
            segments(s, 0, x1 = s, perm_mean[s] + cover_constant *
                perm_se[s], col = "blue")
        })
        par(mar = c(5.1, 4.1, 4.1, 2.1))
        par(mfrow = c(1, 1))
    }
    invisible(list(important_vars_local_names = important_vars_pointwise_names,
        important_vars_global_max_names = important_vars_simul_max_names,
        important_vars_global_se_names = important_vars_simul_se_names,
        important_vars_local_col_nums = as.numeric(important_vars_pointwise_col_nums),
      important_vars_global_max_col_nums = as.numeric(important_vars_simul_max_col_nums),
       important_vars_global_se_col_nums = as.numeric(important_vars_simul_se_col_nums),
        var_true_props_avg = var_true_props_avg, permute_mat = permute_mat))
}
```

Index