Exercise Feature Importance

Libraries

```
library(ggplot2)
```

Exercise 1: Permutation feature importance

1a)

Permutation Feature Importance compares the performance of the model on the original data with the performance on perturbed data, where the dependency of the variable of interest (let's call it x_j) with the target Y variable is broken.

In order to break the dependency of x_j with y, x_j is replaced with a permuted version $\tilde{x}_j$ which is independent of the target.

However, by permuting the variable we do not only break the dependency with y, but also with all other covariates. As such, we may create unrealistic observations.

For example, time of the year may be dependent with the highest temperature on a day. If we resample the variable time of the year independently of the temperature high, we may create observations where time of the year is winter and temperature high is 40 degrees celsius.

1b)

If a feature anyway independent of its covariates ($x_j \perp x_{-j}$, then PFI does not extrapolate to unseen regions).

Intuitively, the reason is that no dependency with the covariates is broken, since there were not dependencies between x_j and the remaining variables x_{-j} to begin with..

We discuss the theoretical background of the issue in more detail in the in-class exercise.

1c)

```
### Read in the data set via the read.csv()-function and remove
### the column that only contains the row indices.
df <- read.csv(file = "extrapolation.csv")
df <- df[ , -which(names(df) == "X")]

### The classical training and testing data split (here 70% training, 30% testing data).
set.seed(100)
train <- sample(nrow(df), 0.7 * nrow(df))
```

```r
training_data <- df[train, ]
test_data <- df[-train, ]

### Fit a linear model using the training data.
model <- lm(y ~ ., data = training_data)

### Assess the MSE on the test data set.
preds <- predict(model, newdata = test_data)
mse <- mean((test_data$y - preds) ^ 2)
print(paste("MSE:", mse))
```

```
## [1] "MSE: 0.00872256546120504"
```

1d)

First, we implement PFI (with mean squared error as performance metric).

```r
### Calculate the PFI score for a single feature given by fname.
pfi_fname <- function(fname, model, X_test, y_test) {
  ### Permute the observations for feature fname.
  X_test_perm <- X_test
  X_test_perm[[fname]] <- sample(X_test_perm[[fname]])

  ### Predict on the original data situation as well as on the permuted one.
  preds_original <- predict(model, X_test)
  preds_perm <- predict(model, X_test_perm)

  ### Calculate the MSE on both data situations.
  mse_original <- mean((y_test - preds_original) ^ 2)
  mse_perm <- mean((y_test - preds_perm) ^ 2)

  ### The PFI score is now defined as the increase in MSE when permuting the feature.
  mse_perm - mse_original
}

### Calculate the PFI score defined via the function
### fi_fname_func for all features in X_test.
fi <- function(fi_fname_func, ...) {
  ### Iterate over all features in X_test and calculate their single feature PFI score.
  unlist(lapply(colnames(X_test), fi_fname_func, ...))
}
```

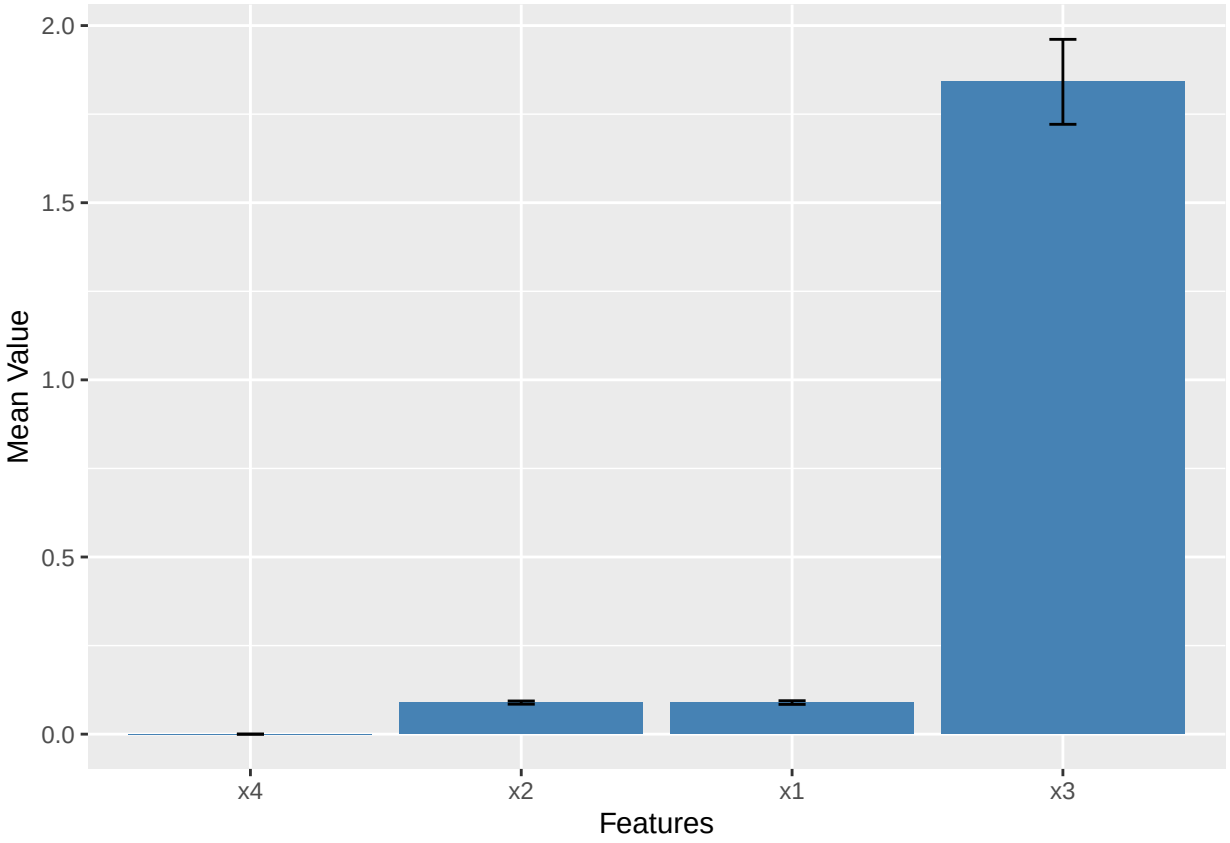
This is the function to run n times for all naive functions. Different arguments will be passed here according to the order of the parameters in each naive function. In total, there are 7 arguments: model, X_test, y_test, X_train, n_marginalize, df, 'y'(y_name).

```r
n_times <- function(func, n, return_raw, ...) {
  ### Apply the function n times.
  ### We need to take the transpose to get the result into the right shape.
  results <- t(sapply(1:n, function(i) func(...)))

  ### Return the mean_fi, the std_fi and if wanted the raw results contained in a list.
```

```
  list(colMeans(results), apply(results, 2, sd), if (return_raw) results)
}
```

Now we apply the method to our model and data set.

```
### Create appropriate data sets to use our implemented functions.
X_train <- training_data[ , -which(names(training_data) == "y")]
X_test <- test_data[ , -which(names(test_data) == "y")]
y_train <- training_data[ , which(names(training_data) == "y")]
y_test <- test_data[ , which(names(test_data) == "y")]

barplot_results <- function(results) {
  ### Create a data.frame to be able to use ggplot2 appropriately.
  results_mean_std <- data.frame(results[1], results[2])
  rownames(results_mean_std) <- c('x1', 'x2', 'x3', 'x4')
  colnames(results_mean_std) <- c('col_means', 'col_stds')

  ### Use ggplot2 to create the barplot.
  ggplot(cbind(Features = rownames(results_mean_std), results_mean_std[1:4, ]),
         aes(x = reorder(Features, results_mean_std$col_means),
             y = results_mean_std$col_means)) +
    ### Plot the mean value bars.
    geom_bar(stat = "identity", fill = "steelblue") +
    ### Plot the standard deviations.
    geom_errorbar(aes(ymin = results_mean_std$col_means - results_mean_std$col_stds,
                      ymax = results_mean_std$col_means + results_mean_std$col_stds),
                  width = .1) +
    ### Set the labels correctly.
    labs(y = "Mean Value", x = "Features")
}

### Put all three above implemented functions
### together to get the PFI score for all features.
pfi_results <- n_times(fi, 10, TRUE, pfi_fname, model, X_test, y_test)

### Plot the result.
barplot_results(pfi_results)
```



1e)

X_3 is the most important feature, with X_1 and X_2 sharing the second place. PFI considers X_4 to be irrelevant.

According to the PFI interpretation rules, without further assumptions about the data, we know that

- X_1, X_2, X_3 are used by the model for it's prediction
- X_1, X_2, X_3 are dependent with Y and/or dependent with the covariates
- X_4 may be independent of Y and covariates and/or not used by the model. We only know that it is not both dependent and used by the model.

If we additionally analyze the data we find out that X_1, X_2 are dependent but X_3 is independent of all covariates.

As such, we can further conclude that X_3 is dependent with Y (but do not know for X_1, X_2 without looking into the data).

1f)

```
### Get the coefficients and the intercept.
model$coefficients
```

```
##  (Intercept)           x1           x2           x3           x4
##  0.006965344 -0.212975913  0.211488041  1.006078433  0.004199414
```

```
### Show the correlation structure in the testing data set.
cor(test_data)
```

```
##             x1          x2          x3          x4           y
## x1  1.00000000  0.99994347  0.08515749 -0.08959454  0.08417708
## x2  0.99994347  1.00000000  0.08627580 -0.08946404  0.08532504
## x3  0.08515749  0.08627580  1.00000000 -0.06232232  0.99553307
## x4 -0.08959454 -0.08946404 -0.06232232  1.00000000 -0.05780726
## y   0.08417708  0.08532504  0.99553307 -0.05780726  1.00000000
```

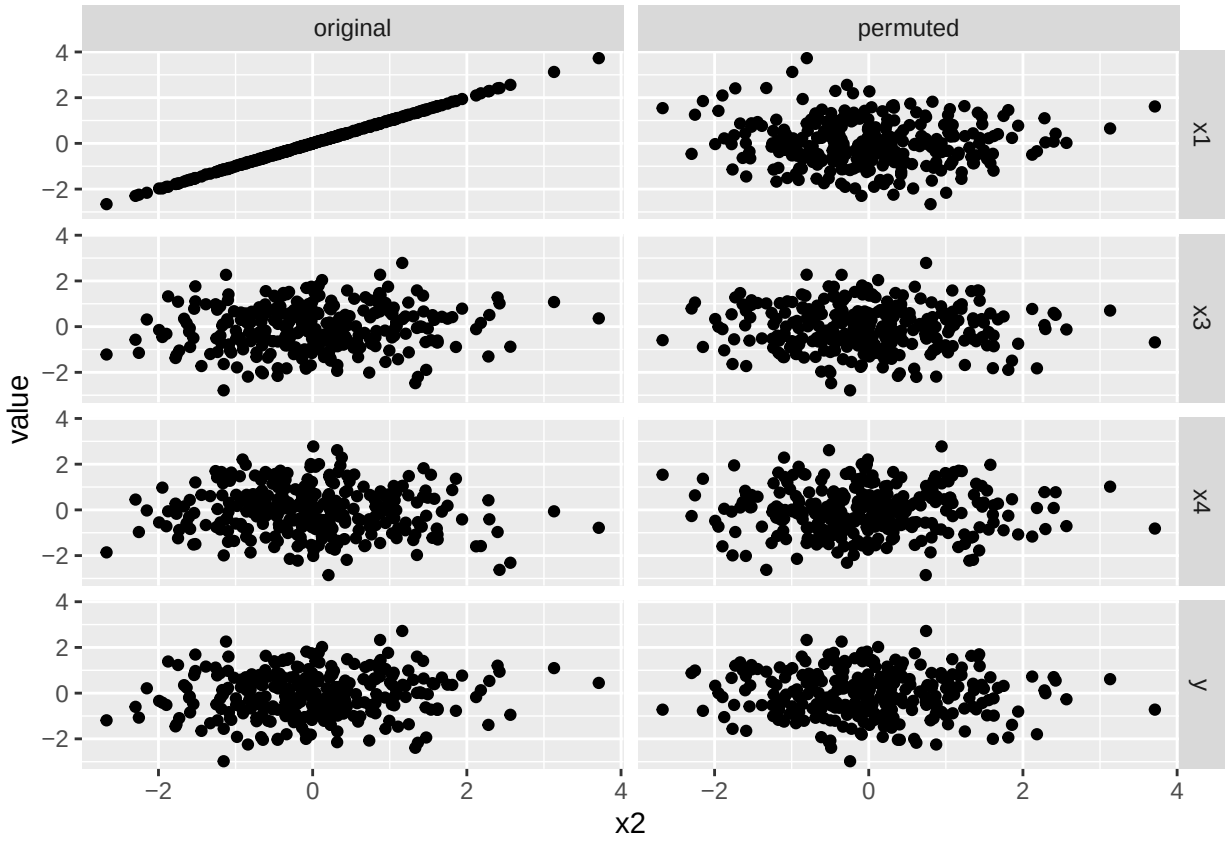
1g)

If we know the dependency structure of the covariates we can infer whether or not the PFI value is nonzero due to a dependency with the covariates or not. In our example we now know that x_3 is independent of its covariates, so we hypothesize that x_3 is actually dependent with y (and would perform importance tests as a consequence). Since x_1, x_2 are dependent, for those variables we cannot infer anything about the dependency with y with the covariates dependency structure and PFI alone.

1h)

We use the extrapolation.csv data set. We create a pairplot showing the pairwise scatterplot of the original feature as well as the corresponding perturbed variable (x2 in our case) with all remaining feature variables (and potentially y).

```
### Bring the data set in the correct form for ggplot2. This means we need every
### combination of x2 value to any other variable (which is done in the value column).
### The series column is for the group membership. Moreover we add a permuted version
### of x2 to the column x2. To keep track of this memberships we have the column type.
reshaped_test_data <- data.frame(x2 = c(rep(test_data$x2, 4),
                                        rep(sample(test_data$x2), 4)),
                                 type = rep(c('original', 'permuted'),
                                            each = 4 * nrow(test_data)),
                                 series = rep(rep(c('x1', 'x3', 'x4', 'y'),
                                                  each = nrow(test_data)), 2),
                                 value = rep(c(test_data$x1, test_data$x3,
                                               test_data$x4, test_data$y), 2))

### Create a scatterplot for both data situations at once.
ggplot(reshaped_test_data, aes(x2, value)) +
  geom_point() +
  facet_grid(series ~ type)
```



We can see that the strong correlation of x_2 with x_1 (top row) is broken after permutation (bottom row). All other pairwise (in)dependencies are unchanged. Note: We only assess pairwise, unconditional dependencies. Without assumptions about the data, we cannot know whether further conditional dependencies with the remaining covariates were broken.

Exercise 2: Conditional sampling based feature importance techniques

2a)

```
sample_cond_gaussian <- function(J, C, X_train, X_test, num_samples) {
  ### Order the training data for the decomposition.
  X_train_reordered <- cbind(X_train[ , J], X_train[ , C])

  ### Get the overall mean vector and covariance matrix.
  mu <- colMeans(X_train_reordered)
  sigma <- cov(X_train_reordered)

  ### Decomposition of the mean vector.
  mu_1 <- mu[1]
  mu_2 <- mu[-1]

  ### Decomposition of the covariance matrix.
```

```
  sigma_11 <- sigma[1, 1]
  sigma_12 <- sigma[1, -1]
  sigma_21 <- sigma[-1, 1]
  sigma_22 <- sigma[-1, -1]

  ### Get a matrix for the mean vectors for all features in the
  ### conditional set at once. The mean vectors are the columns.
  ### When giving just one matrix to the solve()-function it
  ### calculates the inverse via solving linear systems. This
  ### is much better conditioned than explicitly calculating the inverse.
  mu_bar_all_C <- sapply(1:nrow(X_test),
                         function(i) mu_1 + sigma_12 %*%
                           solve(sigma_22) %*%
                           (t(as.matrix(X_test[i , C],
                                        ncol = length(C))) - mu_2))

  ### Compute the covariance matrix with the given formula
  ### (which in our case is just the ordinary variance).
  sigma_bar <- sigma_11 - sigma_12 %*% solve(sigma_22) %*% sigma_21

  ### Now sample from the distributions given by mean vector and
  ### covariance matrix for all features in the conditional set
  ### at once. Return the result as a data.frame (number of columns
  ### is number of test points, number of rows is number of samples).
  as.data.frame(sapply(1:length(mu_bar_all_C),
                       function(i) rnorm(num_samples, mu_bar_all_C[i], sigma_bar)))
}
```

Show the sample results.

```
J <- c('x1')
C <- c('x2', 'x3', 'x4')
sample <- sample_cond_gaussian(J, C, X_train, X_test, 10)
sample[1:6, 1:6]
```

```
##          V1        V2         V3         V4         V5         V6
## 1 0.9780797 0.2995020 -0.3389727 -0.7290135 -0.3517852 -0.7757365
## 2 0.9779700 0.2991625 -0.3388512 -0.7290874 -0.3515434 -0.7758227
## 3 0.9780529 0.2993300 -0.3389262 -0.7289400 -0.3515971 -0.7758919
## 4 0.9780745 0.2992447 -0.3387931 -0.7288259 -0.3515458 -0.7757916
## 5 0.9780262 0.2991486 -0.3389470 -0.7289913 -0.3515663 -0.7758036
## 6 0.9778687 0.2993513 -0.3389349 -0.7287946 -0.3517010 -0.7757860
```

2b)

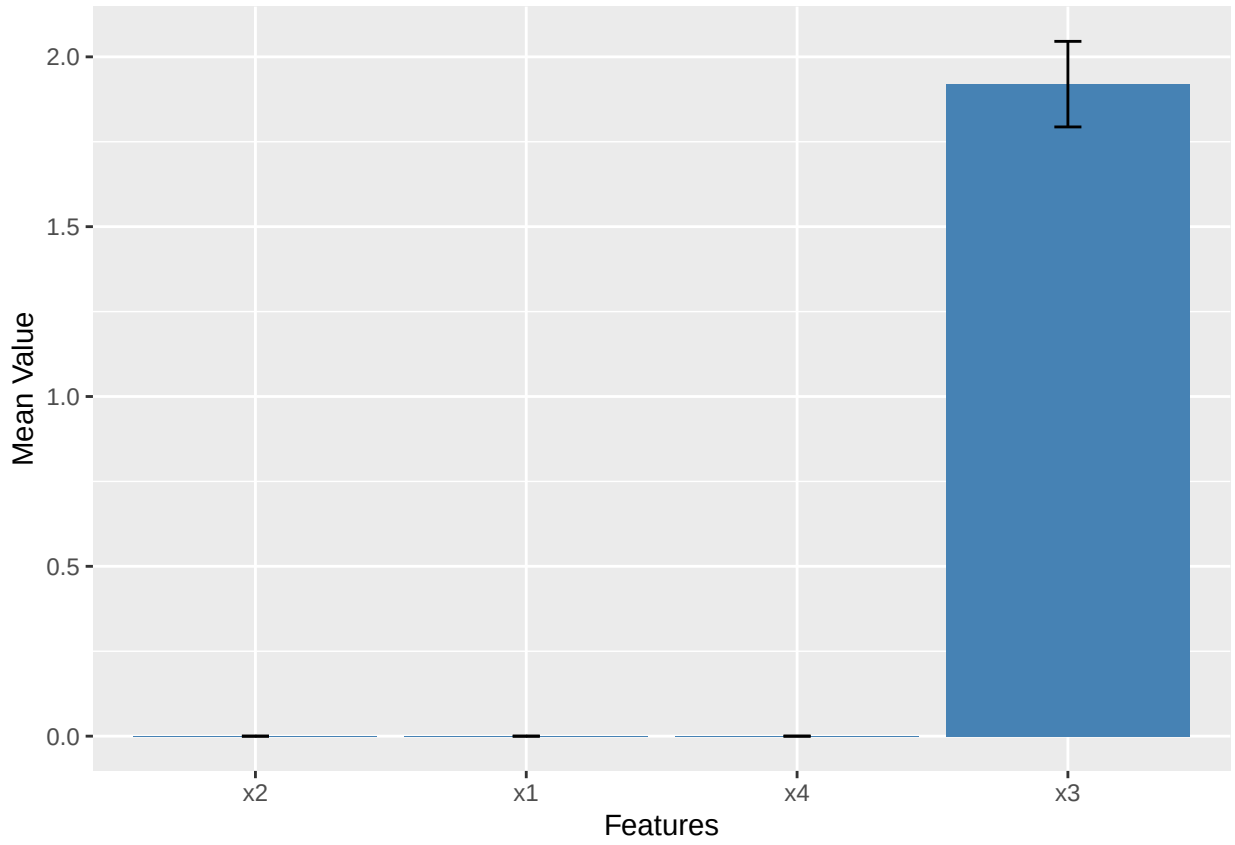
We implement conditional feature importance (CFI)

```
cfi_fname <- function(fname, model, X_train, X_test) {
  ### Get the names of the features to be conditioned on.
  C <- colnames(X_train)[colnames(X_train) != fname]

  ### Apply the conditional sampling procedure.
```

```
  sample <- sample_cond_gaussian(fname, C, X_train, X_test, 1)

  ### Permute the observations for feature fname.
  X_test_perm <- X_test
  X_test_perm[fname] <- sample

  ### Predict on the original data situation as well as on the permuted one.
  preds_original <- predict(model, X_test)
  preds_perm <- predict(model, X_test_perm)

  ### Calculate the MSE on both data situations.
  mse_original <- mean((y_test - preds_original) ^ 2)
  mse_perm <- mean((y_test - preds_perm) ^ 2)

  ### The PFI score is now defined as the increase in MSE when permuting the feature.
  mse_perm - mse_original
}
```

2c)

```
### The exact same procedure to generate the plot as in exercise 1.
cfi_results <- n_times(fi, 10, TRUE, cfi_fname, model, X_train, X_test)

barplot_results(cfi_results)
```



Now, only X_3 is considered relevant. We konw that

- X_3 is conditionally dependent with Y ($X_3 \not\perp Y | X_1, X_2, X_4$) and it is used by the model
- without further assumtions, we cannot make conclusions about the remaining variables. We only know that they are not at the same time conditionally dependent with Y and used by the model for it's prediction.

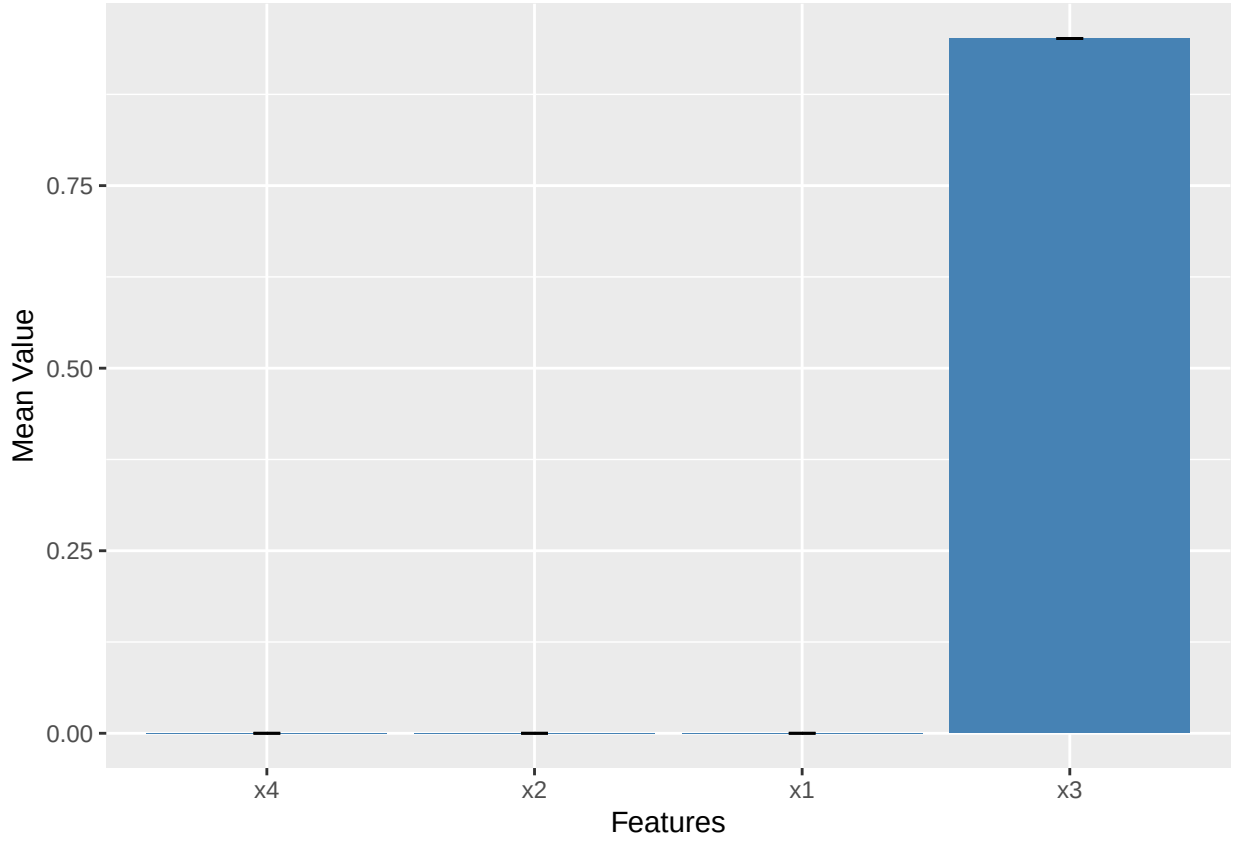
Exercise 3: Refitting based importance

3a)

```
loco <- function(fname, original_model, X_test, y_test, original_df, y_name) {
  ### Get the training data without the column with the feature of interest.
  remainder <- original_df[colnames(original_df) != fname]

  ### The usual training and testing split (with 70% training data).
  set.seed(100)
  inds <- sample(nrow(remainder), 0.7 * nrow(remainder))
  new_training_data <- remainder[inds, ]
  new_test_data <- remainder[-inds, ]

  ### Get the features and the target.
  loco_X_test <- new_test_data[ , colnames(new_test_data) != y_name]
  loco_y_test <- new_test_data[ , y_name]

  ### Generate the formula object we will give to the lm()-function.
  outcome <- names(new_training_data[y_name])
  variables <- names(loco_X_test)
  f <- as.formula(paste(outcome, paste(variables, collapse = " + "), sep = " ~ "))

  ### Train the OLS model.
  new_model <- lm(f, data = new_training_data)

  ### Get the MSE for the model with all features.
  preds_for_original <- predict(original_model, X_test)
  original_mse <- mean((y_test - preds_for_original) ^ 2)

  ### Get the MSE for the model without the feature of interest.
  predict_for_loco <- predict(new_model, loco_X_test)
  loco_mse <- mean((loco_y_test - predict_for_loco) ^ 2)

  ### The performance is given by the differences of the MSEs.
  loco_mse - original_mse
}

loco_naive <- function(original_df, original_model, X_test, y_test, y_name, ...) {
  ### Iterate over all features and apply the above implemented LOCO function.
  sapply(colnames(X_test),
         function(name)loco(name, original_df, original_model, X_test, y_test, y_name))
}
```

3b)

```
### The exact same procedure to generate the plot as in exercise 1.
loco_results <- n_times(fi, 10, FALSE, loco, model, X_test, y_test, df, 'y')

barplot_results(loco_results)
```



3c)

Result is similar as for CFI: only feature x_3 is considered relevant. We learn that x_3 provides more information than the remaining features combined, (while the other features do not contribute to the performance at all).