Exercise Shapley

Libraries

```
library(sets)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following object is masked from 'package:sets':
##
##     %>%

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(randomForest)
```

```
## randomForest 4.7-1.1

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine
```

Exercise 1: The Shapely Value

Defintion of useful functions

```r
### Define the payoff function.
payoff_func <- function(coalition) {
  ### Define boolean variables that indicate whether Timnit, Margret, Samy, Jeff
  ### and Larry are in the set or not.
  t <- 't' %in% coalition
  s <- 's' %in% coalition
  m <- 'm' %in% coalition
  j <- 'j' %in% coalition
  l <- 'l' %in% coalition

  ### Definition of the overall performance as given in the task.
  10*t + 10*m + 2*j + 20*(t & m) + 20*(t & m & s) - 30*((t | m | s) & j)
}

### Define a payoff difference function (which will be useful for several
  ### subtasks. The payoff argument will be helpful for checking the additivity
  ### property in task 1c)).
payoff_diff <- function(member, coalition, payoff=payoff_func) {
  payoff(c(member, coalition)) - payoff(coalition)
}

### Define a function that calculates the payoff differences given a list of
  ### coalitions (which will be useful for several subtasks. The payoff argument
  ### will be helpful for checking the additivity property in task 1c)).
payoff_diff_list <- function(member, coalitions, payoff=payoff_func) {
  sapply(1:length(coalitions), function(i) payoff_diff(member, coalitions[[i]], payoff))
}

### Define a function that returns all Shapley values for a given population
  ### (which will be useful for several subtasks. The payoff argument will be
  ### helpful for checking the additivity property in task 1c))
all_shapley_values <- function(population, payoff=payoff_func) {
  sapply(1:length(population), function(i) shapley(population[i], population, payoff))
}

### Define a preprocessing function which gives us a list of all possible
  ### coalitions (represented as sets) without the member player.
all_coalitions_without_members <- function(members, population) {
  ### Get all players except the member players.
  population_without_members <- setdiff(population, members)

  ### Use set_power function to get all sets of players without the member players.
  all_sets_without_members <- set_power(population_without_members)

  ### Convert them to a list for further processing.
  as.list(all_sets_without_members)
}
```

1a)

```
### Calculate the Shapley value given a player and the set of all players (The
  ### payoff argument will be helpful for checking the additivity property in
  ### task 1c)).
shapley <- function(member, population, payoff=payoff_func) {
  all_coalitions_without_member <- all_coalitions_without_members(member, population)

  ### Calculate the Shapley value according to the formula given in the lecture
  ### or in chapter 9.5.3.1 of Christoph Molnar's book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
  #### Compute the payoff differences with and without the member player for a
  ### coalitions at once.
  payoff_diffs <- payoff_diff_list(member, all_coalitions_without_member, payoff)

  #### Calculate the weights which are the combinatorical number of occurrences
  ### of each set.
  p <- length(population)
  weights <- sapply(1:length(all_coalitions_without_member), function(i) {
    cardinality_coalition <- length(all_coalitions_without_member[[i]])
    factorial(cardinality_coalition) * factorial(p - cardinality_coalition - 1) / factorial(p)
  })

  #### The Shapley value now is the weighted sum of the differences.
  sum(weights * payoff_diffs)
}
```

```
### Show the result.
member <- 't'
population <- c('t', 'm', 's', 'j', 'l')
shapley_result <- shapley(member, population)
print(shapley_result)
```

```
## [1] 24.16667
```

1b)

```
### Calculate the permutation based approximation of the Shapley value with
  ### num_iter iterations.
shapley_perm <- function(member, population, num_iter=10) {
  ### Sample num_iter permutations from the population (columns are the permutations).
  perms <- replicate(num_iter, sample(population))

  ### Find the index of the member player for every permutation (so every column,
  ### that´s what the 2 is indicating).
  member_idxs <- apply(perms, 2, function(x) match(member, x))

  ### Get the coalition for each permutation (which are just all the players with
  ### an index lower than the member player).
  ### They are now saved in a list with the length being the number of iterations.
  coalitions <- lapply(1:num_iter, function(i) perms[1:member_idxs[i] - 1, i])
```

```
  ### Estimate the Shapley value according to the formula given in the lecture or
  ### in chapter 9.5.3.3 of Christoph Molnar's book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
  ### Basically this is just taking the arithmetic mean of the differences of the
  ### payoffs of the coalitions with our member player and without.
  differences <- payoff_diff_list(member, coalitions)
  mean(differences)
}
```

```
### Show the result.
member <- 't'
population <- c('t', 'm', 's', 'j', 'l')
shapley_perm_result <- shapley_perm(member, population, 100000) ### This takes
  ### a few seconds.
print(shapley_perm_result)
```

```
## [1] 24.2138
```

```
### A pretty good estimate compared to the result from task a).
```

1c)

Symmetry Check

```
### Check the symmetry property of the Shapley value as defined in the lecture or
  ### in chapter 9.5.3.1 of Christoph Molnar's book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
symmetry_check <- function(j, k, population) {
  all_coalitions_without_candidates <- all_coalitions_without_members(c(j, k), population)

  ### Calculate the payoff differences for all possible coalitions including j
  ### (but excluding k) and for all possible coalitions including k (but excluding j).
  payoff_diff_j <- payoff_diff_list(j, all_coalitions_without_candidates)
  payoff_diff_k <- payoff_diff_list(k, all_coalitions_without_candidates)

  ### This logical statement now checks two things:
  ### 1. If the condition is violated.
  ### 2. If the Shapley values are identical.
  ### If the condition is violated the symmetry property holds because of the
  ### implication "condition => Shapley values identical" which does not give us
  ### any information whether the Shapley values are identical in this case
  ### (because from FALSE one can not imply anything).
  ### On the other hand, when the condition is met then the Shapley values have
  ### to be identical. This is ensured because in this case the condition is not
  ### violated meaning that the left hand side of the OR is FALSE. Therefore the
  ### right hand side has to hold which checks whether the Shapley values are identical.
  all(any(payoff_diff_j != payoff_diff_k) | shapley(j, population) == shapley(k, population))
}
```

```
### Check the symmetry property for all possible combinations of two players.
#### All combinations of two players.
all_combinations_two_players <- as.list(set_combn(population, m = 2))
#### The results of the application of the individual symmetry checks.
symmetry_check_all_combinations <- sapply(1:length(all_combinations_two_players), function(i) symmetry_
#### The combined result.
print(all(symmetry_check_all_combinations))
```

```
## [1] TRUE
```

```
### Seems to work for our toy example.
```

Dummy Property Check

```
### Check the dummy property of the Shapley value as defined in the lecture or
  ### in chapter 9.5.3.1 of Christoph Molnar's book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
dummy_check <- function(j, population) {
  all_coalitions_without_candidate <- all_coalitions_without_members(j, population)

  ### Calculate the payoff differences for all possible coalitions including j
  ### (but excluding k) and for all possible coalitions including k (but excluding j).
  payoff_diff_j <- payoff_diff_list(j, all_coalitions_without_candidate)

  ### This logical statement now checks two things:
  ### 1. If the condition is violated.
  ### 2. If the Shapley value is 0.
  ### If the condition is violated the dummy property holds because of the implication
  ### "condition => Shapley value is 0" which does not give us any information
  ### whether the Shapley value is 0 in this case (because from FALSE one can not
  ### imply anything).
  ### On the other hand, when the condition is met then the Shapley value has to
  ### be 0. This is ensured because in this case the condition is not violated
  ### meaning that the left hand side of the OR is FALSE. Therefore the right hand
  ### side has to hold which checks whether the Shapley value is 0.
  all(any(payoff_diff_j != 0) | shapley(j, population) == 0)
}
```

```
### Check the dummy property for all possible players.
#### The results of the application of the individual dummy property checks.
dummy_check_all_players <- sapply(1:length(population), function(i) dummy_check(population[i], populati
#### The combined result.
print(all(dummy_check_all_players))
```

```
## [1] TRUE
```

```
### Seems to work for our toy example.
```

Additivity Check

```
### Check the additivity property of the Shapley value as defined in the lecture
  ### or in chapter 9.5.3.1 of Christoph Molnar's book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
### Here we assume that the user is giving us two payoff functions that sum up to the original payoff f
additivity_check <- function(population, payoff_func_1, payoff_func_2) {
  ### Define the combined payoff function.
  payoff_func_1_and_2 <- function(x) payoff_func_1(x) + payoff_func_2(x)

  ### Get all the Shapley values when using payoff_func_1 and when using
  ### payoff_func_2.
  all_shapley_values_payoff_func_1 <- all_shapley_values(population, payoff_func_1)
  all_shapley_values_payoff_func_2 <- all_shapley_values(population, payoff_func_2)

  ### Get all the Shapley values when using the combined payoff function.
  all_shapley_values_payoff_func_1_and_2 <- all_shapley_values(population, payoff_func_1_and_2)

  ### Check if the combined Shapley value equals the sum of the Shapley values
  ### for every member of the population (rounding is necessary here since the
  ### values can differ because of numerical instabilities during computation).
  all(round(all_shapley_values_payoff_func_1_and_2, 5) == round(all_shapley_values_payoff_func_1 + all_s
}
```

```
### Define a new payoff function (which just consists of the first three summands
  ### of the original payoff function).
payoff_func_1 <- function(coalition) {
  ### Define boolean variables that indicate whether Timnit, Margret, Samy, Jeff
  ### and Larry are in the set or not.
  t <- 't' %in% coalition
  s <- 's' %in% coalition
  m <- 'm' %in% coalition
  j <- 'j' %in% coalition
  l <- 'l' %in% coalition

  ### Definition of the overall performance.
  10*t + 10*m + 2*j
}

### Define the second payoff function accordingly (which hence consists of the
  ### second three summands of the original payoff function).
payoff_func_2 <- function(coalition) {
  ### Define boolean variables that indicate whether Timnit, Margret, Samy, Jeff
  ### and Larry are in the set or not.
  t <- 't' %in% coalition
  s <- 's' %in% coalition
  m <- 'm' %in% coalition
  j <- 'j' %in% coalition
  l <- 'l' %in% coalition

  ### Definition of the overall performance.
  20*(t & m) + 20*(t & m & s) - 30*((t | m | s) & j)
}
```

```
### Check the additivity property (at least for the above decomposition of the
  ### original payoff function. Of course others are possible).
additivity_check_result <- additivity_check(population, payoff_func_1, payoff_func_2)
### Show the result.
print(additivity_check_result)
```

```
## [1] TRUE
```

```
### Seems to work for our toy example.
```

Efficiency Check

```
### Check the efficiency property of the Shapley value as defined in the lecture
  ### or in chapter 9.5.3.1 of Christoph Molnar´s book Interpretable Machine Learning
  ### (https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail).
efficiency_check <- function(population) {
  ### The left hand side gives us the total payoff of the population. On the right
  ### hand side we sum over the Shapley values of each individual player. For the
  ### efficiency property to hold these must be equal.
  payoff_func(population) == sum(all_shapley_values(population))
}
```

```
### Check the efficiency property for the whole population.
#### The result of the application of the efficiency property check for the population.
efficiency_check_result <- efficiency_check(population)
#### Show the result.
print(efficiency_check_result)
```

```
## [1] TRUE
```

```
### Seems to work for our toy example.
```

Exercise 2: SHAP

2a)

```
### For modelling a random forest we use the ranger package.
library(ranger)
```

```
##
## Attaching package: 'ranger'

## The following object is masked from 'package:randomForest':
##
##     importance
```

```r
### Read the dataset in via the read.csv()-function.
df <- read.csv(file = "fifa.csv")

### Now transform the character variable Man.of.the.Match which only contains the
  ### values "Yes" and "No" into a logical variable
### with values TRUE (former "Yes") and FALSE (former "No"). Note that this step
  ### has to be done before deleting all character
### variables because else we would remove our target variable as well.
df$Man.of.the.Match <- ifelse(df$Man.of.the.Match == "Yes", TRUE, FALSE)

### The aforementioned removal of all variables of type character.
df <- df[, !sapply(df, is.character)]

### Here we delete all columns containing at least one NA value.
df <- df[ , apply(df, 2, function(x) !any(is.na(x)))]

### The classical training and testing split.
set.seed(100)
train <- sample(nrow(df), 0.7 * nrow(df))
training_data <- df[train, ]
test_data <- df[-train, ]

### Here the split of our training and testing datasets into the features and the
  ### target which will be used later on.
X_train <- training_data[ , -which(names(training_data) == "Man.of.the.Match")]
X_test <- test_data[ , -which(names(test_data) == "Man.of.the.Match")]
y_train <- factor(training_data[ , which(names(training_data) == "Man.of.the.Match")])
y_test <- factor(test_data[ , which(names(test_data) == "Man.of.the.Match")])

### Modelling a random forest via the ranger package. Other packages are possible
  ### as well.
set.seed(120)
classifier_RF <- ranger(Man.of.the.Match ~ ., training_data)
classifier_RF
```

```
## Ranger result
##
## Call:
##  ranger(Man.of.the.Match ~ ., training_data)
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      89
## Number of independent variables:  18
## Mtry:                             4
## Target node size:                 1
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             38.20 %
```

2b)

```
### Computing the marginal sampling based SHAP value function as defined in the
  ### lecture or in chapter 9.6.2 of Christoph Molnar's book Interpretable Machine
  ### Learning (https://christophm.github.io/interpretable-ml-book/shap.html#kernelshap).
m_vfunc <- function(J, obs, X, model, n_samples = 10) {
  ### Get n_samples random observations from the given dataset.
  X_sample <- X[sample(nrow(X), n_samples, replace = TRUE), ]

  ### Now replace the features given in J for our samples by the values of the
  ### features in J given by the observation obs.
  X_sample[ , J] <- obs[ , J]

  ### Use the model to predict the target and take the arithmetic mean of the
  ### predictions.
  mean(predict(model, X_sample)$predictions)
}

m_vfunc(names(X_train)[1:3], X_test[1, ], X_train, classifier_RF, n_samples = 1000)
```

```
## [1] 0.555
```

2c)

```
### A function to sample for every entry in our data whether to include it or not.
sample_mask <- function(nrow, ncol) {
  ### Sample nrow * ncol times from a fair binomial distribution and arrange the
  ### result in an accordingly shaped matrix.
  matrix(rbinom(nrow * ncol, 1, 0.5), nrow = nrow)
}

### A function to return the weights for our randomly drawn mask. The formula can
  ### be found in chapter 9.6.2 of Christoph Molnar's book Interpretable Machine
  ### Learning (https://christophm.github.io/interpretable-ml-book/shap.html#kernelshap),
  ### it's the defintion of pi_x(z') where |z'| is the 1-norm (so in our case just
  ### the number of included features given by a 1 in the mask).
shap_weights <- function(mask) {
  ### The number of features (M in the formula).
  p <- ncol(mask)

  ### The number of included features |z'|.
  z_prime_abs <- rowSums(mask)

  ### The application of the formula.
  (p - 1) / (choose(p, z_prime_abs) * z_prime_abs * (p - z_prime_abs))
}

### A function to create the dataset generated during the SHAP calculations. This
  ### are just n_samples of the original observations
### where at the positions in the mask indicate where we replace the original
  ### values with the ones of the given observation obs.
```

```r
shap_data <- function(obs, X, n_samples, mask) {
  ### Get n_samples random observations from the given dataset.
  X_sample <- X[sample(nrow(X), n_samples, replace = TRUE), ]

  ### Blow up the size of the given observation to be compatible to our sampled
  ### dataset.
  df_obs <- obs[rep(1, n_samples), ]

  ### Now apply the mask. Since it only contains only 0's and 1's we get exactly
  ### what we want. At the positions where the mask is 0 we just get the first
  ### term. At the positions where the mask is 1 we basically subtract X_sample
  ### from X_sample and what is left is the respective entry in df_obs. So at
  ### these positions we effectively replace the original entries by the ones in df_obs.
  X_sample <- X_sample - mask * (df_obs - X_sample)

  ### Return the result.
  X_sample
}

### Now we can put things together and implement the sampling based KernelSHAP funtion.
kernel_shap <- function(obs, X, n_samples, model) {
  ### Get the SHAP weights and the predictions of a random sample of the given data.
  mask <- sample_mask(n_samples, ncol(X))
  df <- shap_data(obs, X, n_samples, mask)
  weights <- shap_weights(mask)
  pred <- predict(model, df)$predictions

  ### Now save the maskings as well as the respective predictions.
  df_shap <- as.data.frame(mask)
  names(df_shap) <- names(X)
  df_shap$pred <- pred

  ### Fit a weighted linear model using the above calculated weights. Since we
  ### are in a binary setting we perform logistic regression.
  wls_model <- glm(pred ~ ., family = binomial, data = df_shap, weights = weights)
  wls_model$coefficients
}

### Calculate the KernelSHAP values for the first instance of the test set.
obs <- X_test[1, ]
shap_vals <- kernel_shap(obs, X_train, 1000, classifier_RF)
```

```
## Warning in eval(family$initialize): non-integer #successes in a binomial glm!
```

```r
shap_vals
```

```
##            (Intercept)           Goal.Scored      Ball.Possession..
##             1.65457557           -3.28284372           -1.35771246
##               Attempts             On.Target            Off.Target
##            -0.64907257           -0.53569517            0.06829845
##                Blocked                Corners               Offsides
##             0.55281192           -2.18349575           -0.74949804
```

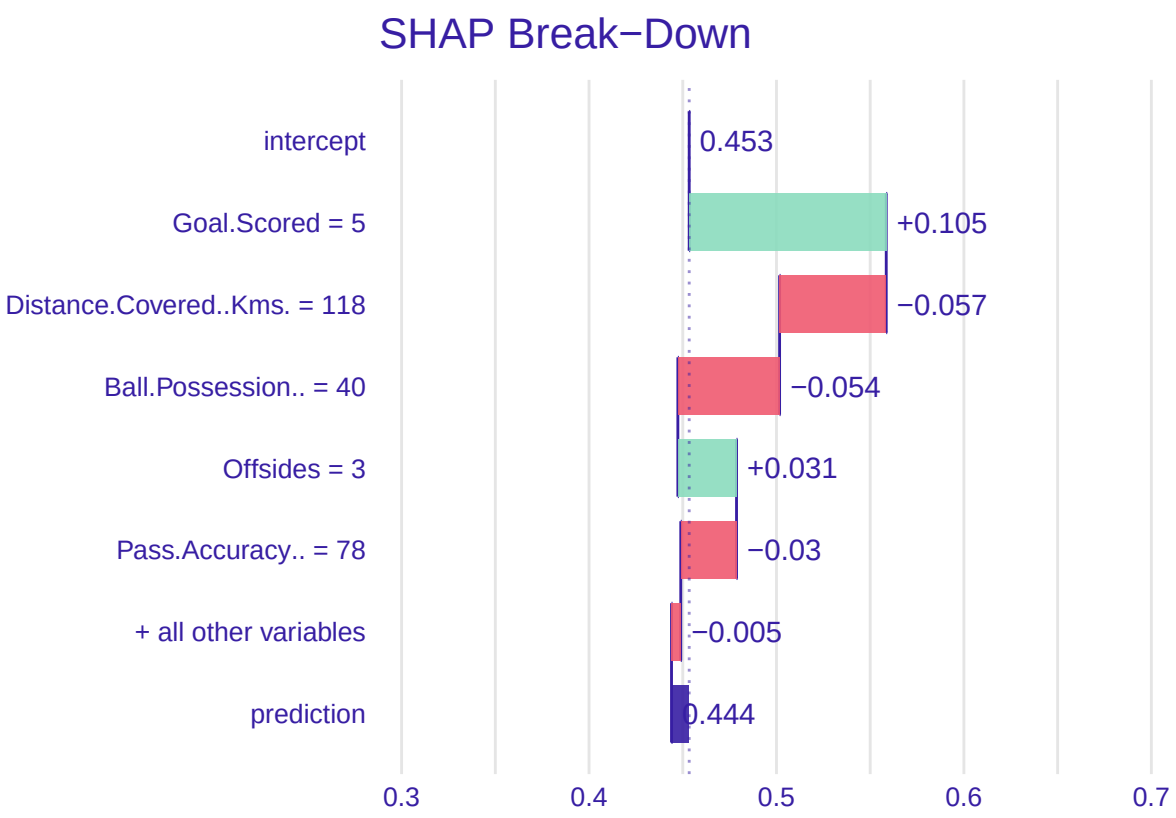
```
##          Free.Kicks               Saves     Pass.Accuracy..
##        -0.51869972          2.86721030        -0.75230186
##              Passes Distance.Covered..Kms.     Fouls.Committed
##        -0.42714693          -0.84923274        -0.96198507
##        Yellow.Card          Yellow...Red                Red
##         0.70772339          0.18069980        -0.12571821
##      Goals.in.PSO
##         0.08003492
```

2d)

Under the most current R version, treeshap does not work. You can either resort to python or downgrade to older versions of R.

SHAP values are expensive to compute. TreeSHAP offers a more efficient implementation that exploits the structure of tree-based models. Advanced knowledge: A further advantage of TreeSHAP is that it resamples the variables such that the joint distribution is preserved.

```
### To install the package.
# devtools::install_github('ModelOriented/treeshap')
library(treeshap)
unified_RF <- ranger.unify(classifier_RF, X_train)
explainer <- treeshap(unified_RF, X_test)
plot_contribution(explainer, obs = 1)
```



Exercise 3: LIME ## 3a)

```r
library("DALEX")
```

```
## Welcome to DALEX (version: 2.4.3).
## Find examples and detailed introduction at: http://ema.drwhy.ai/
## Additional features will be available after installation of: ggpubr.
## Use 'install_dependencies()' to get all suggested dependencies

##
## Attaching package: 'DALEX'

## The following objects are masked from 'package:treeshap':
##
##     colors_breakdown_drwhy, colors_discrete_drwhy, theme_drwhy,
##     theme_drwhy_vertical

## The following object is masked from 'package:dplyr':
##
##     explain
```

```r
library("randomForest")

### Apply a complex model to the data.
rf_model <- randomForest(Man.of.the.Match ~ ., training_data)
```

```
## Warning in randomForest.default(m, y, ...): The response has five or fewer
## unique values.  Are you sure you want to do regression?
```

```r
model_ex <- DALEX::explain(model = rf_model,
                           data = X_train,
                           y = y_train,
                           label = "Random Forest")
```

```
## Preparation of a new explainer is initiated
##   -> model label       :  Random Forest
##   -> data              :  89  rows  18  cols
##   -> target variable   :  89  values
##   -> predict function  :  yhat.randomForest  will be used (  default  )
##   -> predicted values  :  No value for predict function target column. (  default  )
##   -> model_info        :  package randomForest , ver. 4.7.1.1 , task regression (  default  )
##   -> model_info        :  Model info detected regression task but 'y' is a factor .  (  WARNING  )
##   -> model_info        :  By deafult regressions tasks supports only numercical 'y' parameter.
##   -> model_info        :  Consider changing to numerical vector.
##   -> model_info        :  Otherwise I will not be able to calculate residuals or loss function.
##   -> predicted values  :  numerical, min =  0.03453333 , mean =  0.454679 , max =  0.9098333
##   -> residual function :  difference between y and yhat (  default  )

## Warning in Ops.factor(y, predict_function(model, data)): '-' not meaningful for
## factors

##   -> residuals         :  numerical, min =  NA , mean =  NA , max =  NA
##   A new explainer has been created!
```

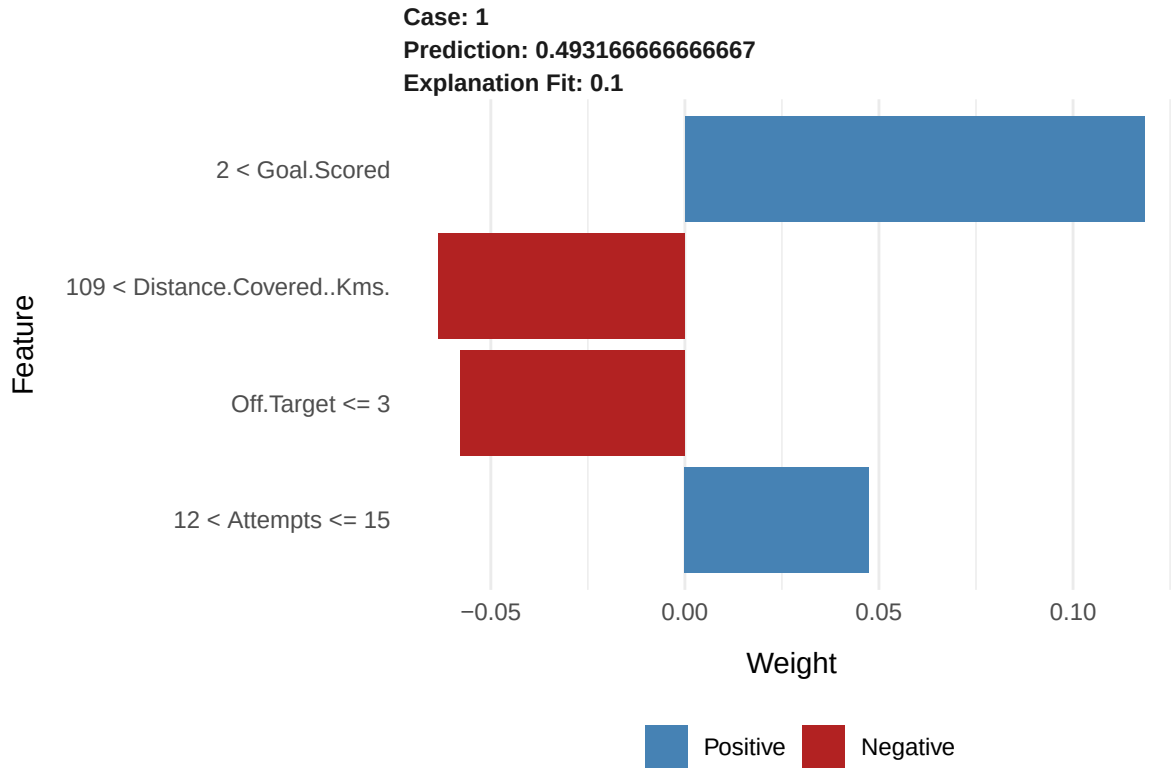
```r
library("DALEXtra")
library("iml")
library("lime")
```

```
##
## Attaching package: 'lime'

## The following object is masked from 'package:DALEX':
##
##     explain

## The following object is masked from 'package:dplyr':
##
##     explain
```

```r
### Now we try to get local explanations for the complicated model via LIME.
model_type.dalex_explainer <- DALEXtra::model_type.dalex_explainer
predict_model.dalex_explainer <- DALEXtra::predict_model.dalex_explainer

iml_obs <- predict_surrogate(explainer = model_ex,
                  new_observation = X_test[1, ],
                  type = "lime")
```

```
## Warning: Yellow...Red does not contain enough variance to use quantile binning.
## Using standard binning instead.

## Warning: Red does not contain enough variance to use quantile binning. Using
## standard binning instead.

## Warning: Goals.in.PSO does not contain enough variance to use quantile binning.
## Using standard binning instead.

## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range

## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range
```

```r
plot(iml_obs)
```



3b)

```
### Set the kernel width to 10.
iml_obs <- predict_surrogate(explainer = model_ex,
                  new_observation = X_test[1, ],
                  type = "lime",
                  kernel_width = 10)
```

```
## Warning: Yellow...Red does not contain enough variance to use quantile binning.
## Using standard binning instead.
```

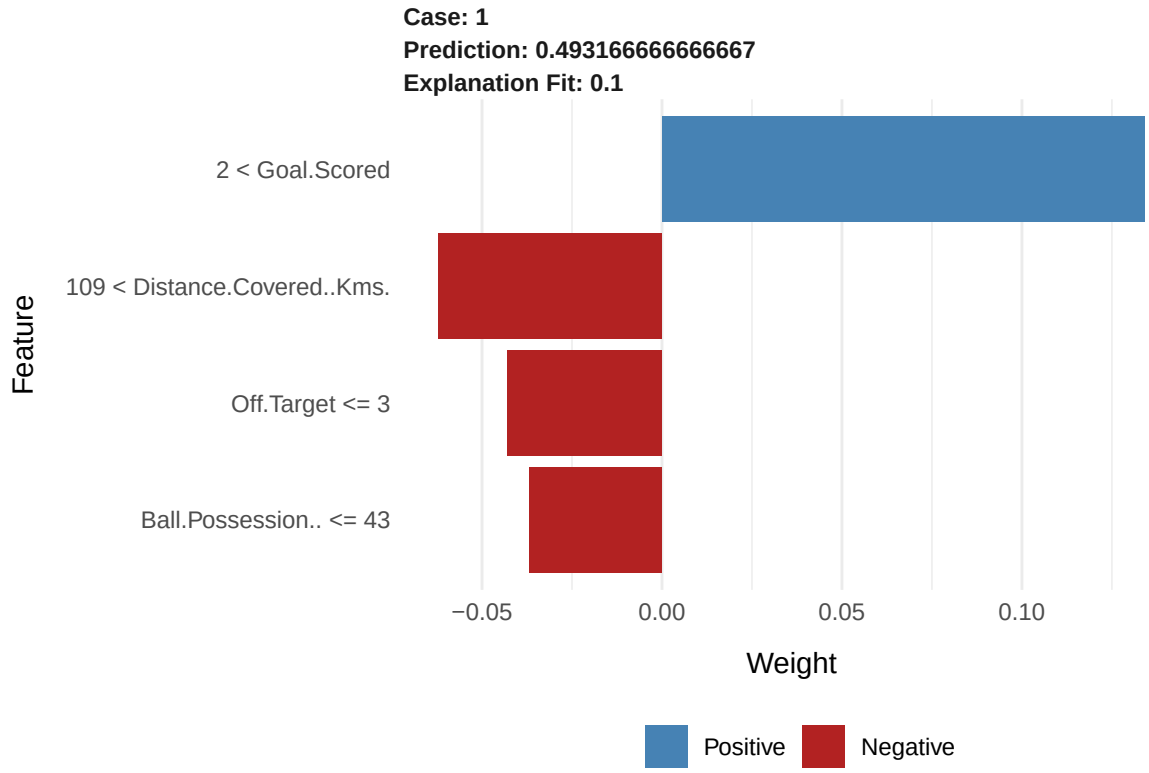
```
## Warning: Red does not contain enough variance to use quantile binning. Using
## standard binning instead.
```

```
## Warning: Goals.in.PSO does not contain enough variance to use quantile binning.
## Using standard binning instead.
```

```
## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range
```

```
## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range
```

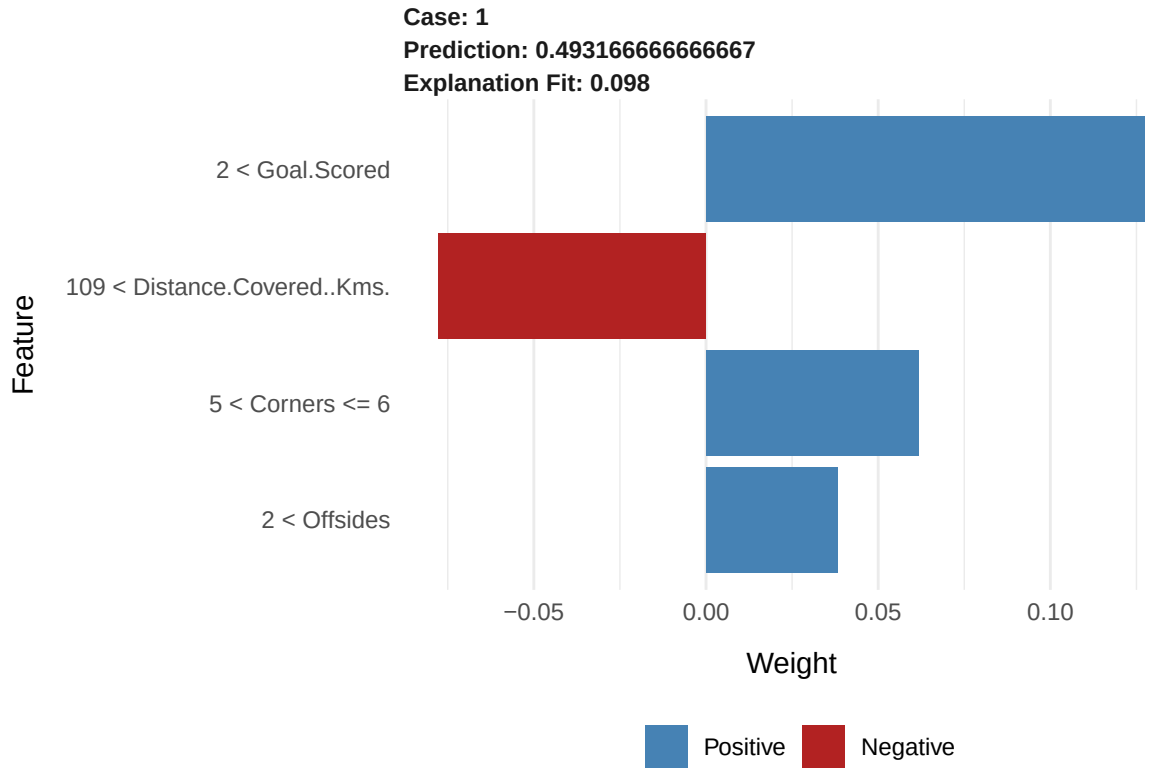
```
plot(iml_obs)
```




```
### Set the kernel width to 0.001.
iml_obs <- predict_surrogate(explainer = model_ex,
                  new_observation = X_test[1, ],
                  type = "lime",
                  kernel_width = 0.001)
```

```
## Warning: Yellow...Red does not contain enough variance to use quantile binning.
## Using standard binning instead.

## Warning: Red does not contain enough variance to use quantile binning. Using
## standard binning instead.

## Warning: Goals.in.PSO does not contain enough variance to use quantile binning.
## Using standard binning instead.

## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range

## Warning in gower_work(x = x, y = y, pair_x = pair_x, pair_y = pair_y, n = NULL,
## : skipping variable with zero or non-finite range
```

```
plot(iml_obs)
```



3c)

Both SHAP and Lime rely on a linear model approximation of the model. For Lime, the normal feature values are used, for SHAP a transformed distribution indicating coalition membership for a sample is relied upon. Lime weights the different samples according to their distance to the observation of interest. In contrast, SHAP weights them according to the Shapley kernel weights (which simulate sampling random permutations).