Solution 1:

a) Inspect Implemented Functions

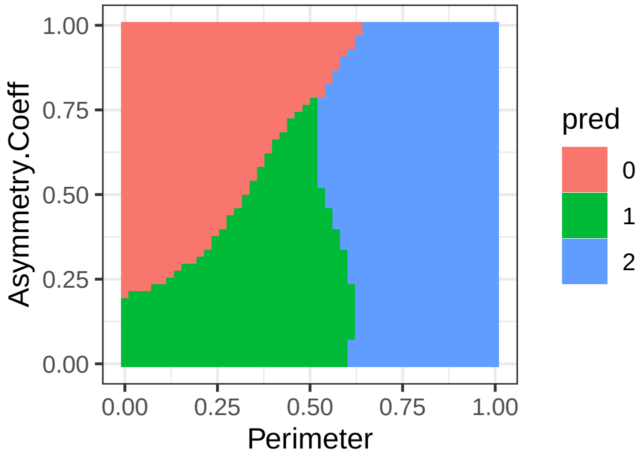
```
library("docstring")
library("ggplot2")
library("rpart")

get_grid = function(model, dataset, points_per_feature = 50) {
  #' Retrieve grid data for plotting a two-dimensional graph with
  #' `points_per_feature` for each axis. The space is created by
  #' the hyperparameters' lower and upper values. Only the first two input
  #' labels are used.

  #' @param model: Classifier which can call a predict method.
  #' @param dataset (data.frame): Input dataset (only contains two features).
  #' @param points_per_feature (integer(1)): How many points in each dimension.

  #' @return  Dataframe with three columns:
  #'      - equidistant grid of first feature
  #'      - equidistant grid of second feature
  #'      - pred: prediction values for given feature input

  range_x1 = range(dataset[,1])
  range_x2 = range(dataset[,2])

  x1 = seq(range_x1[1], range_x1[2], length.out = points_per_feature)
  x2 = seq(range_x2[1], range_x2[2], length.out = points_per_feature)

  X = data.frame(expand.grid(x1, x2))
  names(X) = names(dataset)
  pred = predict(model, X, type = "class")

  return(data.frame(X, pred = pred))

}

plot_grid = function(grid) {
  #'  Uses the grid data to add a color grid to the plot.
  #'
  #'  @param grid (data.frame): Grid data for plot.
  #'
  #'  @return ggplot: grid data plotted with coloring displaying the prediction
  #'  surface.

  xnam = names(grid)[1]
  ynam = names(grid)[2]
  ggplot(grid, aes_string(x = xnam, y = ynam, fill = "pred")) +
    ggplot2::geom_tile() +
    ggplot2::guides(z = ggplot2::guide_legend(title = "pred")) +
    ggplot2::theme_bw() +
    ggplot2::theme(legend.position = "right")
}


plot_points_in_grid = function(plt, df, weights = NULL, x_interest = NULL,
                               size = 4L) {
  #' Given a plot, add scatter points from `df` and `x_interest`.
  #'
  #' @param plt (ggplot): Plot with color grid.
  #' @param df (data.frame): Points which should be added to the plot.
  #' @param weights (numeric): Normalized weights with elements equal to
  #' the number of rows in `df`. Weights are used to determine the size of the
  #' points in the plot. If NULL, size of all points are equal.
  #' @param x_interest (data.frame): Single point (one row dataset)
```

```r
  #' whose prediction we want to explain. If NULL (default) no point is added.
  #' @param size (numeric(1)): Default size of the points. Default 4L.
  #'

  if (!is.null(weights)) {
    w = weights
  } else {
    w = 1L
  }
  xnam = names(df)[1]
  ynam = names(df)[2]
  plt = plt +
    geom_point(mapping = aes_string(x = xnam, y = ynam, color = "pred"),
               size = w*size, data = df, alpha = 2) +
    scale_colour_hue(l = 40)

  if (!is.null(x_interest)) {
    x_interest$pred = "1"
    plt = plt + geom_point(mapping = aes_string(x = xnam, y = ynam),
                           x_interest, colour = "red")
  }
  return(plt)
}
```

Example:

```r
set.seed(2022L)
library("e1071") # SVM
library("gridExtra") # to plot two ggplots next to each other

dataset = read.csv(file = "exercises/local-explanations/rsrc/datasets/wheat_seeds.csv")
dataset$Type = as.factor(dataset$Type)
table(dataset$Type)

min_max_norm <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}

dataset  = dataset[c("Perimeter", "Asymmetry.Coeff", "Type")]
dataset$Perimeter = min_max_norm(dataset$Perimeter)
dataset$Asymmetry.Coeff = min_max_norm(dataset$Asymmetry.Coeff)

traindata = dataset[sample(seq_len(nrow(dataset)),
                           round(0.6*nrow(dataset)), replace = TRUE),]

# Fit a svm to the data
mod = svm(Type ~ ., data = traindata)
dataset$Type = NULL

# Compute counterfactual for first observation
x_interest = data.frame(Perimeter = 0.31, Asymmetry.Coeff = 0.37)

# Parameters for method
points_per_feature = 50L
n_points = 1000L
```

```r
print("Run 'get_grid' ...")
grid = get_grid(model = mod, dataset = dataset,
                points_per_feature = points_per_feature)

print("Run `plot_grid` ...")
plot = plot_grid(grid)
```

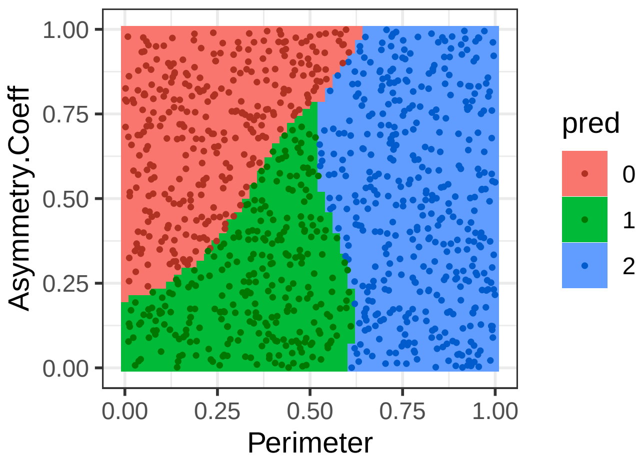


b) Sample Points

```
sample_points = function(model, dataset, num_points, seed=0) {

  #' Samples points for the two first features and uses the model to
  #' receive a prediction for these sampled points.
  #'
  #'  @param model: Classifier which can call a predict method.
  #'    @param dataset (data.frame): Input dataset (only contains two features).
  #'  @param num_points (int): How many points should be sampled.
  #'  @param seed (int): Seed to feed random.
  #'
  #' @return dataset (data.frame) of sampled data including a column 'pred' with
  #' the obtained prediction of the model for the sampled data.

  set.seed(seed)
  range_x1 = range(dataset[, 1])
  range_x2 = range(dataset[, 2])

  x1 = runif(n = num_points, min = range_x1[1], max = range_x1[2])
  x2 = runif(n = num_points, min = range_x2[1], max = range_x2[2])
  Z = data.frame(x1, x2)
  names(Z) = names(dataset)
  pred = predict(model, Z)

  return(data.frame(Z, pred))
```

Example:

```
print("Run `sample_points` ...")
samp = sample_points(model = mod, dataset = dataset, num_points = n_points)

print("Run `plot_points_in_grid` ...")
plot = plot_points_in_grid(plt = plot, df = samp, size = .5)
```

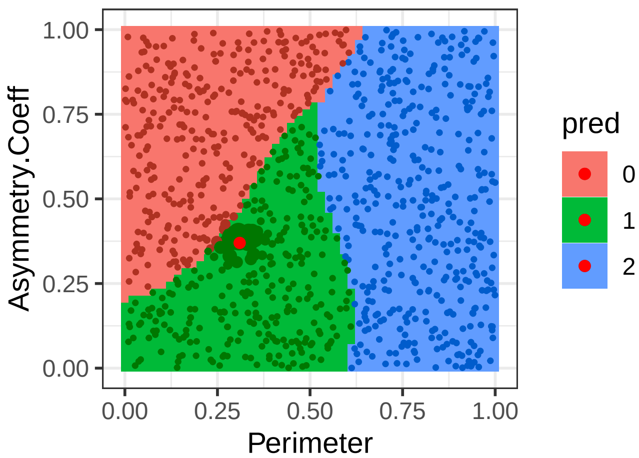


c) Weight Points

```
weight_points = function(x_interest, df, kernel_width=0.2) {
  #' For every x in `df` returns a weight depending on the exponential kernel
  #' distance to `x_interest`.
  #'
  #' @param x_interest (data.frame): Single point (one row dataset)
  #' whose prediction we want to explain.
  #' @param df (data.frame): Data which needs to be weighted
  #' (later used for surrogate model).
  #' @param kernel_width (float): Kernel width for exponential kernel.
  #'
  #' @return weights (numeric): Normalized weights between
  #' 0..1 for all datapoints in df.

  if ("pred" %in% names(df)) {
    df = df[names(df) != "pred"]
  }

  df = as.matrix(df)
  weights = apply(df, MARGIN = 1, FUN = function(x) {
    eucldist = sqrt(sum((x-x_interest)^2))
    exp(-eucldist/(kernel_width*kernel_width))
  })

  # Normalize between 0 and 1
  weights = (weights - min(weights)) / (max(weights) - min(weights))

  return(weights)
```

Example:

```
print("Run `weight_points` ...")
w = weight_points(x_interest = x_interest, df = samp, kernel_width = 0.2)

print("Run `plot_points_in_grid` ...")
plot = plot_points_in_grid(plt = plot, df = samp, weights = w,
                           x_interest = x_interest)
```



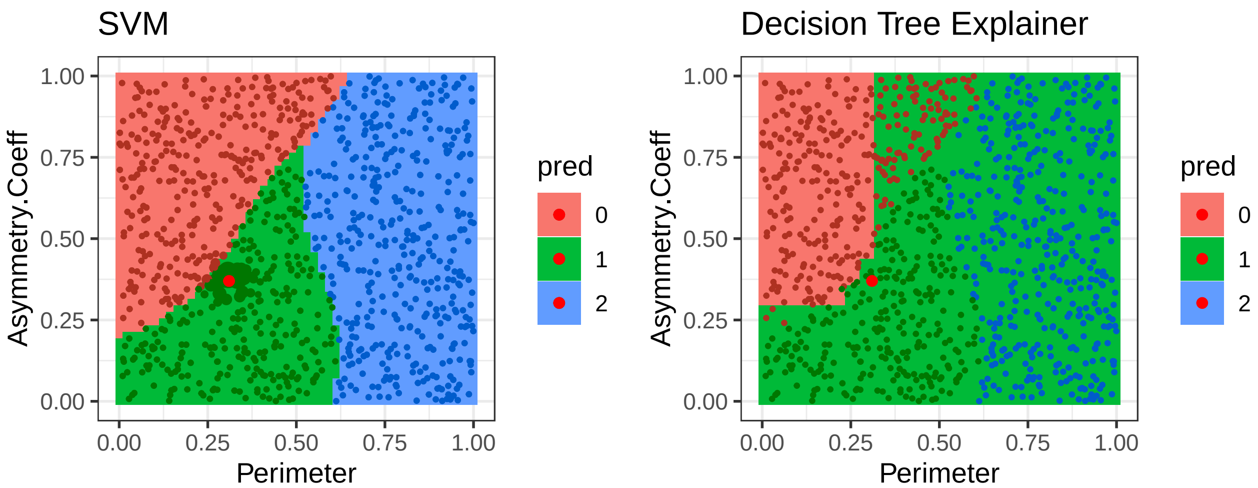
d) Fit Local Surrogate

```
fit_explainer_model = function(df, weights = NULL, seed = 0) {
  #' Fits a decision tree to the weighted data
  #'
  #' @param df (data.frame): Data for surrogate model, must include an outcome
  #'  variable `pred`.
  #' @param weights (numeric): Normalized weights with number of elements equal
  #' to number of rows of `df`.
  #' @param seed (int): Seed for the decision tree.
  #'
  #' @return model (rpart): Fitted explainer model.
  set.seed(seed)
  xnam = names(df)[1]
```

```
  ynam = names(df)[2]
  form = formula(paste("pred ~", xnam, "+", ynam))
  tree = rpart(form, weights = weights, data = df)
  return(tree)
```

Example:

```
print("Run `fit_explainer_model` ...")
explainer = fit_explainer_model(df = samp, weights = w)

print("Compare models ...")
plt1 = plot_points_in_grid(plt = plot, df = samp,
                           x_interest = x_interest, size = .5)
grid2 = get_grid(model = explainer, dataset = dataset,
                 points_per_feature = points_per_feature)
plot2 = plot_grid(grid2)
plt2 = plot_points_in_grid(plt = plot2, df = samp,
                           x_interest = x_interest, size = .5)

plt1 = plt1 + ggplot2::ggtitle("SVM")
plt2 = plt2 + ggplot2::ggtitle("Decision Tree Explainer")

pdf(file="exercises/local-explanations/figure/fit_explainer_model_plot_lime.pdf",height=2.75,width=7)

plot = grid.arrange(plt1, plt2, ncol = 2L)
```



Solution 2:

a) Implementation of WhatIf:

```
library("docstring")
library(StatMatch)

generate_whatif = function(x_interest, model, dataset) {
  #' Computes whatif counterfactuals for binary classification models,
  #' i.e., the closest data point with a different prediction.
  #
  #' @param x_interest (data.frame): Datapoint of interest, a single
  #' row data set.
  #' @param model: Binary classifier which can call a predict method.
  #' @param dataset (data.frame): Input data
  #'
  #' @return counterfactual (data.frame): data.frame with one row
  #' presenting the counterfactuals
  #'     closest to `x_interest` with a different prediction.

  # subset dataset to the observations having a prediction different
```

```r
  # to x_interest
  pred = predict(model, newdata = x_interest)
  preddata = predict(model, dataset)
  idx = which(preddata != pred)
  dataset = dataset[idx, ]

  # Pairwise Gower distances
  dists = StatMatch::gower.dist(data.x = x_interest, data.y = dataset)
  minid = order(dists)[1]

  # Return nearest datapoint
  return(dataset[minid, ])
}
```

Example: wheat_seeds.csv

```r
df = read.csv(file = "exercises/local-explanations/rsrc/datasets/wheat_seeds.csv")
table(df$Type)

# Create a binary classification task
df$Type = as.factor(ifelse(df$Type == "0", 1, df$Type))
table(df$Type)

# Fit a random forest to the data
mod = randomForest::randomForest(Type ~ ., data = df)
df$Type = NULL
# Compute counterfactual for first observation
x_interest = df[1,]
```

	Area	Perimeter	Compactness	Kernel.Length	Kernel.Width	Asymmetry.Coeff	Kernel.Groove
1	15.26	14.84	0.87	5.76	3.31	2.22	5.22

```r
cf = generate_whatif(x_interest = x_interest, model = mod, dataset = df)
```

	Area	Perimeter	Compactness	Kernel.Length	Kernel.Width	Asymmetry.Coeff	Kernel.Groove
133	15.60	15.11	0.86	5.83	3.29	2.73	5.75

b) Counterfactuals generated with WhatIf are valid and proximal, since they reflect the closest training datapoint with the desired/different prediction. The counterfactuals are also plausible since by definition they adhere to the data manifold. The counterfactuals are not sparse and might propose changes to many features - this is a clear disadvantage of this method.