modelStudio - perks and features

Hubert Baniecki

2023-08-31

Source: `vignettes/ms-perks-features.Rmd`

The `modelStudio()` function computes various (instance and dataset level) model explanations and produces a customisable dashboard, which consists of multiple panels for plots with their short descriptions. Easily save the dashboard and share it with others. Tools for Explanatory Model Analysis unite with tools for Exploratory Data Analysis to give a broad overview of the model behavior.

Let's use `HR` dataset to explore `modelStudio` parameters:

```
train <- DALEX::HR
train$fired <- as.factor(ifelse(train$status == "fired", 1, 0))
train$status <- NULL

head(train)
```

DALEX::HR dataset

gender	age	hours	evaluation	salary	fired
male	32.58	41.89	3	1	1
female	41.21	36.34	2	5	1
male	37.71	36.82	3	0	1
female	30.06	38.96	3	2	1
male	21.10	62.15	5	3	0
male	40.12	69.54	2	0	1

Prepare `HR_test` data and a `ranger` model for the explainer:

```
# fit a ranger model
library("ranger")
model <- ranger(fired ~., data = train, probability = TRUE)

# prepare validation dataset
test <- DALEX::HR_test[1:1000,]
test$fired <- ifelse(test$status == "fired", 1, 0)
test$status <- NULL

# create an explainer for the model
explainer <- DALEX::explain(model,
                            data = test,
                            y = test$fired)

# start modelStudio
library("modelStudio")
```

modelStudio parameters

instance explanations

Pass data points to the `new_observation` parameter for instance explanations such as Break Down, Shapley Values and Ceteris Paribus Profiles. Use `new_observation_y` to show their true labels.

```
new_observation <- test[1:3,]
rownames(new_observation) <- c("John Snow", "Arya Stark", "Samwell Tarly")
true_labels <- test[1:3,]$fired

modelStudio(explainer,
            new_observation = new_observation,
            new_observation_y  = true_labels)
```

If `new_observation = NULL`, then choose `new_observation_n` observations, evenly spread by the order of `y_hat`. This shall always include the observations, which ids are `which.min(y_hat)` and `which.max(y_hat)`.

```
modelStudio(explainer, new_observation_n = 5) # default is 3
```

grid size

Achieve bigger or smaller `modelStudio` grid with `facet_dim` parameter.

```
# small dashboard with 2 panels
modelStudio(explainer,
            facet_dim = c(1,2))

# large dashboard with 9 panels
modelStudio(explainer,
            facet_dim = c(3,3))
```

animations

Manipulate `time` parameter to set animation length. Value 0 will make them invisible.

```
# slow down animations
modelStudio(explainer,
            time = 1000)

# turn off animations
modelStudio(explainer,
            time = 0)
```

more calculations means more time

- `N` is a number of observations used for calculation of Partial Dependence and Accumulated Dependence Profiles (default is `300`).
- `N_fi` is a number of observations used for calculation of Feature Importance (default is `N*10`).
- `N_sv` is a number of observations used for calculation of Shapley Values (default is `N*3`).
- `B` is a number of permutation rounds used for calculation of Shapley Values (default is `10`).
- `B_fi` is a number of permutation rounds used for calculation of Feature Importance (default is `B`).

Decrease `N` and `B` parameters to lower the computation time or increase them to get more accurate empirical results.

```
# faster, less precise
modelStudio(explainer,
            N = 200, B = 5)

# slower, more precise
modelStudio(explainer,
            N = 500, B = 15)
```

no EDA mode

Don't compute the EDA plots if they are not needed. Set the `eda` parameter to `FALSE`.

```
modelStudio(explainer,
            eda = FALSE)
```

progress bar

Hide computation progress bar messages with `show_info` parameter.

```
modelStudio(explainer,
            show_info = FALSE)
```

viewer or browser?

Change `viewer` parameter to set where to display `modelStudio`. Best described in `r2d3` documentation.

```
modelStudio(explainer,
            viewer = "browser")
```

parallel computation

Speed up `modelStudio` computation by setting `parallel` parameter to `TRUE`. It uses `parallelMap` package to calculate local explainers faster. It is really useful when using `modelStudio` with complicated models, vast datasets or **many observations are being processed**.

All options can be set outside of the function call. How to use parallelMap.

```
# set up the cluster
options(
  parallelMap.default.mode        = "socket",
  parallelMap.default.cpus        = 4,
  parallelMap.default.show.info   = FALSE
)

# calculations of local explanations will be distributed into 4 cores
modelStudio(explainer,
            new_observation = test[1:16,],
            parallel = TRUE)
```

additional options

Customize some of the `modelStudio` looks by overwriting default options returned by the `ms_options()` function. Full list of options.

```
# set additional graphical parameters
new_options <- ms_options(
  show_subtitle = TRUE,
  bd_subtitle = "Hello World",
  line_size = 5,
  point_size = 9,
  line_color = "pink",
  point_color = "purple",
  bd_positive_color = "yellow",
  bd_negative_color = "orange"
)

modelStudio(explainer,
            options = new_options)
```

All visual options can be changed after the calculations using `ms_update_options()`.

```
old_ms <- modelStudio(explainer)
old_ms

# update the options
new_ms <- ms_update_options(old_ms,
                            time = 0,
                            facet_dim = c(1,2),
                            margin_left = 150)
new_ms
```

update observations

Use `ms_update_observations()` to add more observations with their local explanations to the `modelStudio`.

```
old_ms <- modelStudio(explainer)
old_ms

# add new observations
plus_ms <- ms_update_observations(old_ms,
                                  explainer,
                                  new_observation = test[101:102,])
plus_ms

# overwrite old observations
new_ms <- ms_update_observations(old_ms,
                                 explainer,
                                 new_observation = test[103:104,],
                                 overwrite = TRUE)
new_ms
```

Shiny

Use the `widget_id` argument and `r2d3` package to render the `modelStudio` output in Shiny. See Using r2d3 with Shiny and consider the following example:

```
library(shiny)
library(r2d3)


ui <- fluidPage(
  textInput("text", h3("Text input"),
            value = "Enter text..."),
  uiOutput('dashboard')
)

server <- function(input, output) {
  #:# id of div where modelStudio will appear
  WIDGET_ID = 'MODELSTUDIO'

  #:# create modelStudio
  library(modelStudio)
  library(DALEX)
  model <- glm(survived ~., data = titanic_imputed, family = "binomial")
  explainer <- DALEX::explain(model,
                              data = titanic_imputed,
                              y = titanic_imputed$survived,
                              label = "Titanic GLM",
                              verbose = FALSE)
  ms <- modelStudio(explainer,
                    widget_id = WIDGET_ID,  #:# use the widget_id
                    show_info = FALSE)
  ms$elementId <- NULL                      #:# remove elementId to stop the warnin

  #:# basic render d3 output
  output[[WIDGET_ID]] <- renderD3({
    ms
  })

  #:# use render ui to set proper width and height
  output$dashboard <- renderUI({
    d3Output(WIDGET_ID, width=ms$width, height=ms$height)
  })
}

shinyApp(ui = ui, server = server)
```

DALEXtra

Use `explain_*()` functions from the DALEXtra package to explain various models.

Bellow basic example of making `modelStudio` for a `mlr` model using `explain_mlr()`.

```
library(DALEXtra)
library(mlr)

# fit a model
task <- makeClassifTask(id = "task", data = train, target = "fired")
learner <- makeLearner("classif.ranger", predict.type = "prob")
model <- train(learner, task)

# create an explainer for the model
explainer_mlr <- explain_mlr(model,
                             data = test,
                             y = test$fired,
                             label = "mlr")

# make a studio for the model
modelStudio(explainer_mlr)
```

References

- Theoretical introduction to the plots: Explanatory Model Analysis. Explore, Explain, and Examine Predictive Models.
- The input object is implemented in DALEX
- Feature Importance, Ceteris Paribus, Partial Dependence and Accumulated Dependence explanations are implemented in ingredients
- Break Down and Shapley Values explanations are implemented in iBreakDown