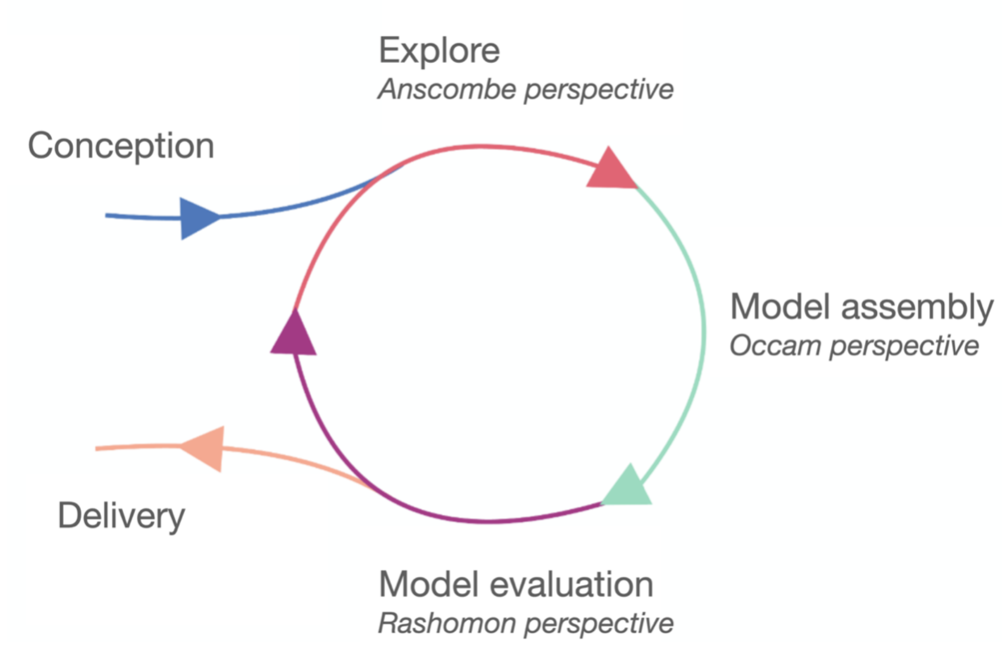
Lifecycle of a predictive model



Notes for tutorial

Introduction to Machine Learning with R

at Biometrisches Kolloquium 2021

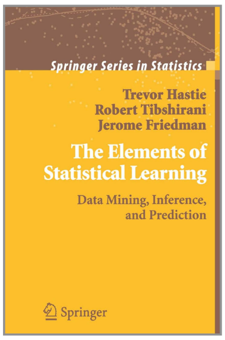
Przemysław Biecek, MI2DataLab @ WUoT

3.14 (online)

The purpose of this tutorial is to present a life cycle of a single predictive model for a problem related to binary classification (but I deliberately don't mention logistic regression). Along the way, we will tackle various interesting topics such as model training, model verification, hyperparameter tuning, and exploratory model analysis.

The examples presented here are inspired by three textbooks books: ESL[1] with mathematical foundations, MLR[2] presenting software package `mlr3` designed for advanced model programming, and EMA[3] overviews methods for model exploration and visualisation. Note that responsible modelling requires knowledge that cannot be learned in three hours. So, after this introduction, I highly recommend checking out these books.

[1] The Elements of Statistical Learning. `https://www.statlearning.com/`



Springer Series in Statistics

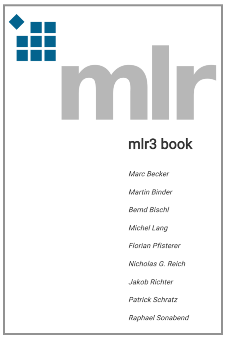
Trevor Hastie
Robert Tibshirani
Jerome Friedman

The Elements of Statistical Learning

Data Mining, Inference, and Prediction

Springer

[2] The mlr3 book. `https://mlr3book.mlr-org.com/`.

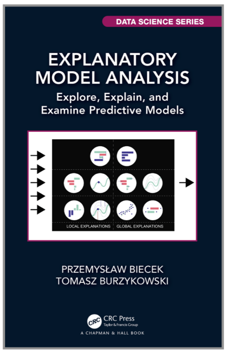


mlr

mlr3 book

Marc Becker
Martin Binder
Bernd Bischl
Michel Lang
Florian Pfisterer
Nicholas G. Reich
Jakob Richter
Patrick Schratz
Raphael Sonabend

[3] Explanatory Model Analysis. `http://ema.drwhy.ai/`



DATA SCIENCE SERIES

EXPLANATORY MODEL ANALYSIS

Explore, Explain, and Examine Predictive Models

PRZEMYSŁAW BIECEK
TOMASZ BURZYKOWSKI

CRC Press
A CHAPMAN & HALL BOOK

Why should I care?

Predictive models have been used throughout entire human history. Priests in Egypt were predicting when the flood of the Nile or a solar eclipse would come. Developments in statistics, increasing the availability of datasets, and increasing computing power allow predictive models to be built faster and faster.

Today, predictive models are used virtually everywhere. Planning the supply chain for a large corporation, recommending lunch or a movie for the evening, or predicting traffic jams in a city. Newspapers are full of interesting applications.

But how are such predictive models developed? In the following sections, we will go through a life cycle of a predictive model. From the concept phase, through design, training, checking, to the deployment. For this example, we will use the data set on the risk of death for Covid-19 patients after SARS-COV-2 infection. But keep in mind that the data presented here is artificial. It is generated to mirror relations in real data, but do not contain real observations for real patients. Still, it should be an interesting use-case to discuss a typical lifetime of a predictive model.

Tools

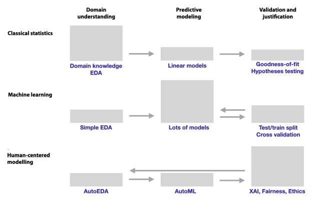
These materials are based on two R packages: `mlr3` for model training and `DALEX` for model visualization and explanation. But there are more packages with similar functionalities, for modelling other popular choices are `mlr`, `tidymodels` and `caret` while for the model explanation you will find lots of interesting features in `flashlight` and `iml`.

The problem

The life cycle of a predictive model begins with a well-defined problem. In this example, we are looking for a model that assesses the risk of death after diagnosed covid. We don't want to guess who will survive and who won't. We want to construct a score that allows us to sort patients by risk of death[4].

[4] For this reason, in the following sections we will use the AUC measure to evaluate models

Why do we need such a model? It could have many applications! Those at higher risk of death could be given more protection, such as providing them with pulse oximeters or preferentially vaccinating them.



The data

To build a model we need good data. In Machine Learning, the word *good* means a large amount of representative data. Collecting representative data is not easy and often requires designing an appropriate experiment.

The best possible scenario is that one can design and run an experiment to collect the necessary data. In less comfortable situations, we look for "natural experiments," i.e., data that have been collected for another purpose but that can be used to build a model. Here we will use the data[5] collected through epidemiological interviews. There will be a lot of data points and it should be fairly representative, although unfortunately it only involves symptomatic patients who are tested positive for SARS-COV-2.

[5] Please note that the attached data are not the real data collected for epidemiological purposes, but artificially generated data preserving the structure and relationships in the real data.

The data is divided into two sets `covid_spring` and `covid_summer`. The first is acquired in spring 2020 and will be used as training data while the second dataset is acquired in summer and will be used for validation[6].

[6] In machine learning, model validation is performed on a separate data set. This controls the risk of overfitting an elastic model to the data. If we do not have a separate set then it is generated using cross-validation, out of sample or out of time techniques.

```
covid_spring <- read.table("covid_spring.csv", sep =";", header = TRUE)
library("tableone")
table1 <- CreateTableOne(vars = colnames(covid_spring)[1:11],
                         data = covid_spring,  strata = "Death")
print(table1)
#                                    Stratified by Death
#                                     No            Yes           p
#   n                                  9487           513
#   Gender = Male (%)                  4554 (48.0)    271 (52.8)   0.037
#   Age (mean (SD))                   44.19 (18.32) 74.44 (13.27) <0.001
#   Cardiovascular.Diseases = Yes (%)   839 ( 8.8)    273 (53.2)  <0.001
#   Diabetes = Yes (%)                  260 ( 2.7)     78 (15.2)  <0.001
#   Neurological.Diseases = Yes (%)     127 ( 1.3)     57 (11.1)  <0.001
#   Kidney.Diseases = Yes (%)           111 ( 1.2)     62 (12.1)  <0.001
#   Cancer = Yes (%)                    158 ( 1.7)     68 (13.3)  <0.001
#   Hospitalization = Yes (%)          2344 (24.7)    481 (93.8)  <0.001
#   Fever = Yes (%)                    3314 (34.9)    335 (65.3)  <0.001
#   Cough = Yes (%)                    3062 (32.3)    253 (49.3)  <0.001
#   Weakness = Yes (%)                 2282 (24.1)    196 (38.2)  <0.001
```

Hello model!

We will think of a predictive model as a function that computes a certain prediction for certain input data. Usually, such a function is built automatically based on the data. But technically the model can be any function defined in any way. The first model will be based on statistics collected by the CDC[7] that determine mortality in different age groups.

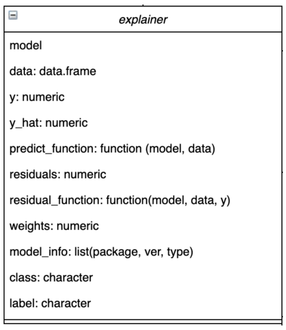
[7] CDC stands for Centers for Disease Control and Prevention. You will find a set of useful statistics related to Covid mortality on this page: https://tinyurl.com/CDCmortality

```
cdc_risk <- function(x, base_risk = 0.00003) {
  bin <- cut(x$Age, c(-Inf, 4.5, 17.5, 29.5, 39.5, 49.5,
                      64.5, 74.5, 84.5, Inf))
  relative_risk <- c(2, 1, 15, 45, 130, 400, 1100, 2800,
                      7900)[as.numeric(bin)]
  relative_risk * base_risk
}

x <- data.frame(Age = c(25, 45, 85))
cdc_risk(x)
# [1] 0.00045 0.00390 0.23700
```

First explainer[8]

[8] Explainer is an object / adapter that wraps the model and creates an uniform structure and interface for operations.

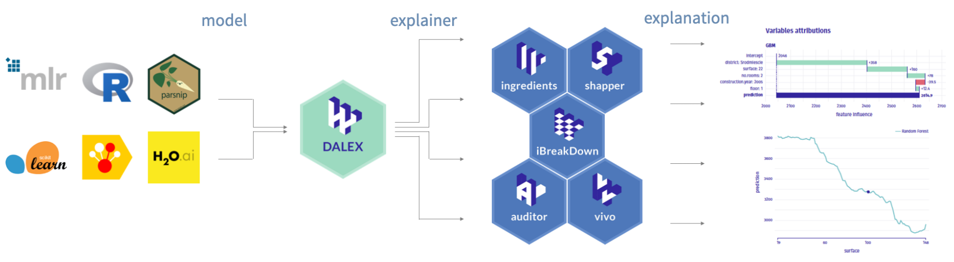


There are many libraries in R that allow you to train predictive models. Unfortunately, each of these libraries has its idea of what structure is best to store predictions for models. To work with different models in a uniform way we need to wrap models in a uniform API.

```
library("DALEX")
model_cdc <-  DALEX::explain(cdc_risk,
                   predict_function = function(m, x) m(x),
                   data  = covid_summer,
                   y     = covid_summer$Death == "Yes",
                   type  = "classification",
                   label = "CDC")
predict(model_cdc, x)
# [1] 0.00045 0.00390 0.23700
```

The DALEX library provides a unified architecture to explore and validate models using different analytical methods.

ranger(y ~ ., data = df) %>% explain() %>% model_parts() %>% plot()



How good is the model?

The evaluation of the model performance for the classification is based on different measures than for the regression.
For regression, commonly used measures are Mean squared error MSE[9] and Rooted mean squared error RMSE[10].
For classification, commonly used measures are Accuracy[11], Precision[12] and Recall[13] and F1 score[14]. In this problem we are interested in ranking of scores, so we will use the AUC measure (the area under the ROC curve).

There are many measures for evaluating predictive models and they are located in various R packages (ROCR, measures, mlr3measures, etc.). For simplicity, in this example, we use only the AUC measure from the DALEX package.

[9] $MSE(f) = \frac{1}{n}\sum_i^n (f(x_i) - y_i)^2$
[10] $RMSE(f) = \sqrt{MSE(f, X, y)}$
[11] $ACC(f) = (TP + TN)/n$
[12] $Prec(f) = TP/(TP + FP)$
[13] $Recall(f) = TP/(TP + FN)$
[14] $F1(f) = 2\frac{Prec(f)*Recall(f)}{Prec(f)+Recall(f)}$

Model performance

Model exploration starts with an assessment of how good is the model. The DALEX::model_performance function calculates a set of the most common measures for the specified model.

```
mp_cdc <- model_performance(model_cdc, cutoff = 0.1)
mp_cdc

# Measures for:  classification
# recall     : 0.2188841
# precision  : 0.2602041
# f1         : 0.2377622
# accuracy   : 0.9673
# auc        : 0.904654
#
# Residuals:
#        0%       10%       20%        30%       40%       50%
# -0.23700 -0.03300 -0.01200  -0.01200 -0.00390 -0.00390
#       60%       70%       80%        90%      100%
# -0.00135 -0.00135 -0.00045  -0.00006  0.99955
```

Note: The model is evaluated on the data given in the explainer. Use DALEX::update_data() to specify another dataset.
Note: Explainer knows whether the model is for classification or regression, so it automatically selects the right measures. It can be overridden if needed.

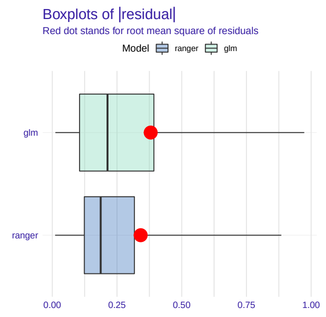


Figure 1: Boxplots for residuals

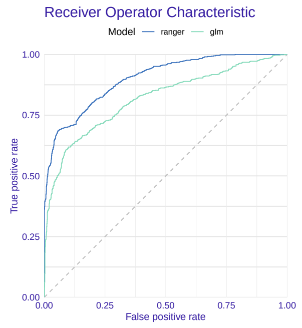


Figure 2: ROC curves

Performance charts

The S3 generic plot function draws a graphical summary of the model performance. With the geom argument, one can determine the type of chart.

```
# Boxplots
plot(mp_cdc, geom = "boxplot")
# ROC curves
plot(mp_cdc, geom = "roc")
```

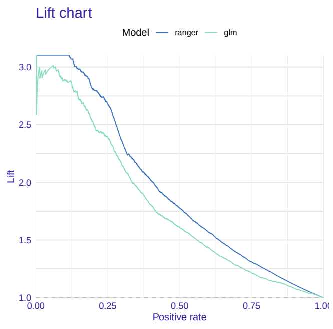


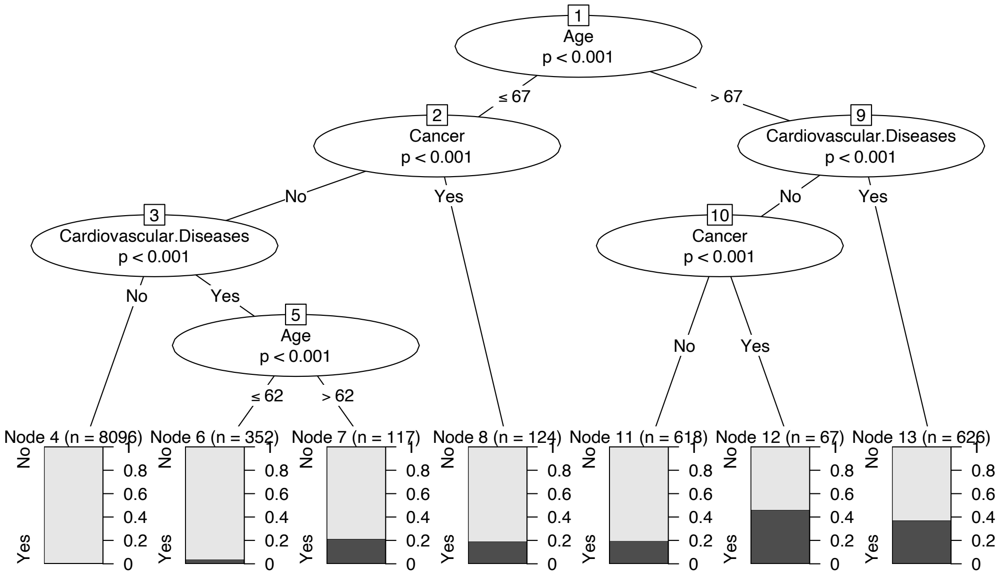
Figure 3: LIFT curves

Grow a tree

Usually, we don't know which function is the best for our problem. This is why we want to use data to find/train such function with the use of some automated algorithm. In the Machine Learning world, there are hundreds of algorithms available. Usually, this training boils down to finding parameters for some family of models. One of the most popular families of models is decision trees. Their great advantage is the transparency of their structure.

```
library("partykit")

tree <- ctree(Death ~., covid_spring,
              control = ctree_control(alpha = 0.0001))
plot(tree)
```



To work with different models uniformly, we will also wrap this one into an explainer.

```
model_tree <-  DALEX::explain(tree,
                   predict_function = function(m, x)
                              predict(m, x, type = "prob")[,2],
                   data = covid_summer,
                   y = covid_summer$Death == "Yes",
                   type = "classification",
                   label = "Tree")
(mp_tree <- model_performance(model_tree, cutoff = 0.1))

# Measures for:  classification
# recall     : 0.8626609
# precision  : 0.1492205
# f1         : 0.2544304
# accuracy   : 0.8822
# auc        : 0.9136169
```

Plant a forest

Decision trees are models that have low bias but high variance. In 2001, Leo Breiman proposed a new family of models, called a random forest, which averages scores from multiple decision trees trained on bootstrap samples of the data[15]. This algorithm reduces variance at the expense of bias. Quite often it leads to a better model.

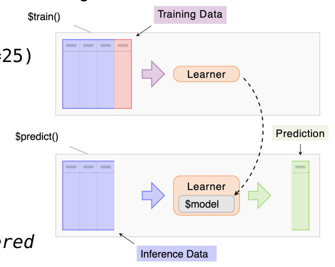
[15] The whole algorithm is a bit more complex but also very fascinating. You can read about it at `https://tinyurl.com/RF2001`. Nowadays a very popular, in a sense complementary technique for improving models is boosting, in which you reduce the model load at the expense of variance.

We will train a random forest with the `mlr3` library. The first step is to define the prediction task.

```
library("mlr3")
covid_task <- TaskClassif$new(id = "covid_spring",
                             backend = covid_spring,
                             target = "Death",
                             positive = "Yes")
covid_task
# <TaskClassif:covid_spring> (10000 x 8)
# * Target: Death
# * Properties: twoclass
# * Features (7):
#   - fct (6): Cancer, Cardiovascular.Diseases, Diabetes,
#     Gender, Kidney.Diseases, Neurological.Diseases
#   - int (1): Age
```

Now we need to define the family of models in which we want to look for a solution. The random forests is specified by the `classif.ranger`' parameter. To find the best model in this family we use the `train()`[16] function.

[16] Source: `https://mlr3book.mlr-org.com/learners.html`



```
library("mlr3learners")
library("ranger")
covid_ranger <- lrn("classif.ranger", predict_type="prob", num.trees=25)
covid_ranger$train(covid_task)
# <LearnerClassifRanger:classif.ranger>
# * Model: ranger
# * Parameters: num.trees=250
# * Packages: ranger
# * Predict Type: prob
# * Feature types: logical, integer, numeric, character, factor, ordered
# * Properties: importance, multiclass, oob_error, twoclass, weights
```

A trained model can be turned into an explainer[17].

[17] Simpler functions can be used to calculate the performance of this model. But using explainers has an advantage that will be seen in all its beauty in just two pages

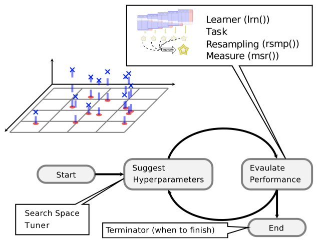
```
model_ranger <-  explain(covid_ranger,
                      predict_function = function(m,x)
                           predict(m, x, predict_type = "prob")[,1],
                      data = covid_summer,
                      y = covid_summer$Death == "Yes",
                      type = "classification",
                      label = "Ranger")

mp_ranger <- model_performance(model_ranger)
mp_ranger
# Measures for:  classification
# recall     : 0.04291845
# precision  : 0.4347826
# f1         : 0.078125
# accuracy   : 0.9764
# auc        : 0.9444727
```

Hyperparameter Optimisation

Machine Learning algorithms typically have many hyperparameters that determine how the model is to be trained. For models with high variance, the selection of such hyperparameters has a strong impact on the quality of the final solution. The mlr3tuning package contains procedures to automate the process of finding good hyperparameters[18].

[18] Source: https://mlr3book.mlr-org.com/tuning.html



To use it, you must specify the space of hyperparameter to search. Not all hyperparameters are worth optimizing. In the example below, we focus on four for the random forest algorithm.

```
library("mlr3tuning")
library("paradox")
search_space = ps(
  num.trees = p_int(lower = 50, upper = 500),
  max.depth = p_int(lower = 1, upper = 10),
  minprop = p_dbl(lower = 0.01, upper = 0.1),
  splitrule = p_fct(levels = c("gini", "extratrees"))
)
```

For automatic hyperparameter search, it is necessary to specify a few more elements: (1) a stopping criterion, below it is the number of 10 evaluations, (2) a search strategy for the parameter space, below it is a random search, (3) a way to evaluate the performance of the proposed models, below it is the AUC determined by 5-fold cross-validation.

```
tuned_ranger = AutoTuner$new(
  learner    = covid_ranger,
  resampling = rsmp("cv", folds = 5),
  measure    = msr("classif.auc"),
  search_space = search_space,
  terminator = trm("evals", n_evals = 10),
  tuner    = tnr("random_search")
)
tuned_ranger$train(covid_task)
tuned_ranger$tuning_result
#   num.trees max.depth    minprop splitrule learner_param_vals
# 1:       264         9 0.06907318      gini          <list[4]>
#     x_domain classif.auc
# 1: <list[4]>   0.9272979
```

There is, of course, no guarantee that the tuner will find better hyperparameters than the defaults[19].

[19] Moreover, some algorithms, like random forests, are not very tunable. However, we had to try!

```
model_tuned <-  explain(tuned_ranger,
                        predict_function = function(m,x)
                            m$predict_newdata(newdata = x)$prob[,1],
                        data = covid_summer,
                        y = covid_summer$Death == "Yes",
                        type = "classification",
                        label = "AutoTune")
(mp_tuned <- model_performance(model_tuned))
# Measures for:  classification
# accuracy   : 0.9763
# auc        : 0.9447171
```

Which variables have an impact on the model?

Many models have built-in ways of assessing the importance of variables. The procedure described below is universal and does not depend on the model structure.

Note that if a variable is important in a model, then after its permutation the model predictions should be less accurate. The permutation importance of a variable i is the difference between the model performance for the original data and the model performance measured on data with the permutated variable i[20].

[20] $V(f,i) = L^i_{perm}(f) - L_{org}(f)$, where $L_{org}(f)$ is the value of loss function for original data, while $L^i_{perm}(f)$ is the value of loss function after permuted i-th variable

The `DALEX::model_parts()` function calculates the importance of variables. The `type` argument determines whether to subtract (`type="difference"`), divide (`type="ratio"`) or present original loss functions (`type="raw"`).

```
mpart_ranger <- model_parts(model_ranger, type = "difference")
mp_ranger
#                  variable mean_dropout_loss  label
# 1            _full_model_        0.05742770 Ranger
# 2                  Gender        0.05558884 Ranger
# 3         Hospitalization        0.05742770 Ranger
# 4                   Fever        0.05742770 Ranger
# 5                   Cough        0.05742770 Ranger
# 6                Weakness        0.05742770 Ranger
# 7                   Death        0.05742770 Ranger
# 8    Neurological.Diseases       0.05804917 Ranger
# 9          Kidney.Diseases       0.06130847 Ranger
# 10                 Cancer        0.06536914 Ranger
# 11               Diabetes        0.06793768 Ranger
# 12 Cardiovascular.Diseases       0.07265630 Ranger
# 13                    Age        0.19095116 Ranger
# 14             _baseline_        0.48216847 Ranger

plot(mp_ranger, show_boxplots = FALSE)
```

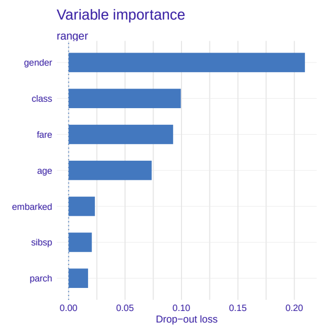


Figure 4: Importance scores $V(f,i)$

The importance of variables is a convenient tool to compare models. It is enough to put several importance objects to the generic S3 `plot()` function.

```
mpart_cdc <- model_parts(model_cdc)
mp_ranger <- model_parts(model_ranger)
plot(mpart_cdc, mp_ranger, show_boxplots = FALSE)
```

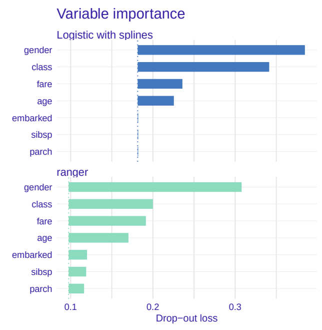
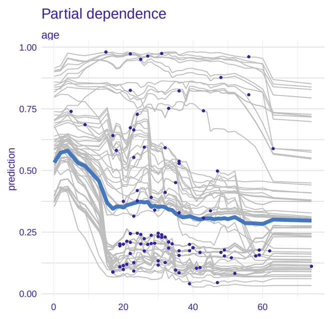


Figure 5: Variable importance plots for two models. Each bar starts in $L^i_{perm}(f)$ and ends in $L_{org}(f)$.

By default, RMSE is used for regression and 1-AUC for classification problems. But you can change the loss function by specifying the `loss_function` argument.



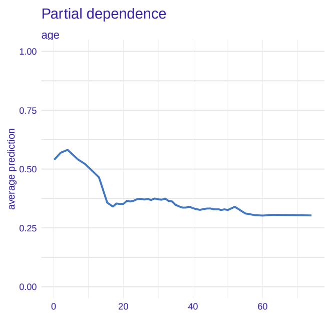


Figure 6: Partial dependence profile for Age variable.

Partial dependence profiles

Partial dependence profiles are averages from CP profiles for all (or a large enough number) observations.
The `model_profiles()` function calculates PD profiles for a specified model and variables (all by default).

```
mp_ranger <- model_profile(model_ranger, "Age")
```

Profiles can be then drawn with the `plot()` function. See an example in Figure 6.

```
plot(mp_ranger)
```

Grouped partial dependence profiles

By default, the average is calculated for all observations. But with the argument `groups=` one can specify a factor variable in which CP profiles will be averaged. See an example in Figure 7.

```
mgroup_ranger <- model_profile(model_ranger, "Age",
                        groups = "Cardiovascular.Diseases")
plot(mgroup_ranger)
```

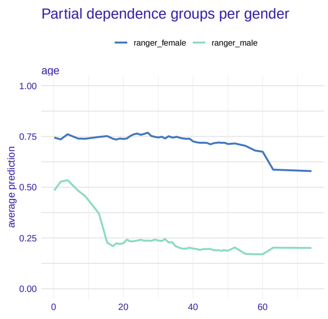


Figure 7: Partial dependence for Age in groups defined by Cardiovascular.Diseases.

Clustered partial dependence profiles

If the model is additive, all CP profiles are parallel. But if the model has interactions, CP profiles may have different shapes for different observations. Defining the k argument allows to find and calculate the average in k segments of CP profiles.

```
mgroup_ranger <- model_profile(model_ranger, "Age",
                        k = 3, center = TRUE)
plot(mgroup_ranger)
```

PDP profiles do not take into account the correlation structure between the variables. For correlated variables, the Ceteris paribus assumption may not make sense. The `model_profile` function can also calculate other types of aggregates, such as marginal profiles and accumulated local profiles. To do this, specify the argument `type=` for `"conditional"` or `"accumulated"`.

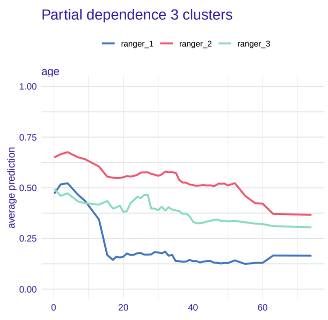


Figure 8: Partial dependence for three segments of CP profiles.

Which variables have an impact on a model prediction?

Once we calculate the model prediction, the question often arises which variables had the greatest impact on it.

```
john <- data.frame(Gender = factor("Male", levels = c("Male", "Female")),
                   Age = 76,
                   Cardiovascular.Diseases = factor("Yes", levels = c("Yes", "No")),
                   Diabetes = factor("No", levels = c("Yes", "No")),
                   Neurological.Diseases = factor("No", levels = c("Yes", "No")),
                   Kidney.Diseases = factor("No", levels = c("Yes", "No")),
                   Cancer = factor("No", levels = c("Yes", "No")))
predict(model_ranger, john)
# 0.3715257
```

For linear models it is easy to assess the impact of individual variables because there is one coefficient for each variable.
It turns out that such attributions can be calculated for any predictive model. The most popular model agnostic method is Shapley values. They may be calculated with a `predict_parts()` function.

```
ppart_ranger <- predict_parts(model_ranger, john, type = "shap")
plot(ppart_ranger, show_boxplots = FALSE)
```

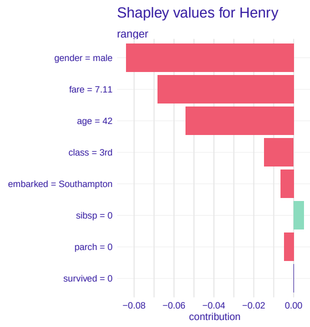


Figure 9: Shapley values assess the impact of each variable on a model prediction. The attributions add up to model prediction.

The Shapley values are additive. For models with interactions, it is often too much of a simplification. The Break Down method allows for the identification of interactions.

```
bd_ranger <- predict_parts(model_ranger, john, type  = "break_down_interactions")
bd_ranger
#                     variable contribution
# 1                  intercept   0.043116782
# 2                   Age = 76   0.162066321
# 3                Cancer = No   0.182584077
# 4         Kidney.Diseases = No   0.036967264
# 5   Neurological.Diseases = No -0.051889697
# 6              Diabetes = No   0.011161295
# 7 Cardiovascular.Diseases = Yes -0.005558188
# 8              Gender = Male -0.006922160
# 9                 prediction   0.371525695

plot(bd_ranger)
```

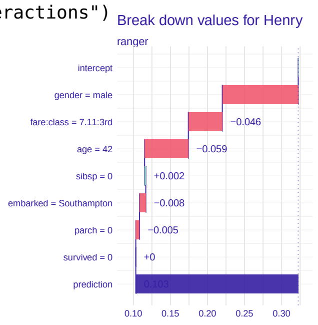


Figure 10: Break down values assess the impact of variable or their interactions on model predictions.

The `show_boxplots` argument allows you to highlight the stability bars of the estimated attributions.
Other possible values of the `type` argument are `oscillations`, `shap`, `break_down`, `break_down_interactions`.
With `order` one can force a certain sequence of variables.

```
bd_ranger <- predict_parts(model_ranger, john,
        order = c("Age", "Cardiovascular.Diseases", "Cancer", "Gender"))
plot(bd_ranger)
```

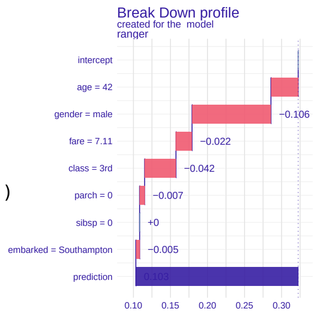


Figure 11: Break down values for selected order of variables.

By default, functions such as `model_parts`, `predict_parts`, `model_profiles` do not calculate statistics on the entire data set, but on `n_samples` of random cases, and the entire procedure is repeated `B` times to estimate the error bars.

What if?

Ceteris-paribus profiles[21] show how the model response would change for a selected observation if one of the coordinates of that observation were changed while leaving the other coordinates unchanged.
The `predict_profiles()` function calculated CP profiles for a selected observation, model and vector of variables (all continuous variables by default).

[21] Ceteris Paribus is a Latin phrase for "other things being equal.

```
mprof_cdc <- predict_profile(model_ranger, john)
mprof_cdc
#  Top profiles    :
#        Gender   Age Cardiovascular.Diseases Diabetes
# 1      Female 76.00                     Yes       No
# 1.1      Male 76.00                     Yes       No
# 11       Male  0.00                     Yes       No
# 1.110    Male  0.99                     Yes       No
```

CP profiles can be visualized with the generic `plot()` function.

```
plot(mprof_cdc, variables = "Age")
```

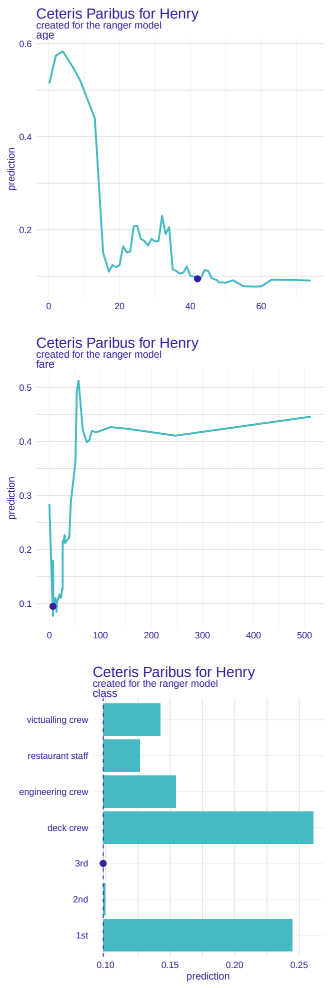


Figure 12: The dot shows the observation under analysis. CP profile shows how the model prediction will change for changes in the selected variable.

For technical reasons, quantitative and qualitative variables cannot be shown in a single chart. So if you want to show the importance of quality variables you need to plot them separately.

```
plot(mprof_cdc, variables = "Cancer", categorical_type = "bars")
```

The `plot` function can combine different models, making it easier to see similarities and differences. The `color` argument allows you to highlight models in the figure.

```
cp_rms <- predict_profile(model_cdc, john)
plot(mprof_cdc, cp_rms, variables = "Age", color = "_label_")
```

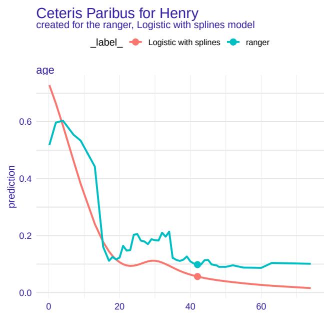


Figure 13: CP profiles for two models.

Oscillations

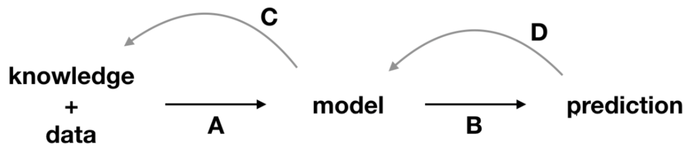
Local importance of variables can be measured as oscillations of CP plots. The greater the variability of the CP profile, the more important is the variable. Set `type = "oscillations"` in the `predict_parts` function.

```
predict_parts(model_ranger, john, type = "oscillations")
#                    _vname_ _ids_ oscillations
# 2                      Age     1  0.117472763
# 6          Kidney.Diseases     1  0.066037709
# 3 Cardiovascular.Diseases     1  0.026543173
# 5    Neurological.Diseases     1  0.024955879
# 7                   Cancer     1  0.010304796
# 1                   Gender     1  0.007029000
# 4                 Diabetes     1  0.001454713
```

What's next?

I hope you enjoyed the workshop. However, we still haven't touched on many important issues like variable filtering, variable engineering, model monitoring, debugging, auditing, etc.

I wanted to show that model exploration strengthens the feedback extracted from a model. In 2001, Leo Breiman wrote a paper called The Two Cultures that presented the gap between modellers who use models to better understand the world around them and modellers who care only about effective predictions. However, we are finding that the needs of these two groups are blending. We need both high-performance tools and tools to better understand how these black boxes work.



A) data and domain knowledge allow building the model. B) predictions are obtained from the model. C) by analyzing the predictions, we learn more about the model. D) a better understanding of the model allows a better understanding of the data and, sometimes, broadens domain knowledge.

In addition to the packages we have already discussed, many others allow you to build models responsibly. The two are part of new trends in model exploration.

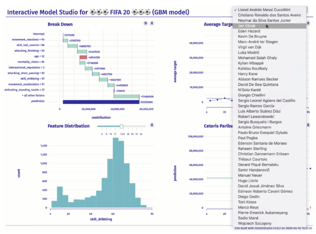


Figure 14: modelStudio dashboard

- `modelStudio` and `ArenaR` are packages based on the evolving grammar of interactive model exploration. They generate a serverless HTML dashboard (one line of code) that consists of various XAI views that can be freely combined.
- `fairmodels` is a package that implements various measures of model bias and fairness. Measures such as Equal opportunity loss, Predictive equality loss, Predictive parity loss or Statistical parity loss. It is easy for one or more explainers to investigate whether they generate unfair decisions.

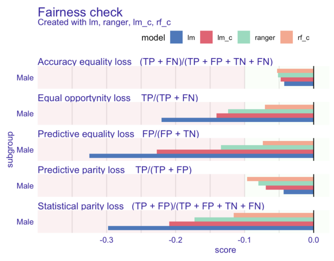


Figure 15: Fairness check

Acknowledgments

Work on the DALEX package was financially supported by the Polish National Science Centre under Opus Grant number 2017/27/B/ST6/0130.