Feature Selection

Table of Contents

Source: `vignettes/tutorial/feature_selection.Rmd` (https://github.com/mlr-org/mlr/blob/master/vignettes/tutorial/feature_selection.Rmd)

Often, data sets include a large number of features. The technique of extracting a subset of relevant features is called feature selection. Feature selection can enhance the interpretability of the model, speed up the learning process and improve the learner performance. There exist different approaches to identify the relevant features. In the literature two different approaches exist: One is called "Filtering" and the other approach is often referred to as "feature subset selection" or "wrapper methods".

What is the difference?

- **Filter**: An external algorithm computes a rank of the variables (e.g. based on the correlation to the response). Then, features are subsetted by a certain criteria, e.g. an absolute number or a percentage of the number of variables. The selected features will then be used to fit a model (with optional hyperparameters selected by tuning). This calculation is usually cheaper than "feature subset selection" in terms of computation time.
- **Feature subset selection**: Here, no ranking of features is done. Features are selected by a (random) subset of the data. Then, a model is fit and the performance is checked. This is done for a lot of feature combinations in a CV setting and the best combination is reported. This method is very computational intense as a lot of models are fitted. Also, strictly all these models would need to be tuned before the performance is estimated which would require an additional nested level in a CV setting. After all this, the selected subset of features is again fitted (with optional hyperparameters selected by tuning).

`mlr` supports both **filter methods (feature_selection.html#filter-methods)** and **wrapper methods (feature_selection.html#wrapper-methods)**.

Filter methods

Filter methods assign an importance value to each feature. Based on these values the features can be ranked and a feature subset can be selected. You can see here (filter_methods.html#current-methods) which algorithms are implemented.

Calculating the feature importance

Different methods for calculating the feature importance are built into `mlr`'s function `generateFilterValuesData() (../../reference/generateFilterValuesData.html)`. Currently, classification, regression and survival analysis tasks are supported. A table showing all available methods can be found in article filter methods (filter_methods.html).

The most basic approach is to use `generateFilterValuesData() (../../reference/generateFilterValuesData.html)` directly on a `Task() (../../reference/Task.html)` with a character string specifying the filter method.

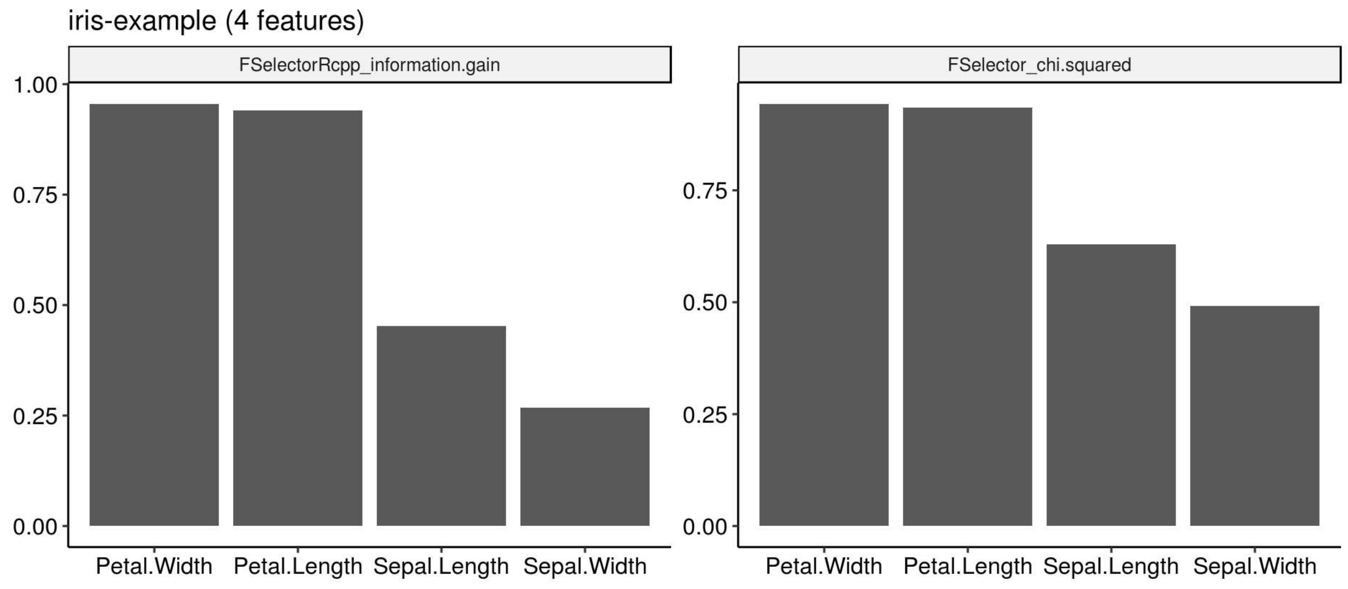
```
fv = generateFilterValuesData (../../reference/generateFilterValuesData.html)(ir
## Loading required namespace: FSelectorRcpp
fv
## FilterValues:
## Task: iris-example
##           name    type FSelectorRcpp_information.gain
## 1 Sepal.Length numeric                      0.4521286
## 2  Sepal.Width numeric                      0.2672750
## 3 Petal.Length numeric                      0.9402853
## 4  Petal.Width numeric                      0.9554360
```

`fv` is a `FilterValues() (../../reference/generateFilterValuesData.html)` object and `fv$data` contains a `data.frame` that gives the importance values for all features. Optionally, a vector of filter methods can be passed.

```
fv2 = generateFilterValuesData (../../reference/generateFilterValuesData.html)(i
  method = c("FSelectorRcpp_information.gain", "FSelector_chi.squared"))
fv2$data
##           name    type FSelectorRcpp_information.gain
## 1 Sepal.Length numeric                      0.4521286
## 2  Sepal.Width numeric                      0.2672750
## 3 Petal.Length numeric                      0.9402853
## 4  Petal.Width numeric                      0.9554360
##   FSelector_chi.squared
## 1             0.6288067
## 2             0.4922162
## 3             0.9346311
## 4             0.9432359
```

A bar plot of importance values for the individual features can be obtained using function `plotFilterValues() (../../reference/plotFilterValues.html)`.

```
plotFilterValues (../../reference/plotFilterValues.html)(fv2) + ggpubr::theme_pub
```



By default `plotFilterValues() (../../reference/plotFilterValues.html)` will create facetted subplots if multiple filter methods are passed as input to `generateFilterValuesData() (../../reference/generateFilterValuesData.html)`.

According to the `"information.gain"` measure, `Petal.Width` and `Petal.Length` contain the most information about the target variable `Species`.

Selecting a feature subset

With `mlr`s function `filterFeatures() (../../reference/filterFeatures.html)` you can create a new `Task() (../../reference/Task.html)` by leaving out features of lower importance.

There are several ways to select a feature subset based on feature importance values:

- Keep a certain **absolute number** (`abs`) of features with highest importance.
- Keep a certain **percentage** (`perc`) of features with highest importance.
- Keep all features whose importance exceeds a certain *threshold value* (`threshold`).

Function `filterFeatures() (../../reference/filterFeatures.html)` supports these three methods as shown in the following example. Moreover, you can either specify the `method` for calculating the feature importance or you can use previously computed importance values via argument `fval` .

```
# Keep the 2 most important features
filtered.task = filterFeatures (../../reference/filterFeatures.html)(iris.task, m

# Keep the 25% most important features
filtered.task = filterFeatures (../../reference/filterFeatures.html)(iris.task, f

# Keep all features with importance greater than 0.5
filtered.task = filterFeatures (../../reference/filterFeatures.html)(iris.task, f
filtered.task
## Supervised task: iris-example
## Type: classif
## Target: Species
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           2           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

Fuse a learner with a filter method

Often feature selection based on a filter method is part of the data preprocessing and in a subsequent step a learning method is applied to the filtered data. In a proper experimental setup you might want to automate the selection of the features so that it can be part of the validation method of your choice. A Learner (`makeLearner() (../../reference/makeLearner.html)`) can be fused with a filter method by function `makeFilterWrapper() (../../reference/makeFilterWrapper.html)` . The resulting Learner (`makeLearner() (../../reference/makeLearner.html)`) has the additional class attribute `FilterWrapper()` . This has the advantage that the filter parameters (`fw.method` , fw.perc. fw.abs) can now be treated as hyperparameters. They can be tuned in a nested CV setting at the same level as the algorithm hyperparameters. You can think of if as "tuning the dataset".

Using fixed parameters

In the following example we calculate the 10-fold cross-validated error rate mmce (measures.html) of the k-nearest neighbor classifier (`FNN::fnn()`) with preceding feature selection on the `iris` (`datasets::iris()` (`http://www.rdocumentation.org/packages/datasets/topics/iris`)) data set. We use `information.gain` as importance measure with the aim to subset the dataset to the two features with the highest importance. In each resampling iteration feature selection is carried out on the corresponding training data set before fitting the learner.

```
lrn = makeFilterWrapper (../../reference/makeFilterWrapper.html)(learner = "class
  fw.method = "FSelectorRcpp_information.gain", fw.abs = 2)
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 10
r = resample (../../reference/resample.html)(learner = lrn, task = iris.task, res
r$aggr
## mmce.test.mean
##           0.04
```

You may want to know which features have been used. Luckily, we have called `resample()` (`../../reference/resample.html`) with the argument `models = TRUE`, which means that `r$models` contains a `list` of models (`makeWrappedModel()` (`../../reference/makeWrappedModel.html`)) fitted in the individual resampling iterations. In order to access the selected feature subsets we can call `getFilteredFeatures()` (`../../reference/getFilteredFeatures.html`) on each model.

```
sfeats = sapply(r$models, getFilteredFeatures)
table(sfeats)
## sfeats
## Petal.Length  Petal.Width
##           10           10
```

The result shows that in the ten folds always `Petal.Length` and `Petal.Width` have been chosen (remember we wanted to have the best two, i.e. 10×2). The selection of features seems to be very stable for this dataset. The features `Sepal.Length` and `Sepal.Width` did not make it into a single fold.

Tuning the size of the feature subset

In the above examples the number/percentage of features to select or the threshold value have been arbitrarily chosen. However, it is usually unclear which subset of features will results in the best performance. To answer this question, we can tune (tune.html) the number of features that are taken (after the ranking of the chosen algorithms was applied) as a subset in each fold. Three tunable parameters exist in `mlr`, documented in `makeFilterWrapper()` (`../../reference/makeFilterWrapper.html`):

- The percentage of features selected (`fw.perc`)
- The absolute number of features selected (`fw.abs`)
- The threshold of the filter method (`fw.threshold`)

In the following regression example we consider the `BostonHousing` (`mlbench::BostonHousing()` (`http://www.rdocumentation.org/packages/mlbench/topics/BostonHousing`)) data set. We use a Support Vector Machine and determine the optimal percentage value for feature selection such that the 3-fold cross-validated mean squared error (`mse()` (`../../reference/measures.html`)) of the learner is minimal. Additionally, we tune (tune.html) the hyperparameters of the algorithm at the same time. As search strategy for tuning a random search with five iterations is used.

```
lrn = makeFilterWrapper (../../reference/makeFilterWrapper.html)(learner = "regr
ps = makeParamSet(makeNumericParam("fw.perc", lower = 0, upper = 1),
                  makeNumericParam("C", lower = -10, upper = 10,
                    trafo = function(x) 2^x),
                  makeNumericParam("sigma", lower = -10, upper = 10,
                    trafo = function(x) 2^x)
                  )
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 3
res = tuneParams (../../reference/tuneParams.html)(lrn, task = bh.task, resampli
  control = makeTuneControlRandom (../../reference/makeTuneControlRandom.html)(m
## [Tune] Started tuning learner regr.ksvm.filtered for parameter set:
##             Type len Def    Constr Req Tunable Trafo
## fw.perc numeric   -   -    0 to 1   -    TRUE     -
## C       numeric   -   - -10 to 10   -    TRUE     Y
## sigma   numeric   -   - -10 to 10   -    TRUE     Y
## With control class: TuneControlRandom
## Imputation value: Inf
## [Tune-x] 1: fw.perc=0.846; C=0.0261; sigma=17.1
## [Tune-y] 1: mse.test.mean=86.1394358; time: 0.0 min
## [Tune-x] 2: fw.perc=0.108; C=47.9; sigma=0.0335
## [Tune-y] 2: mse.test.mean=63.8491956; time: 0.0 min
## [Tune-x] 3: fw.perc=0.548; C=0.00156; sigma=69.7
## [Tune-y] 3: mse.test.mean=86.3187611; time: 0.0 min
## [Tune-x] 4: fw.perc=0.338; C=321; sigma=0.138
## [Tune-y] 4: mse.test.mean=21.3827291; time: 0.0 min
## [Tune-x] 5: fw.perc=0.975; C=0.201; sigma=0.855
## [Tune-y] 5: mse.test.mean=58.7790094; time: 0.0 min
## [Tune] Result: fw.perc=0.338; C=321; sigma=0.138 : mse.test.mean=21.3827291
res
## Tune result:
## Op. pars: fw.perc=0.338; C=321; sigma=0.138
## mse.test.mean=21.3827291
```

The performance of all percentage values visited during tuning is:

```
df = as.data.frame(res$opt.path)
df[, -ncol(df)]
##     fw.perc         C      sigma mse.test.mean dob eol error.message
## 1 0.8461958 -5.262548  4.0975570      86.13944   1  NA          <NA>
## 2 0.1076088  5.581377 -4.8978863      63.84920   2  NA          <NA>
## 3 0.5480433 -9.321970  6.1221895      86.31876   3  NA          <NA>
## 4 0.3377528  8.325717 -2.8578716      21.38273   4  NA          <NA>
## 5 0.9753650 -2.312430 -0.2265293      58.77901   5  NA          <NA>
```

The optimal percentage and the corresponding performance can be accessed as follows:

```
res$x
## $fw.perc
## [1] 0.3377528
##
## $C
## [1] 320.8415
##
## $sigma
## [1] 0.1379415
res$y
## mse.test.mean
##      21.38273
```

After tuning we can generate a new wrapped learner with the optimal percentage value for further use (e.g. to predict to new data).

```
lrn = makeFilterWrapper (../../reference/makeFilterWrapper.html)(learner = "regr.
        fw.perc = res$x$fw.perc, C = res$x$C, sigma = res$x$sigma)
mod = train (../../reference/train.html)(lrn, bh.task)
mod
## Model for learner.id=regr.lm.filtered; learner.class=FilterWrapper
## Trained on: task.id = BostonHousing-example; obs = 506; features = 13
## Hyperparameters: fw.method=FSelector_ch... ,fw.perc=0.338

getFilteredFeatures (../../reference/getFilteredFeatures.html)(mod)
## [1] "crim"  "dis"   "rad"   "lstat"
```

Wrapper methods

Wrapper methods use the performance of a learning algorithm to assess the usefulness of a feature set. In order to select a feature subset a learner is trained repeatedly on different feature subsets and the subset which leads to the best learner performance is chosen.

In order to use the wrapper approach we have to decide:

- How to assess the performance: This involves choosing a performance measure that serves as feature selection criterion and a resampling strategy.
- Which learning method to use.
- How to search the space of possible feature subsets.

The search strategy is defined by functions following the naming convention `makeFeatSelControl<search_strategy`. The following search strategies are available:

- Exhaustive search `makeFeatSelControlExhaustive` (`?FeatSelControl()`),
- Genetic algorithm `makeFeatSelControlGA` (`?FeatSelControl()`),
- Random search `makeFeatSelControlRandom` (`?FeatSelControl()`),
- Deterministic forward or backward search `makeFeatSelControlSequential` (`?FeatSelControl()`).

Select a feature subset

Feature selection can be conducted with function `selectFeatures()` `(../../reference/selectFeatures.html)`.

In the following example we perform an exhaustive search on the `Wisconsin Prognostic Breast Cancer` (`TH.data::wpbc()` `(http://www.rdocumentation.org/packages/TH.data/topics/wpbc)`) data set. As learning method we use the `Cox proportional hazards model` (`survival::coxph()` `(http://www.rdocumentation.org/packages/survival/topics/coxph)`). The performance is assessed by the holdout estimate of the concordance index cindex (measures.html)).

```
# Specify the search strategy
ctrl = makeFeatSelControlRandom (../../reference/FeatSelControl.html)(maxit = 20L
ctrl
## FeatSel control: FeatSelControlRandom
## Same resampling instance: TRUE
## Imputation value: <worst>
## Max. features: <not used>
## Max. iterations: 20
## Tune threshold: FALSE
## Further arguments: prob=0.5
```

`ctrl` is a `FeatSelControl() (../../reference/FeatSelControl.html)` object that contains information about the search strategy and potential parameter values.

```
# Resample description
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Holdout")

# Select features
sfeats = selectFeatures (../../reference/selectFeatures.html)(learner = "surv.co
  control = ctrl, show.info = FALSE)
sfeats
## FeatSel result:
## Features (15): mean_perimeter, mean_smoothness, mean_compactne...
## cindex.test.mean=0.7014085
```

`sfeats` is a `FeatSelResult` (`selectFeatures()` (`../../reference/selectFeatures.html`)) object. The selected features and the corresponding performance can be accessed as follows:

```
sfeats$x
##  [1] "mean_perimeter"      "mean_smoothness"      "mean_compactness"
##  [4] "mean_concavepoints"  "SE_radius"            "SE_area"
##  [7] "SE_compactness"      "SE_concavepoints"     "SE_symmetry"
## [10] "SE_fractaldim"       "worst_texture"        "worst_smoothness"
## [13] "worst_compactness"   "worst_concavepoints" "tsize"
sfeats$y
## cindex.test.mean
##        0.7014085
```

In a second example we fit a simple linear regression model to the `BostonHousing` (`mlbench::BostonHousing()` (`http://www.rdocumentation.org/packages/mlbench/topics/BostonHousing`)) data set and use a sequential search to find a feature set that minimizes the mean squared error mse (measures.html)). `method = "sfs"` indicates that we want to conduct a sequential forward search where features are added to the model until the performance cannot be improved anymore. See the documentation page `makeFeatSelControlSequential` (`?FeatSelControl()`) for other available sequential search methods. The search is stopped if the improvement is smaller than `alpha = 0.02` .

```
# Specify the search strategy
ctrl = makeFeatSelControlSequential (../../reference/FeatSelControl.html)(method

# Select features
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 1(
sfeats = selectFeatures (../../reference/selectFeatures.html)(learner = "regr.lm'
  show.info = FALSE)
sfeats
## FeatSel result:
## Features (11): crim, zn, chas, nox, rm, dis, rad, tax, ptratio...
## mse.test.mean=23.5662834
```

Further information about the sequential feature selection process can be obtained by function `analyzeFeatSelResult()` `(../../reference/analyzeFeatSelResult.html)`.

```
analyzeFeatSelResult (../../reference/analyzeFeatSelResult.html)(sfeats)
## Features         : 11
## Performance      : mse.test.mean=23.5662834
## crim, zn, chas, nox, rm, dis, rad, tax, ptratio, b, lstat
##
## Path to optimum:
## - Features:    0  Init   :                       Perf = 84.98  Diff: NA  *
## - Features:    1  Add    : lstat                 Perf = 39.018  Diff: 45.962
## - Features:    2  Add    : rm                    Perf = 31.119  Diff: 7.8991
## - Features:    3  Add    : ptratio               Perf = 27.914  Diff: 3.2056
## - Features:    4  Add    : b                     Perf = 27.189  Diff: 0.72476
## - Features:    5  Add    : dis                   Perf = 26.271  Diff: 0.91791
## - Features:    6  Add    : nox                   Perf = 25.138  Diff: 1.1332
## - Features:    7  Add    : chas                  Perf = 24.765  Diff: 0.37276
## - Features:    8  Add    : zn                    Perf = 24.472  Diff: 0.29292
## - Features:    9  Add    : crim                  Perf = 24.334  Diff: 0.13811
## - Features:   10  Add    : rad                   Perf = 24.034  Diff: 0.29951
## - Features:   11  Add    : tax                   Perf = 23.566  Diff: 0.46817
##
## Stopped, because no improving feature was found.
```

Fuse a learner with feature selection

A Learner (`makeLearner()` `(../../reference/makeLearner.html)`) can be fused with a feature selection strategy (i.e., a search strategy, a performance measure and a resampling strategy) by function `makeFeatSelWrapper()` `(../../reference/makeFeatSelWrapper.html)`. During training features are selected according to the specified selection scheme. Then, the learner is trained on the selected feature subset.

```
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 3
lrn = makeFeatSelWrapper (../../reference/makeFeatSelWrapper.html)("surv.coxph",
  control = makeFeatSelControlRandom (../../reference/FeatSelControl.html)(maxit
mod = train (../../reference/train.html)(lrn, task = wpbc.task)
mod
## Model for learner.id=surv.coxph.featsel; learner.class=FeatSelWrapper
## Trained on: task.id = wpbc-example; obs = 194; features = 32
## Hyperparameters:
```

The result of the feature selection can be extracted by function `getFeatSelResult()` `(../../reference/getFeatSelResult.html)`.

```
sfeats = getFeatSelResult (../../reference/getFeatSelResult.html)(mod)
sfeats
## FeatSel result:
## Features (17): mean_radius, mean_texture, mean_smoothness, mea...
## cindex.test.mean=0.6796954
```

The selected features are:

```
sfeats$x
##  [1] "mean_radius"        "mean_texture"       "mean_smoothness"
##  [4] "mean_compactness"   "mean_concavepoints" "mean_symmetry"
##  [7] "SE_perimeter"       "SE_area"            "SE_compactness"
## [10] "SE_concavity"       "SE_concavepoints"   "SE_symmetry"
## [13] "worst_texture"      "worst_smoothness"   "worst_compactness"
## [16] "worst_concavity"    "pnodes"
```

The 5-fold cross-validated performance of the learner specified above can be computed as follows:

```
out.rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters

r = resample (../../reference/resample.html)(learner = lrn, task = wpbc.task, res
  show.info = FALSE)
r$aggr
## cindex.test.mean
##        0.6861781
```

The selected feature sets in the individual resampling iterations can be extracted as follows:

```
lapply(r$models, getFeatSelResult)
## [[1]]
## FeatSel result:
## Features (18): mean_radius, mean_perimeter, mean_compactness, ...
## cindex.test.mean=0.5382065
##
## [[2]]
## FeatSel result:
## Features (18): mean_radius, mean_perimeter, mean_area, mean_sm...
## cindex.test.mean=0.6349051
##
## [[3]]
## FeatSel result:
## Features (20): mean_texture, mean_smoothness, mean_concavity, ...
## cindex.test.mean=0.6812985
##
## [[4]]
## FeatSel result:
## Features (11): mean_perimeter, mean_concavity, mean_concavepoi...
## cindex.test.mean=0.6924829
##
## [[5]]
## FeatSel result:
## Features (14): mean_area, mean_smoothness, mean_fractaldim, SE...
## cindex.test.mean=0.6701811
```

Feature importance from trained models

Some algorithms internally compute a feature importance during training. By using `getFeatureImportance()` (`../../reference/getFeatureImportance.html`) it is possible to extract this part from the trained model.

```
task = makeClassifTask (../../reference/Task.html)(data = iris, target = "Species
lrn = makeLearner (../../reference/makeLearner.html)("classif.ranger", importance
mod = train (../../reference/train.html)(lrn, task)

getFeatureImportance (../../reference/getFeatureImportance.html)(mod)
## FeatureImportance:
## Task: iris
##
## Learner: classif.ranger
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1   0.03898548 0.007192708    0.3138316   0.2955986
```