# ST2195 Programming for Data Science
# Block 9 Machine Learning Frameworks

## VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-9

## Data Science Workflow

Recall the workflow of a data science project that typically involves the following steps.

1. Importing the data, cleaning them and preparing them for use (data wrangling)
2. Get some basic level of understanding on the data by exploration and visualisation.
3. Consider several models as well as combinations thereof to separate signal from noise.
4. Compare models and inform future decisions.

In this and the next week we will touch upon the last two steps. Note that this essentially constitutes what is also known as *machine learning*, one of the most substantial parts of data science. Here we will briefly mention some key concepts; for more on that consider taking the course **ST3189 Machine Learning**.

## Machine Learning: Definition and Applications

The recent increase in collecting and using data has resulted in widespread interest in data analysis and *machine learning*. There is no unique definition of machine learning; different sources offer different explanations. One version of those defines machine learning as a set of methods that can automatically detect patterns in data and use those patterns to make predictions (Murphy, 2013).

There are numerous real-world applications of machine learning. One of its big successes is image recognition, which is widely used nowadays for operations such as scanning barcodes in a supermarket. Another big area of success is movie and product recommendation systems on platforms such as Netflix or Amazon, internet search engines etc. Other areas are fraud detection in transactions or identifying spam emails. This is by no means an extensive list with machine learning emerging in several areas such as medical imaging, biology and genetics, artificial intelligence and robotics and recently finance.

# Elements of a Machine Learning Model

A *machine learning model*, or *learner*, can be described as an algorithm that inputs data and outputs actions or predictions based on its parameters. The model is designed to automatically produce an output from input data. In order to calibrate a model we need to apply it to some data and set up a *cost function* that measures how far the predictions from the observations are. The overall aim in machine learning is to identify the parameters and the models that minimise this cost function. Nevertheless, this task is more complex than an optimisation problem as we do not necessarily want to optimise the performance on the observed data but on future versions of them.

The parameters of a machine learning model can involve choices made on the data pre-processing steps. For example, the data could be scaled (subtract the sample mean and divide with the standard deviation), and missing values can be imputed with different strategies. Some of the strategies for imputing missing values are listed below:

1. For each missing point impute the mean, median or the mode of each variable based on the observed data.
2. For each missing point impute a randomly sampled value from the observed data.
3. As in 2 above, but rather than sampling with equal probabilities, sample based on a histogram
4. Do not impute and remove the cases with missing values entirely.

From a programming point of view, when data pre-processing steps are merged with machine learning models, we get a new more extended model termed as *machine learning pipeline*.

# Types of Machine Learning Models

Depending on the type of problem, different types of machine learning models can be used to make predictions. The main types are supervised and unsupervised learning, although recently other types of learning have emerged such as reinforcement learning. We give a brief description of those below:

- *Supervised learning*: For cases where there is a single variable that we want to predict, termed as *target* or *response* or *dependent variable*, based on several other ones, termed as *features* or *covariates* or *independent variables*.
- *Unsupervised learning*: For cases when there is no single target or response variable, where we are interested in exploring the variables together with equal importance on all of them. In such cases we may be interested in identifying groups of variables that share something in common, known as *factor analysis*, or groups of individuals that have something in common, known as *clustering*. Analysis of *networks* also falls into this category.

In the next two weeks we will focus on supervised learning by working through several examples. We can further divide supervised learning into two categories depending on the type of the response variable: When the response is a continuous variable the task is termed as *regression*, whereas when it is categorical it is termed as *classification*. There are several machine learning models for those tasks, below is just an indicative list:

- Linear regression (regression only)
- Logistic regression (classification only)
- Lasso/Ridge regression (regression only)
- Penalised logistic regression (classification only)
- Regression and classification trees
- Random forests

- Gradient boosting
- Deep or shallow neural networks
- Gaussian process regression and classification
- Support vector machines (Classification only)

Most of these methods are covered in the course **ST3189: Machine Learning**. Here we will just use them.

The most standard cost function used for Regression task is the *mean squared error (MSE)*. This is obtained by subtracting each prediction from the actual point, taking the square of this difference and averaging all these squares. For the case of classifications see the notes of next week.

# Machine Learning Workflow

## Train, Validation and Test Sets

The procedure in machine learning is determined to a large extent by (usually) randomly splitting the data into two or three sets.

1. *Train set*: Data to be used to train each model. After training the models we can contrast their predictions of the target against the actual observations in this set. This provides the *train error* which is a measure of the *in sample* performance of each model.
2. *Validation set*: The trained models from the previous step can also provide predictions for the validation set point based the features of this set. Comparing predictions against actual points for the target provides the *test error*, which is a measure of the *out of sample* performance. We typically prefer the model with the lowest test error.
3. *Test set*: It turns out that the test error calculated on the validation set is good for choosing models but downward biased if the aim is to estimate the resulting test error. This is because the choice of the optimal models is made in sight of the supposedly unseen data of the validation set. This is why we often set aside some entirely unseen data in order to apply our chosen model in the end and obtain a more reliable value for the test error.

The above choice often depends on the sample size; if it is not big enough the validation and test sets are often merged into a single set called test set.

## Cross-Validation

A potential issue with having a single train-validation-test split is that we may end of having a fortunate or unfortunate split for some models due to chance. To limit this possibility, this process can be done repeatedly, in other words we can use *cross-validation*. The repeated partitions can be done in various ways.

One of the most widely used techniques is *K-fold* cross-validation: The data is randomly split into K subsamples. Then, each of these subsamples is set as test set in turn, with the remaining K−1 subsamples being used for training purposes. This process is repeated K times and the results are aggregated, e.g. averaged. See below for a schematic representation.

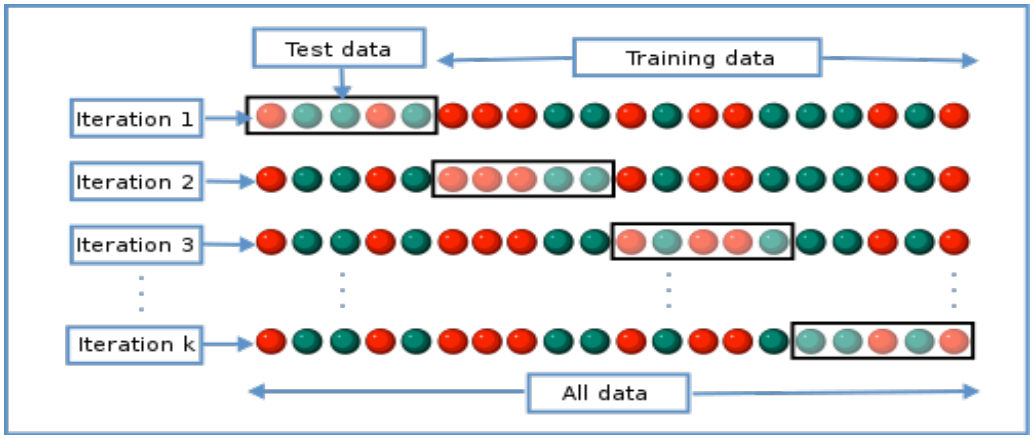



Diagram of K-fold cross-validation. Source: Wikipedia.

Other examples include repeated random subsampling (randomly split in train and test sets many times and aggregate), *leave-one-out* or *Jackknife* (K-fold cross-validation with K=1), etc.

## Machine Learning Steps

To sum up the machine learning framework involves the following steps (resembling the Data Science workflow we outlined earlier):

1. Import and pre-process the data (data wrangling). The decisions made in these steps can be thought of as model parameters.
2. Consider different models/learners, train them and compare them with the validation set.
3. Obtain predictions from the optimal model/learner chosen in the previous steps and check its performance.

# Useful Links and Resources

- 
  - ISLR book for a gentle introduction to machine learning

# References

Murphy, K. P. (2013). *Machine learning : A probabilistic perspective*. MIT Press.

# Machine Learning Frameworks in R

** Note: The code chunks below should be run in the following order **

# Data

In this page the aim is to illustrate the main concepts of machine learning as well as working with *pipelines*. We will do so using the **mlr3** R package and work on the Boston dataset, which consists of the following 14 variables:

| Variable | Description |
|---|---|
| crim | per capita crime rate by town |
| zn | proportion of residential land zoned for lots over 25,000 square feet |
| indus | proportion of non-retail business acres per town. |
| chas | Charles River dummy variable (1 if tract bounds river; 0 otherwise) |
| nox | nitric oxides concentration (parts per 10 million) |
| rm | average number of rooms per dwelling |
| age | proportion of owner-occupied units built prior to 1940 |
| dis | weighted distances to five Boston employment centres |
| rad | index of accessibility to radial highways |
| tax | full-value property-tax rate per $10,000 |
| ptratio | pupil-teacher ratio by town |
| black | 1000(Bk - 0.63)^2 where Bk is the proportion of Black ethnicity residents by town |
| lstat | % lower status of the population |
| medv | median value of owner-occupied homes in $1000's |

See Harrison & Rubinfeld (1978) for more information on the aim the data were used initially. Here we will focus on developing competitive machine learning techniques to predict the value of an owner-occupied home in Boston, based on the features (variables) above. This is a supervised learning task.

As a first step, we load and inspect the data from the `MASS` library.

```
library('MASS')
df<-Boston
head(df)
```

```
     crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat
1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98
2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14
3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03
4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94
5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33
6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21
  medv
```

```
1 24.0
2 21.6
3 34.7
4 33.4
5 36.2
6 28.7
```

To illustrate the use of a machine learning pipeline, we will also change the first value of the `crim` variable into a missing value. The use of pipelines will then be essential. We use the **skimr** package to produce a report on the data; the presence of the missing value is confirmed there.

```
library(skimr)
df$crim[1]<-NA
skim(df)
```

Data summary

| | |
|---|---|
| Name | df |
| Number of rows | 506 |
| Number of columns | 14 |
| Column type frequency: | |
| numeric | 14 |
| Group variables | None |

**Variable type: numeric**

| skim_variable | n_missing | complete_rate | mean | sd | p0 | p25 | p50 | p75 | p100 | hist |
|---|---|---|---|---|---|---|---|---|---|---|
| crim | 1 | 1 | 3.62 | 8.61 | 0.01 | 0.08 | 0.26 | 3.68 | 88.98 | ▇▁▁▁▁ |
| zn | 0 | 1 | 11.36 | 23.32 | 0.00 | 0.00 | 0.00 | 12.50 | 100.00 | ▇▁▁▁▁ |
| indus | 0 | 1 | 11.14 | 6.86 | 0.46 | 5.19 | 9.69 | 18.10 | 27.74 | ▇▆▁▆▁ |
| chas | 0 | 1 | 0.07 | 0.25 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | ▇▁▁▁▁ |
| nox | 0 | 1 | 0.55 | 0.12 | 0.38 | 0.45 | 0.54 | 0.62 | 0.87 | ▇▇▅▃▁ |
| rm | 0 | 1 | 6.28 | 0.70 | 3.56 | 5.89 | 6.21 | 6.62 | 8.78 | ▁▂▇▂▁ |
| age | 0 | 1 | 68.57 | 28.15 | 2.90 | 45.02 | 77.50 | 94.07 | 100.00 | ▂▂▂▃▇ |
| dis | 0 | 1 | 3.80 | 2.11 | 1.13 | 2.10 | 3.21 | 5.19 | 12.13 | ▇▅▂▁▁ |
| rad | 0 | 1 | 9.55 | 8.71 | 1.00 | 4.00 | 5.00 | 24.00 | 24.00 | ▇▂▁▁▂ |
| tax | 0 | 1 | 408.24 | 168.54 | 187.00 | 279.00 | 330.00 | 666.00 | 711.00 | ▇▆▂▃▇ |
| ptratio | 0 | 1 | 18.46 | 2.16 | 12.60 | 17.40 | 19.05 | 20.20 | 22.00 | ▁▂▃▆▇ |
| black | 0 | 1 | 356.67 | 91.29 | 0.32 | 375.38 | 391.44 | 396.22 | 396.90 | ▁▁▁▁▇ |
| lstat | 0 | 1 | 12.65 | 7.14 | 1.73 | 6.95 | 11.36 | 16.96 | 37.97 | ▇▆▅▂▁ |
| medv | 0 | 1 | 22.53 | 9.20 | 5.00 | 17.02 | 21.20 | 25.00 | 50.00 | ▂▇▅▁▁ |

# Machine Lerning Pipelines in **mlr3**

The **mlr3** provides a general platform to train and evaluate a wide range of machine learning models. It also allows to set up machine learning pipelines in a very convenient manner. Below we will demonstrate the key steps for doing so.

## Training and Evaluating a (Machine) Learner

The first step is to set up a *task* such as doing regression or classification on a particular dataset and specifying the response variable.

```
library('mlr3')

task <- TaskRegr$new('boston', backend=df, target = 'medv')

measure <- msr('regr.mse')
```

Next, a learner needs be to chosen. We start with linear regression `regr.lm` which is essentially the `lm()` function in R.

```
library('mlr3learners')

learner_lm <- lrn('regr.lm')
```

If we were to try and fit `lm()` to the data, the cases with missing values will be removed entirely which may not be the best way to proceed. Other learners may even return an error. So it is essential to either remove the cases with missing values or perform some imputation. Below we go with `imputemean`, i.e. imputing the mean of the variable's other values and include this as part of the learner. Other learners can be constructed based on other options and compared to each other, for example:

- `imputemedian` to impute the median of variable's other values
- `imputesample` to impute a sample from variable's other values
- `imputehist` to impute a sample from the histogram of variable's other values

In order to join together the operations of imputing missing values and fitting the linear regression model, **mlr3** creates a pipe for each of these tasks and connects them with the operator `%>>%` from **mlr3pipelines**. The resulting pipeline is termed as a *graph*, denoted by `gr_lm` in the code below, and can be converted in a learner by using `GraphLearner.new()`:

```
library('mlr3pipelines')

gr_lm <- po('imputemean') %>>%

  po(learner_lm)

glrn_lm <- GraphLearner$new(gr_lm)
```

Now we can view the combined learner `glrn_lm` as a separate machine learning algorithm. To evaluate its performance we can proceed by

1. Training it in the training data,
2. Extracting its predictions for the test data
3. Evaluating its performance by contrasting the predictions against the actual data via the model evaluation metric (`mse` in our case)

This can all be done with the code below that returns the value of the `mse`.

```
set.seed(1)
train_set <- sample(task$nrow, 0.7 * task$nrow)
test_set <- setdiff(seq_len(task$nrow), train_set)
glrn_lm$train(task, row_ids = train_set)
glrn_lm$predict(task, row_ids = test_set)$score()
regr.mse
 27.2937
```

One of the advantages of working with pipelines, as above, is that they are quite convenient for guarding against issues such as *data leaking*. In other words it ensures that the imputation is done based entirely on information contained on the training dataset; this would not be the case if we were to impute the value in the beginning on the basis of the entire dataset.

# Ridge Regression: Tuning Hyperparameters

In the previous example there were no hyperparameters but this is rather the exception than the rule when dealing with machine learning models. Let us consider another example, namely *ridge regression*. The learner `regr.glmnet` is now used, which essentially calls the **glmnet** package under the assumption that is installed. In other words, there is no need to include `library(glmnet)` but if `glmnet` is not installed, an error will be returned. Setting `alpha = 0` does ridge regression whereas `alpha = 1` corresponds to *lasso regression*. If you are curious about ridge and lasso regression you can check the relevant section in James et al. (2014), although this is beyond the scope of this course. We will proceed with `alpha = 0` but feel free to experiment with `alpha = 1`. In both cases, a value for `lambda` needs to be provided and we will use `lambda = 0.03` below for illustration purposes.

As before we incorporate missing values imputation in the procedure. We also add the operation of scaling the features (subtracting their sample mean and dividing with their standard deviation) which is generally recommended in ridge regression. Again, this is something that can cause data leaking issues if it was to be done on the entire dataset but if done in the following way it will only use information from the training data.

```
learner_ridge <- lrn('regr.glmnet')
learner_ridge$param_set$values <- list(alpha = 0, lambda = 0.03)
gr_ridge <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_ridge)
glrn_ridge<- GraphLearner$new(gr_ridge)
glrn_ridge$train(task, row_ids = train_set)
glrn_ridge$predict(task, row_ids = test_set)$score()
regr.mse
27.27267
```

The above give a slightly lower `mse` than before. But it relied on the artificial value `lambda = 0.03` which may not represent the best option. To determine its optimal value we can try several different values and choose the optimal via cross-validation. This process is known as *tuning hyperparameters*.

First we need to create a slightly different version of the previous learner. More specifically the values of `lambda` will not be fixed beforehand but it will be a *free* parameter to be optimised. As before, we create a pipeline by incorporating imputation and scaling. This gives another learner, termed `glrn_ridge2`.

```
learner_ridge2 <- lrn('regr.glmnet')
learner_ridge2$param_set$values <- list(alpha = 0)
gr_ridge2 <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_ridge2)
glrn_ridge2 <- GraphLearner$new(gr_ridge2)
```

The next steps are the following:

- Specify the hyperparameters and the range of their values to be explored
- Determine how the search will be done, for example grid search
- Determine how many times (evaluations) the procedure will be repeated

The above can be done (in that order) with the following code. In the end we combine everything into a new pipeline `at_ridge` which is trained on the training data.

```
library('mlr3tuning')
library('paradox')
# Set up tuning environment
tune_lambda <- ParamSet$new (list(
 ParamDbl$new('regr.glmnet.lambda', lower = 0.001, upper = 2)
))
tuner<-tnr('grid_search')
terminator <- trm('evals', n_evals = 20)
#Put everything together in a new learner
at_ridge <- AutoTuner$new(
  learner = glrn_ridge2,
  resampling = rsmp('cv', folds = 3),
  measure = measure,
  search_space = tune_lambda,
  terminator = terminator,
  tuner = tuner
)
#Train the learner on the training data
at_ridge$train(task, row_ids = train_set)
```

After training the ridge machine learning pipeline, the next step is to evaluate its performance. Below we just report it `mse`. Further inspection of the training, for example finding out the optimal value of `lambda` can be done with `at_ridge$model`.

```
at_ridge$predict(task, row_ids = test_set)$score()
regr.mse
27.18366
```

The result is a slightly lower `mse` which is not convincing. A similar learner is offered by *lasso regression* and can be fit using the previous code and simply setting `alpha = 1`; it is recommended to do it as an exercise.

# Random Forests

So, next, we move away from a linear regression setting and try *random forests*, a very successful technique in Kaggle competitions. Random forests are known to do very well in finding non-linear associations as well as synergies between the features. For (entirely optional) reading on random forests, the James et al. (2014) textbook is a good place to start. To specify a random forest pipeline the code is almost identical and only differs in the learner (`regr.ranger`) and its parameters. Again, the code below assumes that the R package **ranger** is installed in your computer.

```
learner_rf <- lrn('regr.ranger')
learner_rf$param_set$values <- list(min.node.size = 4)
gr_rf <- po('scale') %>>%
  po('imputemean') %>>%
  po(learner_rf)
glrn_rf <- GraphLearner$new(gr_rf)
tune_ntrees <- ParamSet$new (list(
 ParamInt$new('regr.ranger.num.trees', lower = 50, upper = 600)
))
at_rf <- AutoTuner$new(
  learner = glrn_rf,
  resampling = rsmp('cv', folds = 3),
  measure = measure,
  search_space = tune_ntrees,
  terminator = terminator,
  tuner = tuner
)
at_rf$train(task, row_ids = train_set)
```

To get the prediction we use the same code as before:

```
at_rf$predict(task, row_ids = test_set)$score()
regr.mse
15.12726
```

We see that the improvement offered by random forests is substantial in this case.

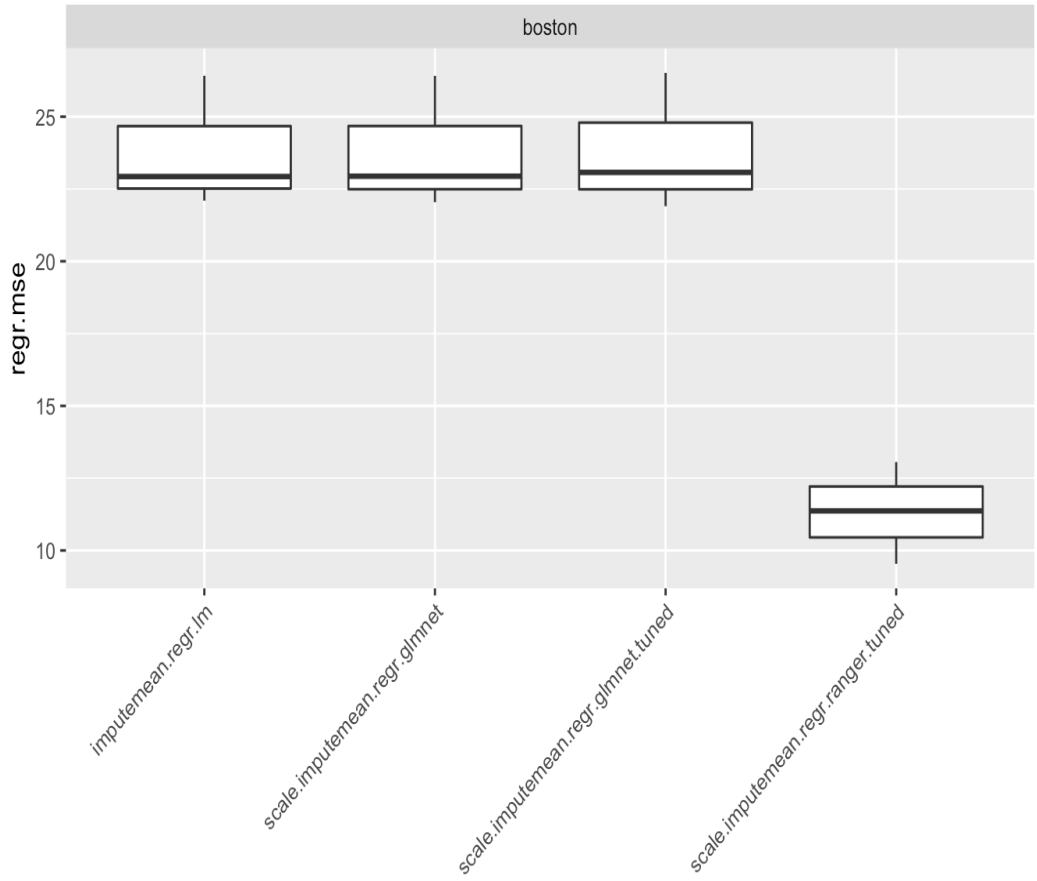
# Benchmarking

An alternative and easier-to-code way to compare the different learners is by the `benchmark()` function which automates the train/predict steps. It requires to setup a benchmark design as an input which can be set by the function `benchmark_grid()`. The latter function tabulates the different comparisons to be made that consist of combinations of tasks, different ways to resample and different learners. Below, we stick to regression task of the Boston data, choose three-fold cross-validation and the four learners we checked so far (the code below may take a while to run):

```
set.seed(123) # for reproducible results

# list of learners

lrn_list <- list(

  glrn_lm,

  glrn_ridge,

  at_ridge,

  at_rf

)

# set the benchmark design and run the comparisons

bm_design <- benchmark_grid(task = task, resamplings = rsmp('cv', folds = 3), learn
ers = lrn_list)

bmr <- benchmark(bm_design, store_models = TRUE)
```

A nice way to visualise the comparisons made is with boxplots. We can also see the overall `mse` for each learner using `bmr$aggregate(measure)`.

```
library('mlr3viz')

library('ggplot2')

autoplot(bmr) + theme(axis.text.x = element_text(angle = 45, hjust = 1))
```




```
bmr$aggregate(measure)
   nr      resample_result task_id                          learner_id
1:  1 <ResampleResult[21]>  boston                imputemean.regr.lm
2:  2 <ResampleResult[21]>  boston       scale.imputemean.regr.glmnet
3:  3 <ResampleResult[21]>  boston scale.imputemean.regr.glmnet.tuned
4:  4 <ResampleResult[21]>  boston scale.imputemean.regr.ranger.tuned
   resampling_id iters regr.mse
1:            cv     3 23.81437
2:            cv     3 23.79866
3:            cv     3 23.83034
4:            cv     3 11.32159
```

The results above confirm the clear dominance or random forests over regression methods in the Boston dataset.

## Parallelisation

Training pipelines may end up taking a lot of time. One way to reduce this is by splitting the procedure into multiple jobs and execute them in parallel, simultaneously. This procedure is known as *parallelisation* and allows for significant gains in terms of computing power and substantial reduction in the execution times.

The package `mlr3` is compatible with the **`future`** package for parallelisation, thus making it very easy to parallelise. All you have to do is essentially install the package **`future`** and simply run the following line in the beginning of your code

```
future::plan()
```

# Useful Links and Resources

- Book for the `mlr3` package
- Function Reference for the `mlr3` package
- Project `mlr` with several resources
- The `caret` package; an alternative to `mlr3`

# References

Harrison, D., & Rubinfeld, D. L. (1978). Hedonic prices and the demand for clean air. *Journal Environmental Economics and Management*, *5*, 81–102.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2014). *An introduction to statistical learning: With applications in R*. Springer.

# Categorical Variables in Machine Learning

As already mentioned categorical variables take values in a predetermined finite set of categories. An example of a categorical variable is whether an email is spam or not. Below we will briefly discuss how to handle such variables in machine learning (mostly supervised learning) depending on whether they are forming the target variable or feature variables.

## Categorical Predictions

In order to illustrate the concepts let us consider binary classification with the target being \(Y\) taking values of 1: a person will default on its credit card or 0: no default. Typically machine learning models, also called *classifiers* in this case, provide probabilities of \(Y=1\) for each individual given their set of features. Often a threshold is selected for these probabilities, above which we classify the case as 1 otherwise it is classified as 0. Without loss of generality, let us assume that this threshold is 0.5. In our example, the individuals with probability higher than 0.5 are classified as projected to default, otherwise they are considered safe. In the presence of the data we can then check the actual outcome, i.e. whether they did default or not. When contrasting projected classifications and outcomes, there are four different cases:

- *True positives*: Projected to default and defaulted
- *False positives*: Projected to default but did not default
- *True negatives*: Did not project to default and did not default
- *False negatives*: Did not project to default but defaulted

The rate of *true positives*, correctly identified is also known as *sensitivity* whereas the rate of *true negatives* correctly identified is also known as *specificity*. These quantities can be summarised in the so called *confusion matrix*. Below we see an example of a confusion matrix, taken from James et al. (2014), that corresponds to the threshold of 0.5:

| | | *True default status* | | |
|---|---|---|---|---|
| | | No | Yes | Total |
| *Predicted default status* | No | 9,644 | 252 | 9,896 |
| | Yes | 23 | 81 | 104 |
| | Total | 9,667 | 333 | 10,000 |

> Confusion matrix of the previous example based on a probability threshold of 0.5; Table 4.4 in James et al. (2014).

Based on the above matrix we see that overall classification accuracy is `(9644+81)/10000`, or else 97.25%. This may seem high but if we focus on the 333 individuals who defaulted only 81 were identified. In other words the sensitivity is `81/333=0.24`, which suggests that 76% were missed. On the other hand the model is very successful in classifying the negatives, i.e. those less likely to default, as the specificity is `9644/9667=0.99`. Note however that the confusion matrix depends on the choice of threshold whose choice may also be determined by the context. In this case the aim may be to identify individuals that are likely to default and take up an action to prevent it. In that sense we may be willing to tolerate some false negative, in other words

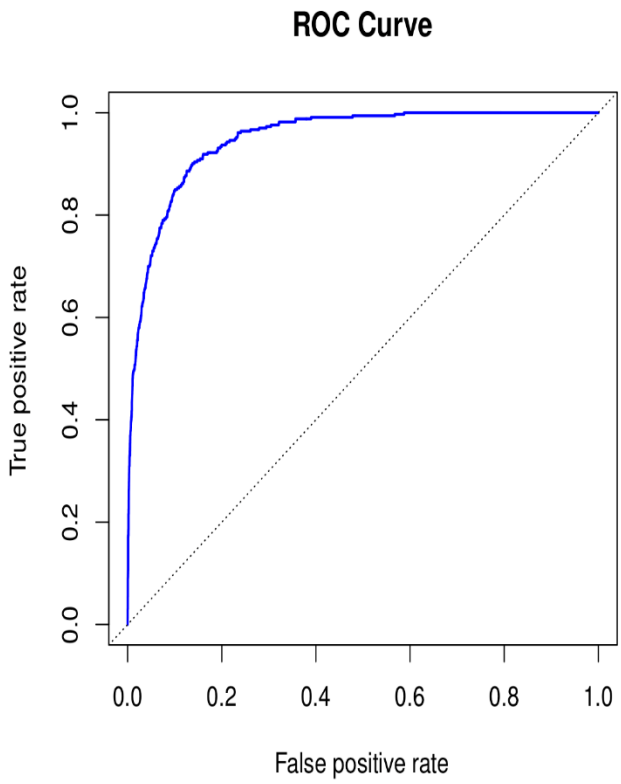
triggering false alarms indicating people are in danger when they are not, if this results in higher chances of identifying people in danger. This can be calibrated by choosing another threshold. Below is the confusion matrix from the same example for a threshold of 0.2:

| | | *True default status* | | |
|---|---|---|---|---|
| | | No | Yes | Total |
| *Predicted default status* | No | 9,432 | 138 | 9,570 |
| | Yes | 235 | 195 | 430 |
| | Total | 9,667 | 333 | 10,000 |

> Confusion matrix of the previous example based on a probability threshold of 0.2; Table 4.5 in James et al. (2014).

Now we get a sensitivity of `195/333=0.59` and a specificity of `9432/9667=0.97`, which is better for the purposes of this classification task.

In other cases however setting a fixed threshold is not so straightforward and we are interested in getting a measure that does not depend in its choice. This is provided by the *receiver operating characteristics (ROC) curve* which is constructed by taking a range of thresholds and plotting the sensitivity (true positive rate) versus one minus the specificity (false positive rate). It turns out that the *area under the ROC curve*, known as *AUC*, corresponds to the probability of correctly classifying a positive and a negative case. In our example the ROC curve is shown below and corresponds to an AUC of 0.95.




> ROC curve for the previous example with AUC of 0.95. Figure 4.8 in James et al. (2014)

In the ROC plot above the ideal ROC curve would come as close as possible to the top left corner, which would correspond to high true positive rate (high sensitivity) and low false positive rate (low specificity). The dotted line corresponds to a classifier that is essential guessing at random and classifies correctly with 50% chance.

# Categorical Features

There are two ways to handle categorical variables as features (inputs) in regression or classification tasks. The first way is to assign a numerical score on their values and treat them as continuous. This is clearly suboptimal from an interpretation viewpoint as the assignment of numerical scores is arbitrary. In the case of nominal variables (e.g. 1:red, 2:yellow, 3:green) there is no reason why one of these colours should have bigger values than the others. But even in ordinal variables (e.g. 1:low, 2:medium, 3:high) there is no guarantee why the distance between medium and low is the same as that of between high and medium. So we may as well have used the numbers (1, 3, 7) or any other ordered triplet of real numbers. Despite this, the use of numerical scores (quantification) is still being used as it is convenient and result in fewer features in the model.

A more meaningful option is to create *dummy variable(s)* from each categorical variable. Dummy variables take values of either 0 or 1, indicating the presence (1) or absence (0) of a category. Typically the number of dummy variables for each categorical variable is equal to the number of categories (say k) minus 1; one category is treated as the reference category. Let us consider the (1:red, 2:yellow, 3:green) example, where we can set the green category as reference and create two dummy variables: one taking value 1 if the colour is red and 0 otherwise, and another one taking the value 1 if the colour is yellow and 0 otherwise. The reason for creating k-1 dummy variables, rather than k, is that the k-th dummy variable is redundant in the presence of the others. Still on the (1:red, 2:yellow, 3:green) example, if we know whether the colour is red (or not) and whether it is yellow (or not), then we can also infer if it is green or not. In other words, if the dummy variable for red is 1 or if the dummy variable for yellow is 1, the colour is not green. But if both of the dummy variables for red and yellow are 0, then the colour is red.

Now consider a regression model where the target is $y$ and the dummy variables for red and yellow are the features. The coefficient of the red dummy variable will then reflect the mean difference (in the values of $y$) between cases corresponding to red and green colours. Similarly the coefficient of the yellow dummy variable will reflect the mean difference (in the values of $y$) between cases corresponding to yellow and green colours. The constant in the regression equation will reflect the mean value of $y$ when both dummy variables are zero or, in other words, when the colour is green.

Hence, we can setup a regression model with a categorical feature in that way and the regression coefficients will reflect the mean values of $y$ in each of the categories. All we need to do is construct these dummy variables, use them as inputs, and then proceed as in the case of continuous inputs. Note also that the choice of the reference category only affects the interpretation of the regression coefficients. In terms of predictive performance, models with dummy variables extracted on the basis of different reference categories will be equivalent.

# Useful Links and Resources

- ISLR book for a gentle introduction to machine learning

# References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2014). *An introduction to statistical learning: With applications in R*. Springer.

# ST2195 Programming for Data Science
# Machine Learning Frameworks in Python

** Note: The code chunks below should be run in the following order **

## Python Setup

We start by activating the necessary Python packages. In addition to the standard use of `pandas` and `matplotlib`, we will be using several features from the `scikit-learn` or `sklearn` package which is the default option for machine learning in Python. The `scikit-learn` package is an open-source collection of simple and efficient tools for machine learning, built on `numpy`, `scipy`, and `matplotlib`. More details about it can be found here.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml  #using openml to import data
from sklearn.metrics import plot_roc_curve
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer #transform different types
```

# Data

We will be working on the *titanic* dataset that is available from the OpenML website. The features and response are stored in separate arrays, `X_initial`, and `y` respectively. To view all the data we combine them into a single pandas data frame called `combined_dataset`.

```
# Dataset details at-

X_initial, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

combine_dataset = pd.concat([X_initial, y], axis=1)

combine_dataset.head()
```

| | pclass | name | sex | age | sibsp | parch | ticket | fare | cabin | embarked | boat | body | home.dest | survived |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.0 | Allen, Miss. Elisabeth Walton | female | 29.0000 | 0.0 | 0.0 | 24160 | 211.3375 | B5 | S | 2 | NaN | St Louis, MO | 1 |
| 1 | 1.0 | Allison, Master. Hudson Trevor | male | 0.9167 | 1.0 | 2.0 | 113781 | 151.5500 | C22 C26 | S | 11 | NaN | Montreal, PQ / Chesterville, ON | 1 |
| 2 | 1.0 | Allison, Miss. Helen Loraine | female | 2.0000 | 1.0 | 2.0 | 113781 | 151.5500 | C22 C26 | S | None | NaN | Montreal, PQ / Chesterville, ON | 0 |
| 3 | 1.0 | Allison, Mr. Hudson Joshua Creighton | male | 30.0000 | 1.0 | 2.0 | 113781 | 151.5500 | C22 C26 | S | None | 135.0 | Montreal, PQ / Chesterville, ON | 0 |
| 4 | 1.0 | Allison, Mrs. Hudson J C (Bessie Waldo Daniels) | female | 25.0000 | 1.0 | 2.0 | 113781 | 151.5500 | C22 C26 | S | None | NaN | Montreal, PQ / Chesterville, ON | 0 |

Titanic pandas data frame

To illustrate concepts we will focus on five features: age, fare, embarked, sex and class. The matrix of these five features to be used in the following examples will be labelled as `X`.

```
# features from the dataset

features = ['age', 'fare','embarked','sex','pclass']

X = X_initial[features].copy()
```

# Pipelines: Pre-Processing Stage

We will now start to build the pipelines. To do so we will use different pre-processing steps for continuous (numerical) and categorical features. For the numerical features (age and fare) we apply imputation on the missing values with the function `SimpleImputer()` (imputing either the mean or the median of the observations of each variable) and scaling with the function `StandardScaler()`. These operations are bundled together in a pipeline called `numerical_transformer`. The function `Pipeline()` is being used where we simply define the different steps. Each step is specified within brackets, where the name of the step is specified (e.g. `name of step`) followed by the Python function used:

```
numerical_features = ['age', 'fare']


# Applying SimpleImputer and StandardScaler into a pipelin

numerical_transformer = Pipeline(steps=[

    ('imputer', SimpleImputer()),
```

```
    ('scaler', StandardScaler())])
```

For categorical features, imputation is applied as before and we also use the `OneHotEcoder()` function to create the appropriate dummy variables (see notes on categorical variables in this week's material). The resulting pipeline is called `categorical_transformer`.

```
categorical_features = ['embarked', 'sex', 'pclass']


# Applying SimpleImputer and then OneHotEncoder into another pipeline
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer()),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

Having pre-processed the numerical and categorical features separately we can now merge them with the function `ColumnTransformer()`, in which we specify the two pipelines to be merged using the argument `transformers`. This creates a new pipeline called `data_transformer`.

```
data_transformer = ColumnTransformer(
    transformers=[
        ('numerical', numerical_transformer, numerical_features),
        ('categorical', categorical_transformer, categorical_features)])
```

The `data_transformer` pipeline now contains all the pre-processing steps. It can be used as the starting point for different (machine) learners as we illustrate in the next sections.

# Pipelines With Logistic Regression

We are now ready to complete our first machine learning pipeline by simply attaching a learner to the `data_transformer` pipeline from the previous section. In this section we use logistic regression since the response (`y`: survived) is binary.

```
pipe_lr = Pipeline(steps=[('data_transformer', data_transformer),
                      ('pipe_lr', LogisticRegression(max_iter=10000))])
```

The next stage is to perform the standard machine learning operations of training the pipeline and evaluating its predictive performance. We begin by splitting the data into train and test datasets with the very convenient function `train_test_split()`.

```
X_train, X_test, y_train, y_test = train_test_split(X_initial, y, test_size=0.5,ran
dom_state=1)
```

To tune the pipeline we can use cross-validation for each combination of its hyperparameters. In other words we can use *grid search cross-validation* that can be done with the function `GridSearchCV()`. This requires setting up a parameter grid, which is a dictionary containing the names of each hyperparameter followed by its range of values. The hyperparameters to be tuned below are the imputation strategies used in `SimpleImputer()`. In the case of numerical variables there are two options, mean or median. For categorical variables one option is to impute the most frequent value of each variable (`most_frequent`), whereas another option is to impute the

missing value to be the reference category (`constant` with filled value being left to the default of 0). The best possible combination is chosen based on fitting the models on the training data.

```
param_grid = {
    'data_transformer__numerical__imputer__strategy': ['mean', 'median'],
    'data_transformer__categorical__imputer__strategy': ['constant','most_frequent']
}
grid_lr = GridSearchCV(pipe_lr, param_grid=param_grid)
grid_lr.fit(X_train, y_train);
```

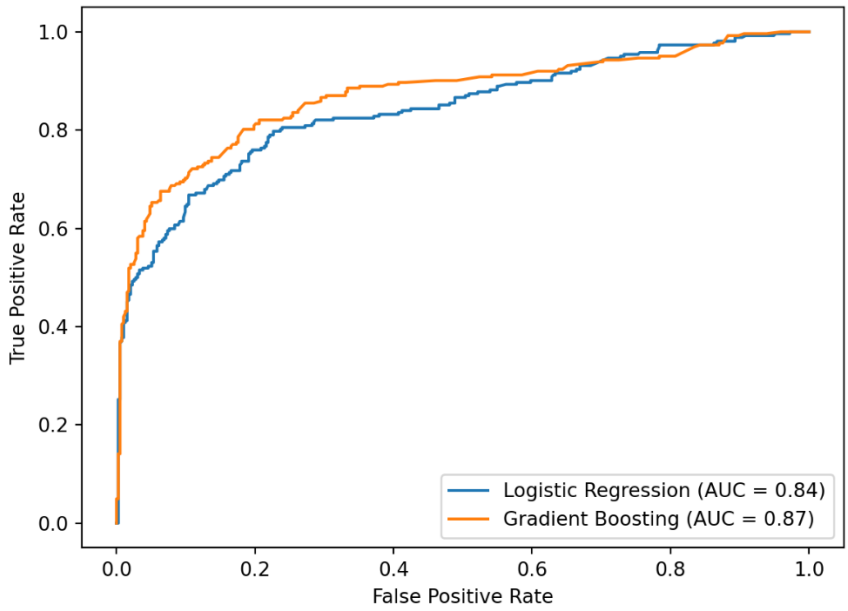
# Pipelines With Gradient Boosting

Before we evaluate the logistic regression pipeline, let us consider also the Gradient Boosting learner. For simplicity we do not explore tuning the hyperparameters of this learner and simply use the default values for them. Hence, the code is very similar as before, we do add the argument `random_state=2` to ensure reproducible results.

```
pipe_gdb = Pipeline(steps=[('data_transformer', data_transformer),
       ('pipe_gdb',GradientBoostingClassifier(random_state=2))])

grid_gdb = GridSearchCV(pipe_gdb, param_grid=param_grid)
grid_gdb.fit(X_train, y_train);
```

Finally let us compare the two machine learning pipelines by overlaying their receiver operating characteristic (ROC) curves.

```
ax = plt.gca()
plot_roc_curve(grid_lr, X_test, y_test, ax=ax, name='Logistic Regression')
<sklearn.metrics._plot.roc_curve.RocCurveDisplay object at 0x14a0606d8>
plot_roc_curve(grid_gdb, X_test, y_test, ax=ax, name='Gradient Boosting')
<sklearn.metrics._plot.roc_curve.RocCurveDisplay object at 0x14a064198>
plt.show()
```




The Gradient Boosting offers a small improvement which can potentially be improved by tuning its hyperparameters on the training dataset. This is left as an exercise.

# Parallelisation

As in R, it is straightforward to use parallel computing in order to speed up the training time. The function `GridSearchCV()` has the argument `njobs`, which if set to any value other than 1 results in parallel computing. Setting `njobs=-1` results in using all the available cores. An alternative to `GridSearchCV()`, which can also be parallelised in the same way, is `RandomizedSearchCV()` that searches randomly over the possible combinations of hyperparameter values.

# Useful Links and Resources

- OpenML: Website with several datasets and useful resources for machine learning
- The `scikit learn` project
- More details on the Pipeline function of the sklearn package