Feature Engineering

Alfonso R. Reyes

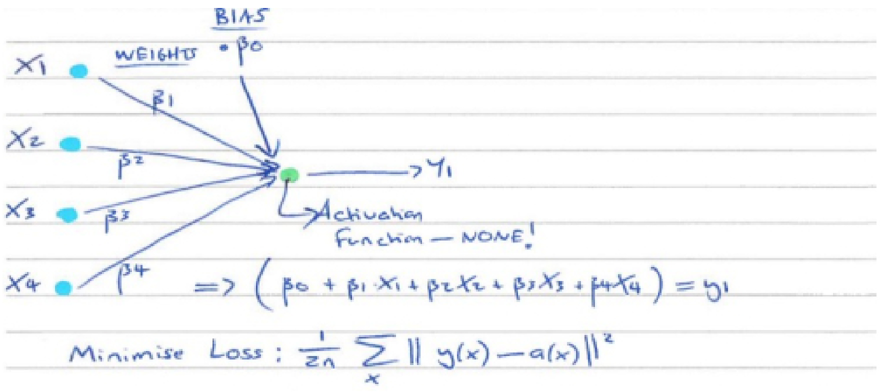
2019-09-18

Contents

Prerequisites

```
print(assets_dir)
#> [1] "/home/datascience/repos/machine-learning-rsuite/import/assets"
image_file <- file.path(assets_dir, "linear_regression.jpg")
file.exists(image_file)
#> [1] TRUE
```

```
knitr::include_graphics(image_file)
```


BIAS
β_0
WEIGHTS
X_1
X_2
X_3
X_4
β_1
β_2
β_3
β_4
Y_1
Activation Function — NONE!
$\Rightarrow (\beta_0 + \beta_1 \cdot X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4) = y_1$
Minimise Loss : $\frac{1}{2n} \sum_x \| y(x) - a(x) \|^2$

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

Employee attrition. Employee-Attrition dataset. *LIME* package

Article: https://www.business-science.io/business/2017/09/18/hr_employee_attrition.html Data: https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/

1.1 Introduction

1.1.1 Employee attrition: a major problem

Bill Gates was once quoted as saying,

> "You take away our top 20 employees and we [Microsoft] become a mediocre company".

His statement cuts to the core of a major problem: employee attrition. An organization is only as good as its employees, and these people are the true source of its competitive advantage.

Organizations face huge costs resulting from employee turnover. Some costs are tangible such as training expenses and the time it takes from when an employee starts to when they become a productive member. However, the most important costs are intangible. Consider what's lost when a productive employee quits: new product ideas, great project management, or customer relationships.

With advances in machine learning and data science, its possible to not only predict employee attrition but to understand the key variables that influence turnover. We'll take a look at two cutting edge techniques:

1. Machine Learning with `h2o.automl()` from the h2o package: This function takes automated machine learning to the next level by testing a number of advanced algorithms such as random forests, ensemble methods, and deep learning along with more traditional algorithms such as logistic regression. The main takeaway is that we can now easily achieve predictive performance that is in the same ball park (and in some cases even better than) commercial algorithms and ML/AI software.
2. Feature Importance with the `lime` package: The problem with advanced machine learning algorithms such as deep learning is that it's near impossible to understand the algorithm because of its complexity. This has all changed with the lime package. The major advancement with lime is that, by recursively analyzing the models locally, it can extract feature importance that repeats globally. What this means to us is that lime has opened the door to understanding the ML models regardless of complexity. Now

the best (and typically very complex) models can also be investigated and potentially understood as to what variables or features make the model tick.

1.1.2 Employee attrition: machine learning analysis

With these new automated ML tools combined with tools to uncover critical variables, we now have capabilities for both extreme predictive accuracy and understandability, which was previously impossible! We'll investigate an HR Analytic example of employee attrition that was evaluated by IBM Watson.

1.1.3 Where we got the data

The example comes from IBM Watson Analytics website. You can download the data and read the analysis here:

Get data used in this post here. Read IBM Watson Analytics article here. To summarize, the article makes a usage case for IBM Watson as an automated ML platform. The article shows that using Watson, the analyst was able to detect features that led to increased probability of attrition.

1.1.4 Automated machine learning (what we did with the data)

In this example we'll show how we can use the combination of H2O for developing a complex model with high predictive accuracy on unseen data and then how we can use LIME to understand important features related to employee attrition.

1.1.5 Load packages

Load the following packages.

```r
# Load the following packages
library(tidyquant)  # Loads tidyverse and several other pkgs
library(readxl)     # Super simple excel reader
library(h2o)        # Professional grade ML pkg
library(lime)       # Explain complex black-box ML models
```

1.1.6 Load data

Download the data here. You can load the data using read_excel(), pointing the path to your local file.

```r
# Read excel data
hr_data_raw <- read_excel(path = file.path(data_raw_dir,
                                           "WA_Fn-UseC_-HR-Employee-Attrition.xlsx"))
```

Let's check out the raw data. It's 1470 rows (observations) by 35 columns (features). The "Attrition" column is our target. We'll use all other columns as features to our model.

```r
# View first 10 rows
hr_data_raw[1:10,] %>%
    knitr::kable(caption = "First 10 rows")
```

The only pre-processing we'll do in this example is change all character data types to factors. This is needed for H2O. We could make a number of other numeric data that is actually categorical factors, but this tends to increase modeling time and can have little improvement on model performance.

Table 1.1: First 10 rows

Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	Educat
41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sc
49	No	Travel_Frequently	279	Research & Development	8	1	Life Sc
37	Yes	Travel_Rarely	1373	Research & Development	2	2	Other
33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sc
27	No	Travel_Rarely	591	Research & Development	2	1	Medica
32	No	Travel_Frequently	1005	Research & Development	2	2	Life Sc
59	No	Travel_Rarely	1324	Research & Development	3	3	Medica
30	No	Travel_Rarely	1358	Research & Development	24	1	Life Sc
38	No	Travel_Frequently	216	Research & Development	23	3	Life Sc
36	No	Travel_Rarely	1299	Research & Development	27	3	Medica

```r
hr_data <- hr_data_raw %>%
    mutate_if(is.character, as.factor) %>%
    select(Attrition, everything())
```

Let's take a glimpse at the processed dataset. We can see all of the columns. Note our target ("Attrition") is the first column.

```r
glimpse(hr_data)
#> Observations: 1,470
#> Variables: 35
#> $ Attrition                <fct> Yes, No, Yes, No, No, No, No, No, No,...
#> $ Age                      <dbl> 41, 49, 37, 33, 27, 32, 59, 30, 38, 3...
#> $ BusinessTravel           <fct> Travel_Rarely, Travel_Frequently, Tra...
#> $ DailyRate                <dbl> 1102, 279, 1373, 1392, 591, 1005, 132...
#> $ Department               <fct> Sales, Research & Development, Resear...
#> $ DistanceFromHome         <dbl> 1, 8, 2, 3, 2, 2, 3, 24, 23, 27, 16, ...
#> $ Education                <dbl> 2, 1, 2, 4, 1, 2, 3, 1, 3, 3, 3, 2, 1...
#> $ EducationField           <fct> Life Sciences, Life Sciences, Other, ...
#> $ EmployeeCount            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
#> $ EmployeeNumber           <dbl> 1, 2, 4, 5, 7, 8, 10, 11, 12, 13, 14,...
#> $ EnvironmentSatisfaction  <dbl> 2, 3, 4, 4, 1, 4, 3, 4, 4, 3, 1, 4, 1...
#> $ Gender                   <fct> Female, Male, Male, Female, Male, Mal...
#> $ HourlyRate               <dbl> 94, 61, 92, 56, 40, 79, 81, 67, 44, 9...
#> $ JobInvolvement           <dbl> 3, 2, 2, 3, 3, 3, 4, 3, 2, 3, 4, 2, 3...
#> $ JobLevel                 <dbl> 2, 2, 1, 1, 1, 1, 1, 1, 3, 2, 1, 2, 1...
#> $ JobRole                  <fct> Sales Executive, Research Scientist, ...
#> $ JobSatisfaction          <dbl> 4, 2, 3, 3, 2, 4, 1, 3, 3, 3, 2, 3, 3...
#> $ MaritalStatus            <fct> Single, Married, Single, Married, Mar...
#> $ MonthlyIncome            <dbl> 5993, 5130, 2090, 2909, 3468, 3068, 2...
#> $ MonthlyRate              <dbl> 19479, 24907, 2396, 23159, 16632, 118...
#> $ NumCompaniesWorked       <dbl> 8, 1, 6, 1, 9, 0, 4, 1, 0, 6, 0, 0, 1...
#> $ Over18                   <fct> Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y...
#> $ OverTime                 <fct> Yes, No, Yes, Yes, No, No, Yes, No, N...
#> $ PercentSalaryHike        <dbl> 11, 23, 15, 11, 12, 13, 20, 22, 21, 1...
#> $ PerformanceRating        <dbl> 3, 4, 3, 3, 3, 3, 4, 4, 4, 3, 3, 3, 3...
#> $ RelationshipSatisfaction <dbl> 1, 4, 2, 3, 4, 3, 1, 2, 2, 2, 3, 4, 4...
#> $ StandardHours            <dbl> 80, 80, 80, 80, 80, 80, 80, 80, 80, 8...
#> $ StockOptionLevel         <dbl> 0, 1, 0, 0, 1, 0, 3, 1, 0, 2, 1, 0, 1...
#> $ TotalWorkingYears        <dbl> 8, 10, 7, 8, 6, 8, 12, 1, 10, 17, 6, ...
#> $ TrainingTimesLastYear    <dbl> 0, 3, 3, 3, 3, 2, 3, 2, 2, 3, 5, 3, 1...
#> $ WorkLifeBalance          <dbl> 1, 3, 3, 3, 3, 2, 2, 3, 3, 2, 3, 3, 2...
#> $ YearsAtCompany           <dbl> 6, 10, 0, 8, 2, 7, 1, 1, 9, 7, 5, 9, ...
#> $ YearsInCurrentRole       <dbl> 4, 7, 0, 7, 2, 7, 0, 0, 7, 7, 4, 5, 2...
#> $ YearsSinceLastPromotion  <dbl> 0, 1, 0, 3, 2, 3, 0, 0, 1, 7, 0, 0, 4...
#> $ YearsWithCurrManager     <dbl> 5, 7, 0, 0, 2, 6, 0, 0, 8, 7, 3, 8, 3...
```

1.2 Modeling Employee attrition

We are going to use the `h2o.automl()` function from the H2O platform to model employee attrition.

1.2.1 Machine Learning with `h2o`

First, we need to initialize the *Java Virtual Machine (JVM)* that H2O uses locally.

```
# Initialize H2O JVM
h2o.init()
#>
#> H2O is not running yet, starting it now...
#>
#> Note:  In case of errors look at the following log files:
#>     /tmp/RtmpCSPWrv/h2o_datascience_started_from_r.out
#>     /tmp/RtmpCSPWrv/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>     H2O cluster uptime:         1 seconds 256 milliseconds
#>     H2O cluster timezone:       America/Chicago
#>     H2O data parsing timezone:  UTC
#>     H2O cluster version:        3.22.1.1
#>     H2O cluster version age:    8 months and 20 days !!!
#>     H2O cluster name:           H2O_started_from_R_datascience_mwl453
#>     H2O cluster total nodes:    1
#>     H2O cluster total memory:   6.96 GB
#>     H2O cluster total cores:    8
#>     H2O cluster allowed cores:  8
#>     H2O cluster healthy:        TRUE
#>     H2O Connection ip:          localhost
#>     H2O Connection port:        54321
#>     H2O Connection proxy:       NA
#>     H2O Internal Security:      FALSE
#>     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
#>     R Version:                  R version 3.6.0 (2019-04-26)
h2o.no_progress() # Turn off output of progress bars
```

Next, we change our data to an h2o object that the package can interpret. We also split the data into training, validation, and test sets. Our preference is to use 70%, 15%, 15%, respectively.

```
# Split data into Train/Validation/Test Sets
hr_data_h2o <- as.h2o(hr_data)
```

```r
split_h2o <- h2o.splitFrame(hr_data_h2o, c(0.7, 0.15), seed = 1234 )
train_h2o <- h2o.assign(split_h2o[[1]], "train" ) # 70%
valid_h2o <- h2o.assign(split_h2o[[2]], "valid" ) # 15%
test_h2o  <- h2o.assign(split_h2o[[3]], "test" )  # 15%
```

1.3 Model

Now we are ready to model. We'll set the target and feature names. The target is what we aim to predict (in our case "Attrition"). The features (every other column) are what we will use to model the prediction.

```r
# Set names for h2o
y <- "Attrition"
x <- setdiff(names(train_h2o), y)
```

Now the fun begins. We run the h2o.automl() setting the arguments it needs to run models against. For more information, see the h2o.automl documentation.

- `x = x`: The names of our feature columns.
- `y = y`: The name of our target column.
- `training_frame` = train_h2o: Our training set consisting of 70% of the data.
- `leaderboard_frame` = valid_h2o: Our validation set consisting of 15% of the data. H2O uses this to ensure the model does not overfit the data.
- `max_runtime_secs = 30`: We supply this to speed up H2O's modeling. The algorithm has a large number of complex models so we want to keep things moving at the expense of some accuracy.

```r
# Run the automated machine learning
automl_models_h2o <- h2o.automl(
    x = x,
    y = y,
    training_frame    = train_h2o,
    leaderboard_frame = valid_h2o,
    max_runtime_secs  = 30
    )
```

All of the models are stored the `automl_models_h2o` object. However, we are only concerned with the leader, which is the best model in terms of accuracy on the validation set. We'll extract it from the models object.

```r
# Extract leader model
automl_leader <- automl_models_h2o@leader
```

1.4 Predict

Now we are ready to predict on our test set, which is unseen from during our modeling process. This is the true test of performance. We use the `h2o.predict()` function to make predictions.

```r
# Predict on hold-out set, test_h2o
pred_h2o <- h2o.predict(object = automl_leader, newdata = test_h2o)
```

1.5 Performance

Now we can evaluate our leader model. We'll reformat the test set an add the predictions as column so we have the actual and prediction columns side-by-side.

```
# Prep for performance assessment
test_performance <- test_h2o %>%
    tibble::as_tibble() %>%
    select(Attrition) %>%
    add_column(pred = as.vector(pred_h2o$predict)) %>%
    mutate_if(is.character, as.factor)
test_performance
#> # A tibble: 211 x 2
#>   Attrition pred
#>   <fct>     <fct>
#> 1 No        No
#> 2 No        No
#> 3 Yes       Yes
#> 4 No        No
#> 5 No        No
#> 6 No        No
#> # ... with 205 more rows
```

We can use the table() function to quickly get a confusion table of the results. We see that the leader model wasn't perfect, but it did a decent job identifying employees that are likely to quit. For perspective, a logistic regression would not perform nearly this well.

```
# Confusion table counts
confusion_matrix <- test_performance %>%
    table()
confusion_matrix
#>          pred
#> Attrition  No Yes
#>       No  165  17
#>       Yes   9  20
```

We'll run through a binary classification analysis to understand the model performance.

```
# Performance analysis
tn <- confusion_matrix[1]
tp <- confusion_matrix[4]
fp <- confusion_matrix[3]
fn <- confusion_matrix[2]

accuracy <- (tp + tn) / (tp + tn + fp + fn)
misclassification_rate <- 1 - accuracy
recall <- tp / (tp + fn)
precision <- tp / (tp + fp)
null_error_rate <- tn / (tp + tn + fp + fn)

tibble(
    accuracy,
    misclassification_rate,
    recall,
    precision,
    null_error_rate
) %>%
    transpose()
#> [[1]]
#> [[1]]$accuracy
#> [1] 0.877
#>
#> [[1]]$misclassification_rate
#> [1] 0.123
#>
#> [[1]]$recall
#> [1] 0.69
#>
#> [[1]]$precision
#> [1] 0.541
#>
#> [[1]]$null_error_rate
#> [1] 0.782
```

It is important to understand is that the accuracy can be misleading: 88% sounds pretty good especially for modeling HR data, but if we just pick `Attrition = NO` we would get an accuracy of about 79%. Doesn't sound so great now.

Before we make our final judgement, let's dive a little deeper into precision and recall. Precision is when the model predicts yes, how often is it actually yes. Recall (also true positive rate or specificity) is when the actual value is yes how often is the model correct. Confused yet? Let's explain in terms of what's important to HR.

Most HR groups would probably prefer to incorrectly classify folks not looking to quit as high potential of quiting rather than classify those that are likely to quit as not at risk. Because it's important to not miss at risk employees, HR will really care about recall or when the actual value is `Attrition = YES` how often the model predicts YES.

Recall for our model is 62%. In an HR context, this is 62% more employees that could potentially be targeted prior to quiting. From that standpoint, an organization that loses 100 people per year could possibly target 62 implementing measures to retain.

1.6 The `lime` package

We have a very good model that is capable of making very accurate predictions on unseen data, but what can it tell us about what causes attrition? Let's find out using LIME.

1.6.1 Set up

The `lime` package implements LIME in R. One thing to note is that it's not setup out-of-the-box to work with `h2o`. The good news is with a few functions we can get everything working properly. We'll need to make two custom functions:

- `model_type`: Used to tell lime what type of model we are dealing with. It could be classification, regression, survival, etc.
- `predict_model`: Used to allow lime to perform predictions that its algorithm can interpret.

The first thing we need to do is identify the class of our model leader object. We do this with the `class()` function.

```
class(automl_leader)
#> [1] "H2OBinomialModel"
#> attr(,"package")
#> [1] "h2o"
```

Next we create our `model_type` function. It's only input is x the h2o model. The function simply returns "classification", which tells LIME we are classifying.

```
# Setup lime::model_type() function for h2o
model_type.H2OBinomialModel <- function(x, ...) {
    # Function tells lime() what model type we are dealing with
    # 'classification', 'regression', 'survival', 'clustering', 'multilabel', etc
    #
    # x is our h2o model

    return("classification")
}
```

Now we can create our `predict_model` function. The trick here is to realize that it's inputs must be x a model, newdata a dataframe object (this is important), and type which is not used but can be use to switch the output type. The output is also a little tricky because it must be in the format of probabilities by classification (this is important; shown next). Internally we just call the `h2o.predict()` function.

```
# Setup lime::predict_model() function for h2o
predict_model.H2OBinomialModel <- function(x, newdata, type, ...) {
    # Function performs prediction and returns dataframe with Response
    #
    # x is h2o model
    # newdata is data frame
    # type is only setup for data frame

    pred <- h2o.predict(x, as.h2o(newdata))

    # return probs
    return(as.data.frame(pred[,-1]))

}
```

Run this next script to show you what the output looks like and to test our `predict_model` function. See how it's the probabilities by classification. It must be in this form for `model_type = "classification"`.

```
# Test our predict_model() function
predict_model(x = automl_leader, newdata = as.data.frame(test_h2o[,-1]), type = 'raw') %>%
    tibble::as_tibble()
#> # A tibble: 211 x 2
#>      No     Yes
#>   <dbl>   <dbl>
#> 1 0.812 0.188
#> 2 0.940 0.0604
#> 3 0.164 0.836
#> 4 0.985 0.0152
#> 5 0.808 0.192
#> 6 0.991 0.00903
#> # ... with 205 more rows
```

Now the fun part, we create an explainer using the `lime()` function. Just pass the training data set without the "Attribution column". The form must be a data frame, which is OK since our predict_model function will switch it to an h2o object. Set`model = automl_leader` our leader model, and `bin_continuous = FALSE`. We could tell the algorithm to bin continuous variables, but this may not make sense for categorical numeric data that we didn't change to factors.

```
# Run lime() on training set
explainer <- lime::lime(
    as.data.frame(train_h2o[,-1]),
    model          = automl_leader,
    bin_continuous = FALSE)
#> Warning: Data contains numeric columns with zero variance
```

Now we run the `explain()` function, which returns our explanation. This can take a minute to run so we limit it to just the first ten rows of the test data set. We set n_labels = 1 because we care about explaining a single class. Setting n_features = 4 returns the top four features that are critical to each case. Finally, setting kernel_width = 0.5 allows us to increase the "model_r2" value by shrinking the localized evaluation.

```
# Run explain() on explainer
explanation <- lime::explain(
    as.data.frame(test_h2o[1:10,-1]),
    explainer    = explainer,
    n_labels     = 1,
    n_features   = 4,
    kernel_width = 0.5)
```

1.7 Feature Importance Visualization

The payoff for the work we put in using LIME is this feature importance plot. This allows us to visualize each of the ten cases (observations) from the test data. The top four features for each case are shown. Note that they are not the same for each case. The green bars mean that the feature supports the model conclusion, and the red bars contradict. We'll focus in on Cases with `Label = Yes`, which are predicted to have attrition. We can see a common theme with Case 3 and Case 7: Training Time, Job Role, and Over Time are among the top factors influencing attrition. These are only two cases, but they can be used to potentially generalize to the larger population as we will see next.

```
plot_features(explanation) +
    labs(title = "HR Predictive Analytics: LIME Feature Importance Visualization",
         subtitle = "Hold Out (Test) Set, First 10 Cases Shown")
```

Probability: 0.81
Explanation Fit: 0.32
Probability: 0.94
Explanation Fit: 0.30
Case: 3
Label: Yes
Probability: 0.84
Explanation Fit: 0.31
Case: 4
Label: No
Probability: 0.98
Explanation Fit: 0.31
Case: 5
Label: No
Probability: 0.81
Explanation Fit: 0.33
Case: 6
Label: No
Probability: 0.99
Explanation Fit: 0.31
Case: 7
Label: Yes
Probability: 0.81
Explanation Fit: 0.32
Case: 8
Label: No
Probability: 0.93
Explanation Fit: 0.32
Case: 9
Label: No
Probability: 0.86
Explanation Fit: 0.31
Case: 10
Label: No
Probability: 0.60
Explanation Fit: 0.29
Feature
−0.2 −0.1 0.0 0.1 0.2

1.7.1 What features are linked to employee attrition

Now we turn to our three critical features from the LIME Feature Importance Plot:

- Training Time
- Job Role
- Over Time

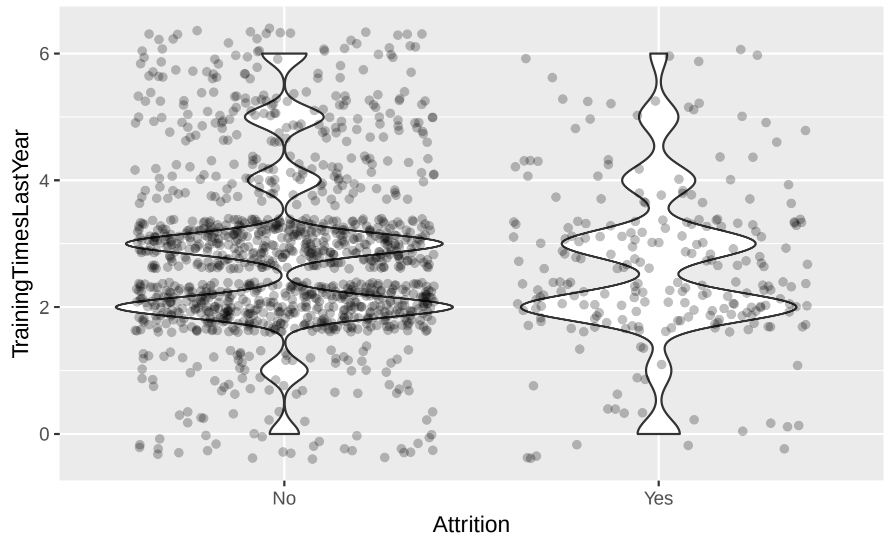
We'll subset this data and visualize to detect trends.

```r
# Focus on critical features of attrition
attrition_critical_features <- hr_data %>%
    tibble::as_tibble() %>%
    select(Attrition, TrainingTimesLastYear, JobRole, OverTime) %>%
    rowid_to_column(var = "Case")
attrition_critical_features
#> # A tibble: 1,470 x 5
#>    Case Attrition TrainingTimesLastYear JobRole               OverTime
#>   <int> <fct>                     <dbl> <fct>                 <fct>
#> 1     1 Yes                           0 Sales Executive       Yes
#> 2     2 No                            3 Research Scientist    No
#> 3     3 Yes                           3 Laboratory Technician Yes
#> 4     4 No                            3 Research Scientist    Yes
#> 5     5 No                            3 Laboratory Technician No
#> 6     6 No                            2 Laboratory Technician No
#> # ... with 1,464 more rows
```

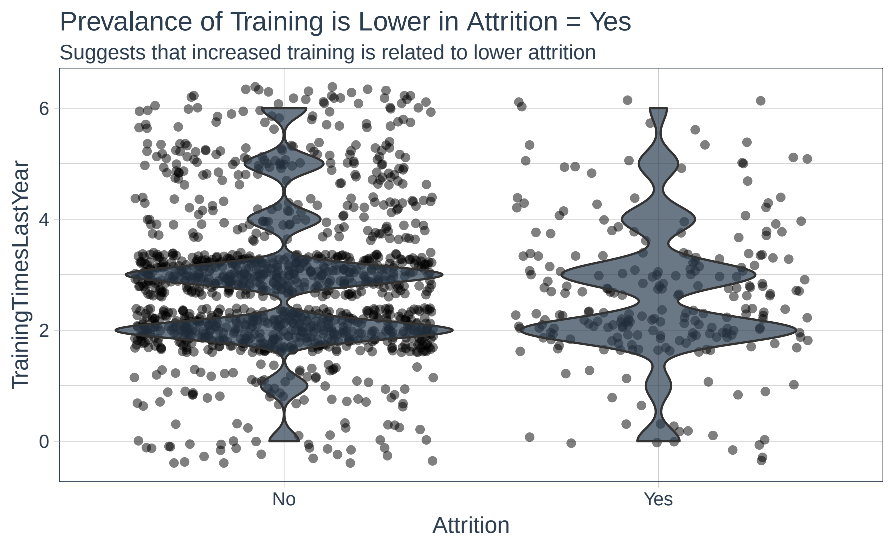
1.7.2 Training

From the violin plot, the employees that stay tend to have a large peaks at two and three trainings per year whereas the employees that leave tend to have a large peak at two trainings per year. This suggests that employees with more trainings may be less likely to leave.

```
ggplot(attrition_critical_features, aes(x = Attrition,
                                        y = TrainingTimesLastYear)) +
    geom_violin()  +
    geom_jitter(alpha = 0.25)
```



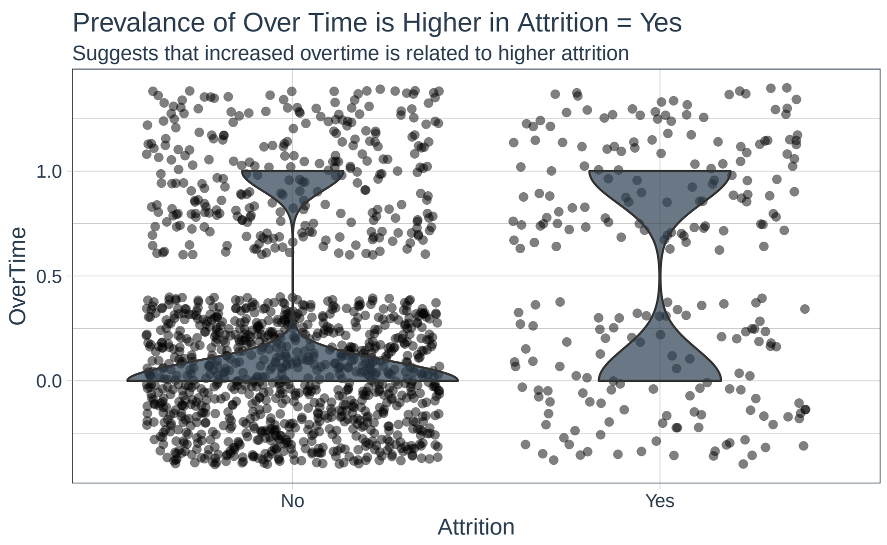
```
attrition_critical_features %>%
    ggplot(aes(Attrition, TrainingTimesLastYear)) +
    geom_jitter(alpha = 0.5, fill = palette_light()[[1]]) +
    geom_violin(alpha = 0.7, fill = palette_light()[[1]]) +
    theme_tq() +
    labs(
    title = "Prevalance of Training is Lower in Attrition = Yes",
    subtitle = "Suggests that increased training is related to lower attrition"
    )
```



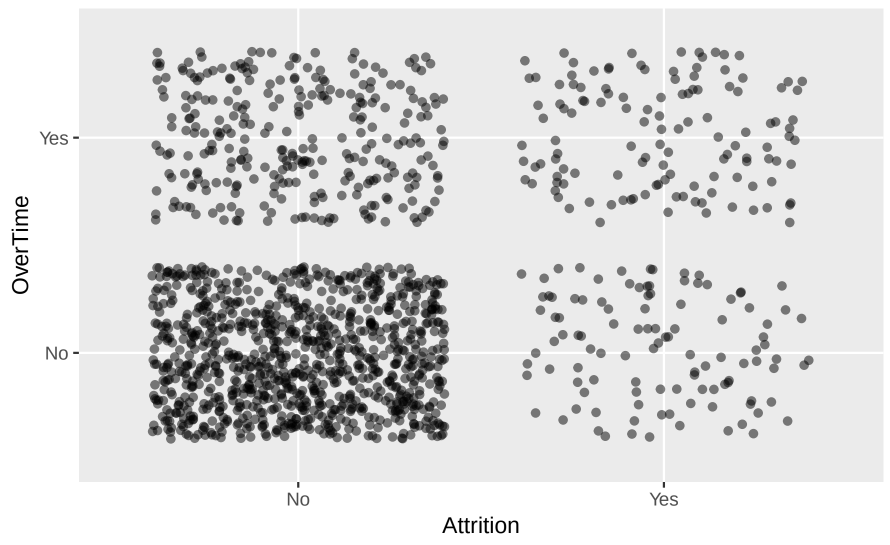
1.7.3 Overtime

The plot below shows a very interesting relationship: a very high proportion of employees that turnover are working over time. The opposite is true for employees that stay.

```
attrition_critical_features %>%
    mutate(OverTime = case_when(
                        OverTime == "Yes" ~ 1,
                        OverTime == "No"  ~ 0 )) %>%
    ggplot(aes(Attrition, OverTime)) +
    geom_jitter(alpha = 0.5, fill = palette_light()[[1]]) +
    geom_violin(alpha = 0.7, fill = palette_light()[[1]]) +
    theme_tq() +
    labs(
    title = "Prevalance of Over Time is Higher in Attrition = Yes",
    subtitle = "Suggests that increased overtime is related to higher attrition")
```



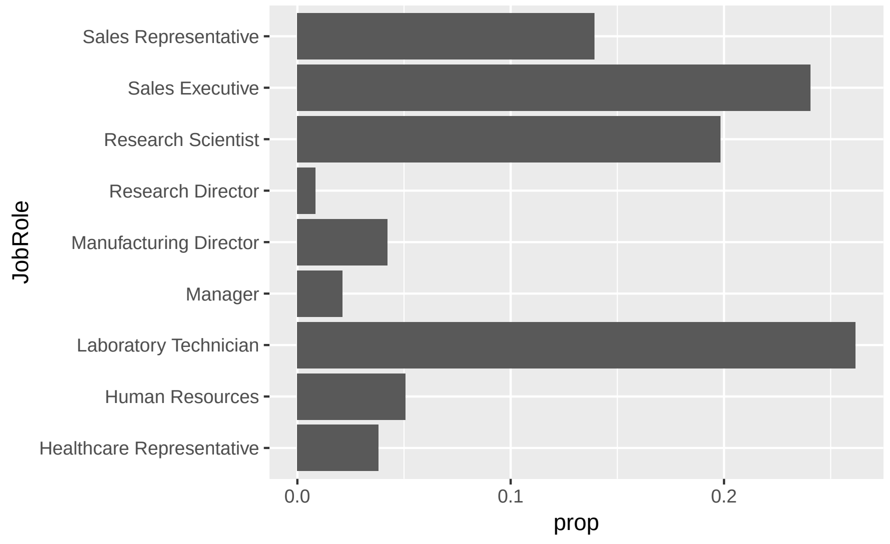
```
ggplot(attrition_critical_features, aes(x = Attrition,
                                        y = OverTime,
                                        )) +
    # geom_violin(aes(y = ..prop.., group = 1)) +
    geom_jitter(alpha = 0.5)
```



1.7.4 Job Role

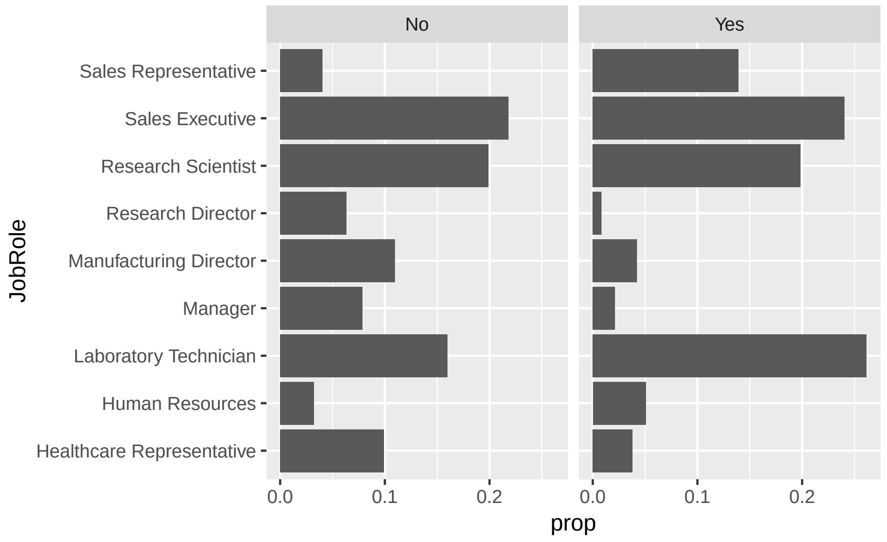
Several job roles are experiencing more turnover. Sales reps have the highest turnover at about 40% followed by Lab Technician, Human Resources, Sales Executive, and Research Scientist. It may be worthwhile to investigate what localized issues could be creating the high turnover among these groups within the organization.

```r
p <- ggplot(data = subset(attrition_critical_features, Attrition == "Yes"),
            mapping = aes(x = JobRole))
p + geom_bar(mapping = aes(y = ..prop.., group = 1)) +
    coord_flip()
```



```r
# geom_bar(mapping = aes(y = ..prop.., group = 1))
```

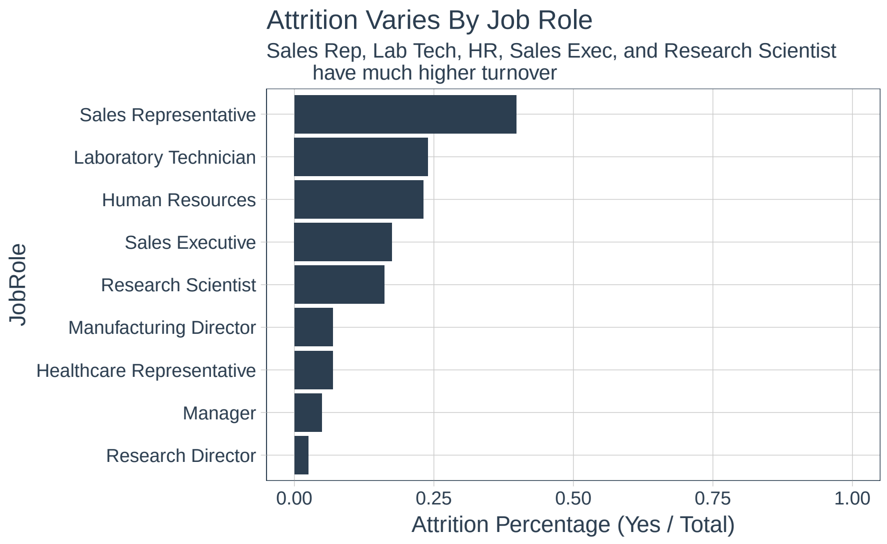
```r
p <- ggplot(data = attrition_critical_features,
            mapping = aes(x = JobRole))
p + geom_bar(mapping = aes(y = ..prop.., group = 1)) +
    coord_flip() +
    facet_wrap(Attrition ~ .)
```



```
attrition_critical_features %>%
    group_by(JobRole, Attrition) %>%
    summarize(total = n())
#> # A tibble: 18 x 3
#> # Groups:   JobRole [9]
#>   JobRole                   Attrition total
#>   <fct>                     <fct>     <int>
#> 1 Healthcare Representative No          122
#> 2 Healthcare Representative Yes           9
#> 3 Human Resources           No           40
#> 4 Human Resources           Yes          12
#> 5 Laboratory Technician     No          197
#> 6 Laboratory Technician     Yes          62
#> # ... with 12 more rows
```

```
attrition_critical_features %>%
    group_by(JobRole, Attrition) %>%
    summarize(total = n()) %>%
    spread(key = Attrition, value = total) %>%
    mutate(pct_attrition = Yes / (Yes + No))
#> # A tibble: 9 x 4
#> # Groups:   JobRole [9]
#>   JobRole                      No   Yes pct_attrition
#>   <fct>                     <int> <int>         <dbl>
#> 1 Healthcare Representative   122     9        0.0687
#> 2 Human Resources              40    12        0.231
#> 3 Laboratory Technician       197    62        0.239
#> 4 Manager                      97     5        0.0490
#> 5 Manufacturing Director      135    10        0.0690
#> 6 Research Director            78     2        0.025
#> # ... with 3 more rows
```

```
attrition_critical_features %>%
    group_by(JobRole, Attrition) %>%
    summarize(total = n()) %>%
    spread(key = Attrition, value = total) %>%
    mutate(pct_attrition = Yes / (Yes + No)) %>%
    ggplot(aes(x = forcats::fct_reorder(JobRole, pct_attrition), y = pct_attrition)) +
    geom_bar(stat = "identity", alpha = 1, fill = palette_light()[[1]]) +
    expand_limits(y = c(0, 1)) +
    coord_flip() +
    theme_tq() +
    labs(
        title = "Attrition Varies By Job Role",
        subtitle = "Sales Rep, Lab Tech, HR, Sales Exec, and Research Scientist
        have much higher turnover",
        y = "Attrition Percentage (Yes / Total)",
        x = "JobRole"
    )
```



1.8 Conclusions

There's a lot to take away from this article. We showed how you can use predictive analytics to develop sophisticated models that very accurately detect employees that are at risk of turnover. The autoML algorithm from H2O.ai worked well for classifying attrition with an accuracy around 87% on unseen / unmodeled data. We then used LIME to breakdown the complex ensemble model returned from H2O into critical features that are related to attrition. Overall, this is a really useful example where we can see how machine learning and data science can be used in business applications.

Chapter 2

Dealing with unbalanced data

2.1 Breast cancer dataset

2.2 Introduction

Source: https://shiring.github.io/machine_learning/2017/04/02/unbalanced

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
library(mice)
#>
#> Attaching package: 'mice'
#> The following objects are masked from 'package:base':
#>
#>     cbind, rbind
library(ggplot2)
```

In my last post, where I shared the code that I used to produce an example analysis to go along with my webinar on building meaningful models for disease prediction, I mentioned that it is advised to consider over- or under-sampling when you have unbalanced data sets. Because my focus in this webinar was on evaluating model performance, I did not want to add an additional layer of complexity and therefore did not further discuss how to specifically deal with unbalanced data.

But because I had gotten a few questions regarding this, I thought it would be worthwhile to explain over- and under-sampling techniques in more detail and show how you can very easily implement them with `caret`.

2.3 Read and process the data

```
bc_data <- read.table(file.path(data_raw_dir, "breast-cancer-wisconsin.data"),
                      header = FALSE, sep = ",")
```

```r
colnames(bc_data) <- c("sample_code_number", "clump_thickness",
                       "uniformity_of_cell_size", "uniformity_of_cell_shape",
                       "marginal_adhesion", "single_epithelial_cell_size",
                       "bare_nuclei", "bland_chromatin", "normal_nucleoli",
                       "mitosis", "classes")

bc_data$classes <- ifelse(bc_data$classes == "2", "benign",
                          ifelse(bc_data$classes == "4", "malignant", NA))

bc_data[bc_data == "?"] <- NA

# how many NAs are in the data
length(which(is.na(bc_data)))
#> [1] 16
```

```r
# impute missing data

# skip columns: sample_code_number and classes
bc_data[,2:10] <- apply(bc_data[, 2:10], 2, function(x) as.numeric(as.character(x)))

# impute but stay mute
dataset_impute <- mice(bc_data[, 2:10],  print = FALSE)

# bind "classes" with the rest. skip "sample_code_number"
bc_data <- cbind(bc_data[, 11, drop = FALSE],
                 mice::complete(dataset_impute, action = 1))

bc_data$classes <- as.factor(bc_data$classes)
```

2.3.1 Unbalanced data

In this context, unbalanced data refers to classification problems where we have unequal instances for different classes. Having unbalanced data is actually very common in general, but it is especially prevalent when working with disease data where we usually have more healthy control samples than disease cases. Even more extreme unbalance is seen with fraud detection, where e.g. most credit card uses are okay and only very few will be fraudulent. In the example I used for my webinar, a *breast cancer* dataset, we had about twice as many benign than malignant samples.

```r
# how many benign and malignant cases are there?
summary(bc_data$classes)
#>    benign malignant
#>       458       241
```

2.3.1.1 Why is unbalanced data a problem in machine learning?

Most machine learning classification algorithms are sensitive to unbalance in the predictor classes. Let's consider an even more extreme example than our breast cancer dataset: assume we had 10 malignant vs 90 benign samples. A machine learning model that has been trained and tested on such a dataset could now predict "benign" for all samples and still gain a very high accuracy. An unbalanced dataset will bias the prediction model towards the more common class!

2.3.1.2 How to balance data for modeling

The basic theoretical concepts behind over- and under-sampling are very simple:

With under-sampling, we randomly select a subset of samples from the class with more instances to match the number of samples coming from each class. In our example, we would randomly pick 241 out of the 458 benign cases. The main disadvantage of under-sampling is that we lose potentially relevant information from the left-out samples.

With oversampling, we randomly duplicate samples from the class with fewer instances or we generate additional instances based on the data that we have, so as to match the number of samples in each class. While we avoid losing information with this approach, we also run the risk of overfitting our model as we are more likely to get the same samples in the training and in the test data, i.e. the test data is no longer independent from training data. This would lead to an overestimation of our model's performance and generalizability.

In reality though, we should not simply perform over- or under-sampling on our training data and then run the model. We need to account for cross-validation and perform over- or under-sampling on each fold independently to get an honest estimate of model performance!

2.3.1.3 Modeling the original unbalanced data

Here is the same model I used in my webinar example: I randomly divide the data into training and test sets (stratified by class) and perform Random Forest modeling with 10 x 10 repeated cross-validation. Final model performance is then measured on the test set.

```r
set.seed(42)
index <- createDataPartition(bc_data$classes, p = 0.7, list = FALSE)
train_data <- bc_data[index, ]
test_data  <- bc_data[-index, ]
```

```r
set.seed(42)
model_rf <- caret::train(classes ~ .,
                         data = train_data,
                         method = "rf",
                         preProcess = c("scale", "center"),
                         trControl = trainControl(method = "repeatedcv",
                                                  number = 10,
                                                  repeats = 10,
                                                  verboseIter = FALSE))
```

```r
final <- data.frame(actual = test_data$classes,
                    predict(model_rf,
                            newdata = test_data,
                            type = "prob"))

final$predict <- ifelse(final$benign > 0.5, "benign", "malignant")
```

```r
final_predict <- as.factor(final$predict)
test_data_classes <- as.factor(test_data$classes)

cm_original <- confusionMatrix(final_predict, test_data_classes)
cm_original$byClass['Sensitivity']
#> Sensitivity
#>       0.978
```

2.4 Under-sampling

Luckily, `caret` makes it very easy to incorporate over- and under-sampling techniques with cross-validation resampling. We can simply add the sampling option to our `trainControl` and choose down for under- (also called `down-`) sampling. The rest stays the same as with our original model.

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                     number = 10,
                     repeats = 10,
                     verboseIter = FALSE,
                     sampling = "down")


model_rf_under <- caret::train(classes ~ .,
                         data = train_data,
                         method = "rf",
                         preProcess = c("scale", "center"),
                         trControl = ctrl)
```

```
final_under <- data.frame(actual = test_data$classes,
                    predict(model_rf_under,
                           newdata = test_data,
                           type = "prob"))

final_under$predict <- ifelse(final_under$benign > 0.5, "benign", "malignant")
```

```
final_under_predict <- as.factor(final_under$predict)
test_data_classes <- test_data$classes

cm_under <- confusionMatrix(final_under_predict, test_data_classes)
cm_under$byClass['Sensitivity']
#> Sensitivity
#>       0.978
```

2.5 Oversampling

For over- (also called up-) sampling we simply specify sampling = “up”.

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                     number = 10,
                     repeats = 10,
                     verboseIter = FALSE,
                     sampling = "up")


model_rf_over <- caret::train(classes ~ .,
                         data = train_data,
                         method = "rf",
                         preProcess = c("scale", "center"),
                         trControl = ctrl)
```

```
final_over <- data.frame(actual = test_data$classes,
                          predict(model_rf_over,
                                  newdata = test_data,
                                  type = "prob"))

final_over$predict <- ifelse(final_over$benign > 0.5, "benign", "malignant")
```

```
final_over_predict <- as.factor(final_over$predict)
test_data_classes <- test_data$classes

cm_over <- confusionMatrix(final_over_predict, test_data_classes)
cm_over$byClass['Sensitivity']
#> Sensitivity
#>       0.978
```

2.5.1 ROSE

Besides over- and under-sampling, there are hybrid methods that combine under-sampling with the generation of additional data. Two of the most popular are ROSE and SMOTE.

> From Nicola Lunardon, Giovanna Menardi and Nicola Torelli's "ROSE: A Package for Binary Imbalanced Learning" (R Journal, 2014, Vol. 6 Issue 1, p. 79): "The ROSE package provides functions to deal with binary classification problems in the presence of imbalanced classes. Artificial balanced samples are generated according to a smoothed bootstrap approach and allow for aiding both the phases of estimation and accuracy evaluation of a binary classifier in the presence of a rare class. Functions that implement more traditional remedies for the class imbalance and different metrics to evaluate accuracy are also provided. These are estimated by holdout, bootstrap, or cross-validation methods."

You implement them the same way as before, this time choosing sampling = "rose"…

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                     number = 10,
                     repeats = 10,
                     verboseIter = FALSE,
                     sampling = "rose")

model_rf_rose <- caret::train(classes ~ .,
                              data = train_data,
                              method = "rf",
                              preProcess = c("scale", "center"),
                              trControl = ctrl)
#> Loaded ROSE 0.0-3
```

```
final_rose <- data.frame(actual = test_data$classes,
                         predict(model_rf_rose,
                                 newdata = test_data,
                                 type = "prob"))

final_rose$predict <- ifelse(final_rose$benign > 0.5, "benign", "malignant")
```

```
cm_rose <- confusionMatrix(as.factor(final_rose$predict),
                           as.factor(test_data$classes))
cm_rose$byClass['Sensitivity']
```

```
#> Sensitivity
#>       0.985
```

2.5.2 SMOTE

… or by choosing sampling = "smote" in the `trainControl` settings.

> From Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall and W. Philip Kegelmeyer's "SMOTE: Synthetic Minority Over-sampling Technique" (Journal of Artificial Intelligence Research, 2002, Vol. 16, pp. 321–357): "This paper shows that a combination of our method of over-sampling the minority (abnormal) class and under-sampling the majority (normal) class can achieve better classifier performance (in ROC space) than only under-sampling the majority class. This paper also shows that a combination of our method of over-sampling the minority class and under-sampling the majority class can achieve better classifier performance (in ROC space) than varying the loss ratios in Ripper or class priors in Naive Bayes. Our method of over-sampling the minority class involves creating synthetic minority class examples."

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                     number = 10,
                     repeats = 10,
                     verboseIter = FALSE,
                     sampling = "smote")

model_rf_smote <- caret::train(classes ~ .,
                               data = train_data,
                               method = "rf",
                               preProcess = c("scale", "center"),
                               trControl = ctrl)
#> Loading required package: grid
#> Registered S3 method overwritten by 'xts':
#>   method     from
#>   as.zoo.xts zoo
#> Registered S3 method overwritten by 'quantmod':
#>   method            from
#>   as.zoo.data.frame zoo
```

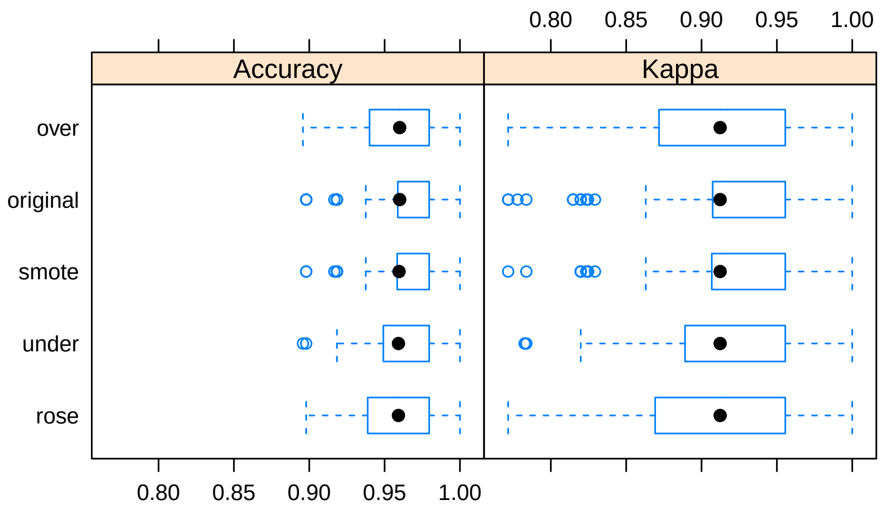
```
final_smote <- data.frame(actual = test_data$classes,
                         predict(model_rf_smote,
                                 newdata = test_data,
                                 type = "prob"))

final_smote$predict <- ifelse(final_smote$benign > 0.5, "benign", "malignant")
```

```
cm_smote <- confusionMatrix(as.factor(final_smote$predict),
                            as.factor(test_data$classes))
cm_smote$byClass['Sensitivity']
#> Sensitivity
#>       0.978
```

2.6 Predictions

Now let's compare the predictions of all these models:

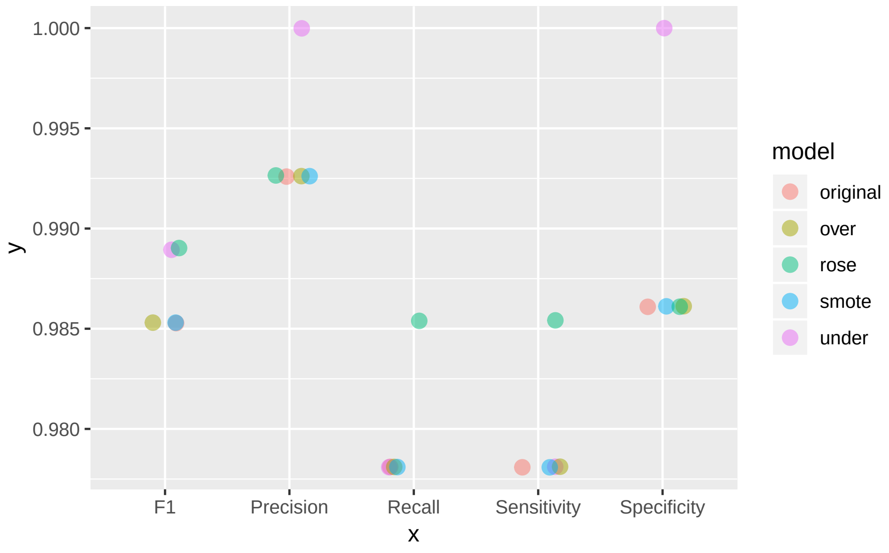
```r
models <- list(
                original = model_rf,
                under = model_rf_under,
                over = model_rf_over,
                smote = model_rf_smote,
                rose = model_rf_rose)

resampling <- resamples(models)
bwplot(resampling)
```



```r
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
comparison <- data.frame(model = names(models),
                         Sensitivity = rep(NA, length(models)),
                         Specificity = rep(NA, length(models)),
                         Precision   = rep(NA, length(models)),
                         Recall      = rep(NA, length(models)),
                         F1          = rep(NA, length(models)))

for (name in names(models)) {
    cm_model <- get(paste0("cm_", name))
    comparison[comparison$model==name, ] <- filter(comparison, model==name) %>%
    mutate(Sensitivity = cm_model$byClass["Sensitivity"],
           Specificity = cm_model$byClass["Specificity"],
           Precision   = cm_model$byClass["Precision"],
           Recall      = cm_model$byClass["Recall"],
           F1          = cm_model$byClass["F1"]
```

```
  )
}

print(comparison)
#>      model Sensitivity Specificity Precision Recall    F1
#> 1 original       0.978       0.986     0.993  0.978 0.985
#> 2    under       0.978       1.000     1.000  0.978 0.989
#> 3     over       0.978       0.986     0.993  0.978 0.985
#> 4    smote       0.978       0.986     0.993  0.978 0.985
#> 5     rose       0.985       0.986     0.993  0.985 0.989
```

```
library(tidyr)
#>
#> Attaching package: 'tidyr'
#> The following object is masked from 'package:mice':
#>
#>     complete
comparison %>%
  gather(x, y, Sensitivity:F1) %>%
  ggplot(aes(x = x, y = y, color = model)) +
    geom_jitter(width = 0.2, alpha = 0.5, size = 3)
```



With this small dataset, we can already see how the different techniques can influence model performance. **Sensitivity** (or **recall**) describes the proportion of benign cases that have been predicted correctly, while **specificity** describes the proportion of malignant cases that have been predicted correctly. **Precision describes** the true positives, i.e. the proportion of benign predictions that were actual from benign samples. **F1** is the weighted average of precision and sensitivity/ recall.

2.7 Final notes

Here, all four methods improved specificity and precision compared to the original model. Under-sampling, over-sampling and ROSE additionally improved precision and the F1 score.

This post shows a simple example of how to correct for unbalance in datasets for machine learning. For more advanced instructions and potential caveats with these techniques, check out the excellent `caret` documentation.

If you are interested in more machine learning posts, check out the category listing for machine_learning on my blog.

Chapter 3

Ten different methods to assess Variable Importance

3.1 Glaucoma dataset

Source: https://www.machinelearningplus.com/machine-learning/feature-selection/

3.2 Introduction

In real-world datasets, it is fairly common to have columns that are nothing but noise.

You are better off getting rid of such variables because of the memory space they occupy, the time and the computational esources it is going to cost, especially in large datasets.

Sometimes, you have a variable that makes business sense, but you are not sure if it actually helps in predicting the Y. You also need to consider the fact that, a feature that could be useful in one ML algorithm (say a decision tree) may go underrepresented or unused by another (like a regression model).

Having said that, it is still possible that a variable that shows poor signs of helping to explain the response variable (Y), can turn out to be significantly useful in the presence of (or combination with) other predictors. What I mean by that is, a variable might have a low correlation value of (~0.2) with Y. But in the presence of other variables, it can help to explain certain patterns/phenomenon that other variables can't explain.

In such cases, it can be hard to make a call whether to include or exclude such variables.

The strategies we are about to discuss can help fix such problems. Not only that, it will also help understand if a particular variable is important or not and how much it is contributing to the model

An important caveat. It is always best to have variables that have sound business logic backing the inclusion of a variable and rely solely on variable importance metrics.

Alright. Let's load up the 'Glaucoma' dataset where the goal is to predict if a patient has Glaucoma or not based on 63 different physiological measurements. You can directly run the codes or download the dataset here.

A lot of interesting examples ahead. Let's get started.

```r
# Load Packages and prepare dataset
library(TH.data)
#> Loading required package: survival
#> Loading required package: MASS
#>
#> Attaching package: 'TH.data'
#> The following object is masked from 'package:MASS':
#>
#>     geyser
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#>
#> Attaching package: 'caret'
#> The following object is masked from 'package:survival':
#>
#>     cluster
library(tictoc)

data("GlaucomaM", package = "TH.data")
trainData <- GlaucomaM
head(trainData)
#>      ag    at    as    an    ai   eag   eat   eas   ean   eai  abrg  abrt
#> 2  2.22 0.354 0.580 0.686 0.601 1.267 0.336 0.346 0.255 0.331 0.479 0.260
#> 43 2.68 0.475 0.672 0.868 0.667 2.053 0.440 0.520 0.639 0.454 1.090 0.377
#> 25 1.98 0.343 0.508 0.624 0.504 1.200 0.299 0.396 0.259 0.246 0.465 0.209
#> 65 1.75 0.269 0.476 0.525 0.476 0.612 0.147 0.017 0.044 0.405 0.170 0.062
#> 70 2.99 0.599 0.686 1.039 0.667 2.513 0.543 0.607 0.871 0.492 1.800 0.431
#> 16 2.92 0.483 0.763 0.901 0.770 2.200 0.462 0.637 0.504 0.597 1.311 0.394
#>     abrs  abrn  abri    hic   mhcg  mhct   mhcs   mhcn   mhci   phcg
#> 2  0.107 0.014 0.098  0.214  0.111 0.412  0.036  0.105 -0.022 -0.139
#> 43 0.257 0.212 0.245  0.382  0.140 0.338  0.104  0.080  0.109 -0.015
#> 25 0.112 0.041 0.103  0.195  0.062 0.356  0.045 -0.009 -0.048 -0.149
#> 65 0.000 0.000 0.108 -0.030 -0.015 0.074 -0.084 -0.050  0.035 -0.182
#> 70 0.494 0.601 0.274  0.383  0.089 0.233  0.145  0.023  0.007 -0.131
#> 16 0.365 0.251 0.301  0.442  0.128 0.375  0.049  0.111  0.052 -0.088
#>      phct   phcs   phcn   phci   hvc  vbsg  vbst  vbss  vbsn  vbsi  vasg
#> 2   0.242 -0.053  0.010 -0.139 0.613 0.303 0.103 0.088 0.022 0.090 0.062
#> 43  0.296 -0.015 -0.015  0.036 0.382 0.676 0.181 0.186 0.141 0.169 0.029
#> 25  0.206 -0.092 -0.081 -0.149 0.557 0.300 0.084 0.088 0.046 0.082 0.036
#> 65 -0.097 -0.125 -0.138 -0.182 0.373 0.048 0.011 0.000 0.000 0.036 0.070
#> 70  0.163  0.055 -0.131 -0.115 0.405 0.889 0.151 0.253 0.330 0.155 0.020
#> 16  0.281 -0.067 -0.062 -0.088 0.507 0.972 0.213 0.316 0.197 0.246 0.043
#>     vast  vass  vasn  vasi  vbrg  vbrt  vbrs  vbrn  vbri  varg  vart  vars
#> 2  0.000 0.011 0.032 0.018 0.075 0.039 0.021 0.002 0.014 0.756 0.009 0.209
#> 43 0.001 0.007 0.011 0.010 0.370 0.127 0.099 0.050 0.093 0.410 0.006 0.105
#> 25 0.002 0.004 0.016 0.013 0.081 0.034 0.019 0.007 0.021 0.565 0.014 0.132
#> 65 0.005 0.030 0.033 0.002 0.005 0.001 0.000 0.000 0.004 0.380 0.032 0.147
#> 70 0.001 0.004 0.008 0.007 0.532 0.103 0.173 0.181 0.075 0.228 0.011 0.026
#> 16 0.001 0.005 0.028 0.009 0.467 0.136 0.148 0.078 0.104 0.540 0.008 0.133
#>     varn  vari   mdg   mdt   mds   mdn   mdi    tmg    tmt    tms    tmn
#> 2  0.298 0.240 0.705 0.637 0.738 0.596 0.691 -0.236 -0.018 -0.230 -0.510
#> 43 0.181 0.117 0.898 0.850 0.907 0.771 0.940 -0.211 -0.014 -0.165 -0.317
#> 25 0.243 0.177 0.687 0.643 0.689 0.684 0.700 -0.185 -0.097 -0.235 -0.337
#> 65 0.151 0.050 0.207 0.171 0.022 0.046 0.221 -0.148 -0.035 -0.449 -0.217
#> 70 0.105 0.087 0.721 0.638 0.730 0.730 0.640 -0.052 -0.105  0.084 -0.012
#> 16 0.232 0.167 0.927 0.842 0.953 0.906 0.898 -0.040  0.087  0.018 -0.094
#>       tmi    mr   rnf  mdic   emd    mv  Class
#> 2  -0.158 0.841 0.410 0.137 0.239 0.035 normal
#> 43 -0.192 0.924 0.256 0.252 0.329 0.022 normal
#> 25 -0.020 0.795 0.378 0.152 0.250 0.029 normal
#> 65 -0.091 0.746 0.200 0.027 0.078 0.023 normal
#> 70 -0.054 0.977 0.193 0.297 0.354 0.034 normal
#> 16 -0.051 0.965 0.339 0.333 0.442 0.028 normal
```

3.3 1. Boruta

Boruta is a feature ranking and selection algorithm based on random forests algorithm.

The advantage with Boruta is that it clearly decides if a variable is important or not and helps to select variables that are statistically significant. Besides, you can adjust the strictness of the algorithm by adjusting the p values that defaults to 0.01 and the `maxRuns`.

`maxRuns` is the number of times the algorithm is run. The higher the `maxRuns` the more selective you get in picking the variables. The default value is 100.

In the process of deciding if a feature is important or not, some features may be marked by Boruta as 'Tentative'. Sometimes increasing the maxRuns can help resolve the 'Tentativeness' of the feature.

Lets see an example based on the Glaucoma dataset from `TH.data` package that I created earlier.

```
# install.packages('Boruta')
library(Boruta)
#> Loading required package: ranger
```

The `boruta` function uses a formula interface just like most predictive modeling functions. So the first argument to `boruta()` is the formula with the response variable on the left and all the predictors on the right.

By placing a dot, all the variables in `trainData` other than Class will be included in the model.

The `doTrace` argument controls the amount of output printed to the console. Higher the value, more the log details you get. So save space I have set it to 0, but try setting it to 1 and 2 if you are running the code.

Finally the output is stored in `boruta_output`.

```
# Perform Boruta search
boruta_output <- Boruta(Class ~ ., data=na.omit(trainData), doTrace=0)
```

Let's see what the boruta_output contains.

```
names(boruta_output)
#>  [1] "finalDecision" "ImpHistory"    "pValue"        "maxRuns"
#>  [5] "light"         "mcAdj"         "timeTaken"     "roughfixed"
#>  [9] "call"          "impSource"
```

```
# Get significant variables including tentatives
boruta_signif <- getSelectedAttributes(boruta_output, withTentative = TRUE)
```

```r
print(boruta_signif)
#>  [1] "as"   "ai"   "eas"  "ean"  "abrg" "abrs" "abrn" "abri" "hic"  "mhcg"
#> [11] "mhcs" "mhcn" "mhci" "phcg" "phcn" "phci" "hvc"  "vbsg" "vbss" "vbsn"
#> [21] "vbsi" "vasg" "vass" "vasi" "vbrg" "vbrs" "vbrn" "vbri" "varg" "vart"
#> [31] "vars" "varn" "vari" "mdn"  "tmg"  "tmt"  "tms"  "tmi"  "mr"   "rnf"
#> [41] "mdic" "emd"
```

If you are not sure about the tentative variables being selected for granted, you can choose a `TentativeRoughFix` on `boruta_output`.

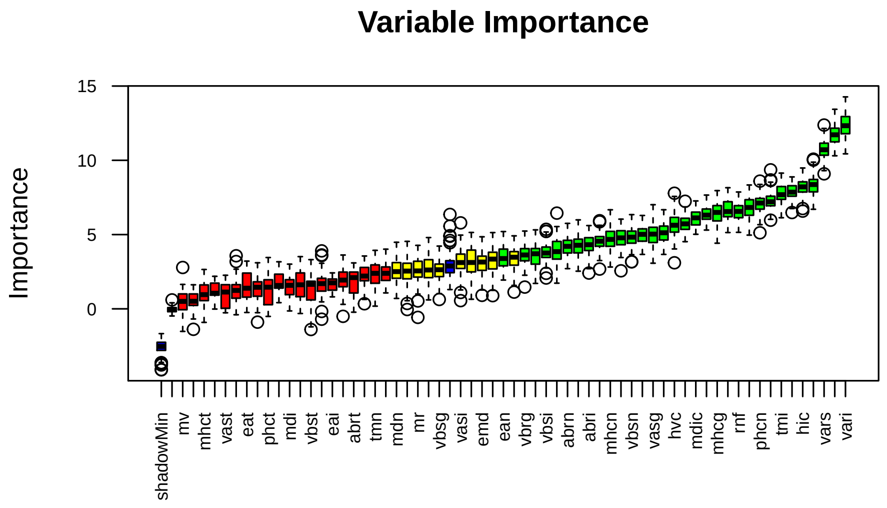
```r
# Do a tentative rough fix
roughFixMod <- TentativeRoughFix(boruta_output)
boruta_signif <- getSelectedAttributes(roughFixMod)
print(boruta_signif)
#>  [1] "as"   "ai"   "ean"  "abrg" "abrs" "abrn" "abri" "hic"  "mhcg" "mhcn"
#> [11] "mhci" "phcg" "phcn" "phci" "hvc"  "vbsn" "vbsi" "vasg" "vass" "vasi"
#> [21] "vbrg" "vbrs" "vbrn" "vbri" "varg" "vart" "vars" "varn" "vari" "mdn"
#> [31] "tmg"  "tms"  "tmi"  "mr"   "rnf"  "mdic"
```

There you go. Boruta has decided on the 'Tentative' variables on our behalf. Let's find out the importance scores of these variables.

```r
# Variable Importance Scores
imps <- attStats(roughFixMod)
imps2 = imps[imps$decision != 'Rejected', c('meanImp', 'decision')]
head(imps2[order(-imps2$meanImp), ])  # descending sort
#>      meanImp  decision
#> vari   12.37 Confirmed
#> varg   11.74 Confirmed
#> vars   10.74 Confirmed
#> phci    8.34 Confirmed
#> hic     8.21 Confirmed
#> varn    7.88 Confirmed
```

Let's plot it to see the importances of these variables.

```r
# Plot variable importance
plot(boruta_output, cex.axis=.7, las=2, xlab="", main="Variable Importance")
```



This plot reveals the importance of each of the features.

The columns in green are 'confirmed' and the ones in red are not. There are couple of blue bars representing `ShadowMax` and `ShadowMin`. They are not actual features, but are used by the boruta algorithm to decide if a variable is important or not.

3.4 Variable Importance from Machine Learning Algorithms

Another way to look at feature selection is to consider variables most used by various ML algorithms the most to be important.

Depending on how the machine learning algorithm learns the relationship between X's and Y, different machine learning algorithms may possibly end up using different variables (but mostly common vars) to various degrees.

What I mean by that is, the variables that proved useful in a tree-based algorithm like `rpart`, can turn out to be less useful in a regression-based model. So all variables need not be equally useful to all algorithms.

So how do we find the variable importance for a given ML algo?

`train()` the desired model using the caret package. Then, use `varImp()` to determine the feature importances.

You may want to try out multiple algorithms, to get a feel of the usefulness of the features across algos.

3.4.1 rpart

```
# Train an rpart model and compute variable importance.
library(caret)
set.seed(100)
rPartMod <- train(Class ~ .,
                  data=trainData,
                  method="rpart")

rpartImp <- varImp(rPartMod)
print(rpartImp)
#> rpart variable importance
#>
#>   only 20 most important variables shown (out of 62)
#>
#>      Overall
#> varg   100.0
#> vari    93.2
#> vars    85.2
#> varn    76.9
#> tmi     72.3
#> mhcn     0.0
#> as       0.0
#> phcs     0.0
#> vbst     0.0
#> abrt     0.0
#> vbsg     0.0
#> eai      0.0
#> vbrs     0.0
#> vbsi     0.0
#> eag      0.0
#> tmt      0.0
#> phcn     0.0
#> vart     0.0
#> mds      0.0
#> an       0.0
```

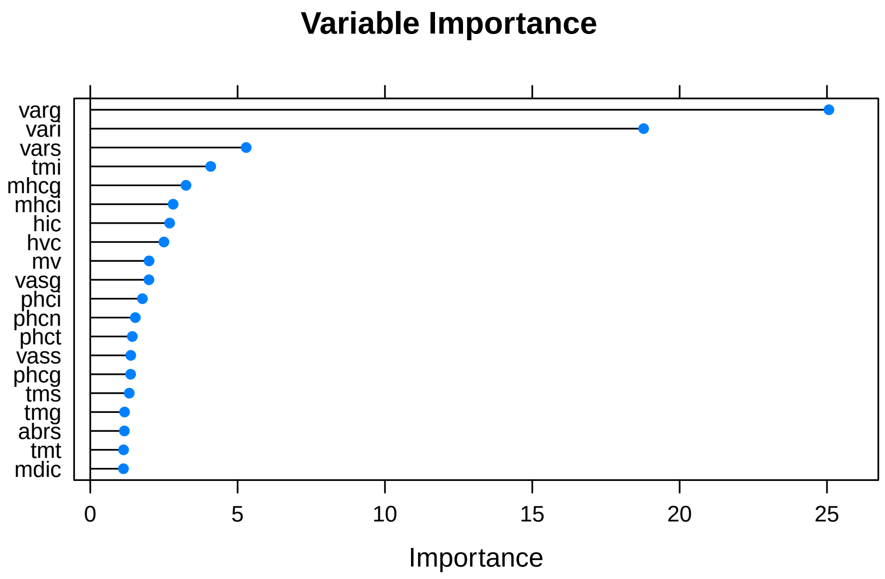
Only 5 of the 63 features was used by rpart and if you look closely, the 5 variables used here are in the top 6 that boruta selected.

Let's do one more: the variable importances from Regularized Random Forest (RRF) algorithm.

3.4.2 Regularized Random Forest (RRF)

```
tic()
# Train an RRF model and compute variable importance.
set.seed(100)
rrfMod <- train(Class ~ .,
                data = trainData,
                method = "RRF")
#> Registered S3 method overwritten by 'RRF':
#>   method      from
#>   plot.margin randomForest

rrfImp <- varImp(rrfMod, scale=F)
toc()
#> 345.177 sec elapsed
rrfImp
#> RRF variable importance
#>
#>   only 20 most important variables shown (out of 62)
#>
#>      Overall
#> varg   25.07
#> vari   18.78
#> vars    5.29
#> tmi     4.09
#> mhcg    3.25
#> mhci    2.81
#> hic     2.69
#> hvc     2.50
#> mv      2.00
#> vasg    1.99
#> phci    1.77
#> phcn    1.53
#> phct    1.43
#> vass    1.37
#> phcg    1.37
#> tms     1.32
#> tmg     1.16
#> abrs    1.16
#> tmt     1.13
#> mdic    1.13
```

```
plot(rrfImp, top = 20, main='Variable Importance')
```



The topmost important variables are pretty much from the top tier of Boruta's selections.

Some of the other algorithms available in `train()` that you can use to compute varImp are the following:

`ada, AdaBag, AdaBoost.M1, adaboost, bagEarth, bagEarthGCV, bagFDA, bagFDAGCV, bartMachine, blasso, BstLm`

3.5 Lasso Regression

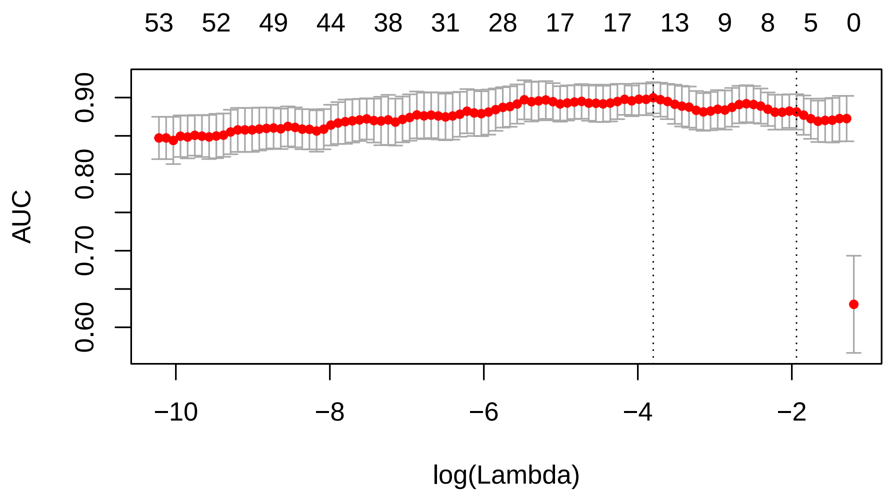
Least Absolute Shrinkage and Selection Operator (LASSO) regression is a type of regularization method that penalizes with L1-norm.

It basically imposes a cost to having large weights (value of coefficients). And its called L1 regularization, because the cost added, is proportional to the absolute value of weight coefficients.

As a result, in the process of shrinking the coefficients, it eventually reduces the coefficients of certain unwanted features all the to zero. That is, it removes the unneeded variables altogether.

So effectively, LASSO regression can be considered as a variable selection technique as well.

```
library(glmnet)
#> Loading required package: Matrix
#> Loading required package: foreach
#> Loaded glmnet 2.0-16

# online data
# trainData <- read.csv('https://raw.githubusercontent.com/selva86/datasets/master/GlaucomaM.csv')

trainData <- read.csv(file.path(data_raw_dir, "glaucoma.csv"))

x <- as.matrix(trainData[,-63]) # all X vars
y <- as.double(as.matrix(ifelse(trainData[, 63]=='normal', 0, 1))) # Only Class

# Fit the LASSO model (Lasso: Alpha = 1)
set.seed(100)
cv.lasso <- cv.glmnet(x, y, family='binomial', alpha=1, parallel=TRUE, standardize=TRUE, type.measure='a
#> Warning: executing %dopar% sequentially: no parallel backend registered

# Results
plot(cv.lasso)
```



Let's see how to interpret this plot.

The X axis of the plot is the log of `lambda`. That means when it is 2 here, the lambda value is actually 100.

The numbers at the top of the plot show how many predictors were included in the model. The position of red dots along the Y-axis tells what `AUC` we got when you include as many variables shown on the top x-axis.

You can also see two dashed vertical lines.

The first one on the left points to the lambda with the lowest mean squared error. The one on the right point to the number of variables with the highest deviance within 1 standard deviation.

The best lambda value is stored inside 'cv.lasso$lambda.min'.

```
# plot(cv.lasso$glmnet.fit, xvar="lambda", label=TRUE)
cat('Min Lambda: ', cv.lasso$lambda.min, '\n 1Sd Lambda: ', cv.lasso$lambda.1se)
#> Min Lambda:  0.0224
#>  1Sd Lambda:  0.144
df_coef <- round(as.matrix(coef(cv.lasso, s=cv.lasso$lambda.min)), 2)

# See all contributing variables
df_coef[df_coef[, 1] != 0, ]
#> (Intercept)          as        mhci        phci         hvc        vast
#>        2.68       -1.59        3.85        5.60       -2.41      -13.90
#>        vars        vari         mdn         mdi         tmg         tms
#>      -20.18       -1.58        0.50        0.99        0.06        2.56
#>         tmi
#>        2.23
```

The above output shows what variables LASSO considered important. A high positive or low negative implies more important is that variable.

3.6 Step wise Forward and Backward Selection

Stepwise regression can be used to select features if the Y variable is a numeric variable. It is particularly used in selecting best linear regression models.

It searches for the best possible regression model by iteratively selecting and dropping variables to arrive at a model with the lowest possible AIC.

It can be implemented using the `step()` function and you need to provide it with a lower model, which is the base model from which it won't remove any features and an upper model, which is a full model that has all possible features you want to have.

Our case is not so complicated (< 20 vars), so lets just do a simple stepwise in 'both' directions.

I will use the `ozone` dataset for this where the objective is to predict the `ozone_reading` based on other weather related observations.

```
# Load data
# online
# trainData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/ozone1.csv",
#                       stringsAsFactors=F)
trainData <- read.csv(file.path(data_raw_dir, "ozone1.csv"))
print(head(trainData))
#>   Month Day_of_month Day_of_week ozone_reading pressure_height Wind_speed
#> 1     1            1           4             3            5480          8
#> 2     1            2           5             3            5660          6
#> 3     1            3           6             3            5710          4
#> 4     1            4           7             5            5700          3
#> 5     1            5           1             5            5760          3
#> 6     1            6           2             6            5720          4
#>   Humidity Temperature_Sandburg Temperature_ElMonte Inversion_base_height
#> 1       20                 40.5                39.8                  5000
#> 2       41                 38.0                46.7                  4109
#> 3       28                 40.0                49.5                  2693
#> 4       37                 45.0                52.3                   590
#> 5       51                 54.0                45.3                  1450
#> 6       69                 35.0                49.6                  1568
#>   Pressure_gradient Inversion_temperature Visibility
#> 1               -15                  30.6        200
#> 2               -14                  48.0        300
#> 3               -25                  47.7        250
#> 4               -24                  55.0        100
#> 5                25                  57.0         60
#> 6                15                  53.8         60
```

The data is ready. Let's perform the stepwise.

```
# Step 1: Define base intercept only model
base.mod <- lm(ozone_reading ~ 1 , data=trainData)

# Step 2: Full model with all predictors
all.mod <- lm(ozone_reading ~ . , data= trainData)

# Step 3: Perform step-wise algorithm. direction='both' implies both forward and backward stepwise
stepMod <- step(base.mod, scope = list(lower = base.mod, upper = all.mod), direction = "both", trace =

# Step 4: Get the shortlisted variable.
```

```
shortlistedVars <- names(unlist(stepMod[[1]]))
shortlistedVars <- shortlistedVars[!shortlistedVars %in% "(Intercept)"] # remove intercept

# Show
print(shortlistedVars)
#> [1] "Temperature_Sandburg"  "Humidity"              "Temperature_ElMonte"
#> [4] "Month"                 "pressure_height"       "Inversion_base_height"
```

The selected model has the above 6 features in it.

But if you have too many features (> 100) in training data, then it might be a good idea to split the dataset into chunks of 10 variables each with Y as mandatory in each dataset. Loop through all the chunks and collect the best features.

We are doing it this way because some variables that came as important in a training data with fewer features may not show up in a linear reg model built on lots of features.

Finally, from a pool of shortlisted features (from small chunk models), run a full stepwise model to get the final set of selected features.

You can take this as a learning assignment to be solved within 20 minutes.

3.7 Relative Importance from Linear Regression

This technique is specific to linear regression models.

Relative importance can be used to assess which variables contributed how much in explaining the linear model's R-squared value. So, if you sum up the produced importances, it will add up to the model's R-sq value.

In essence, it is not directly a feature selection method, because you have already provided the features that go in the model. But after building the model, the `relaimpo` can provide a sense of how important each feature is in contributing to the R-sq, or in other words, in 'explaining the Y variable'.

So, how to calculate relative importance?

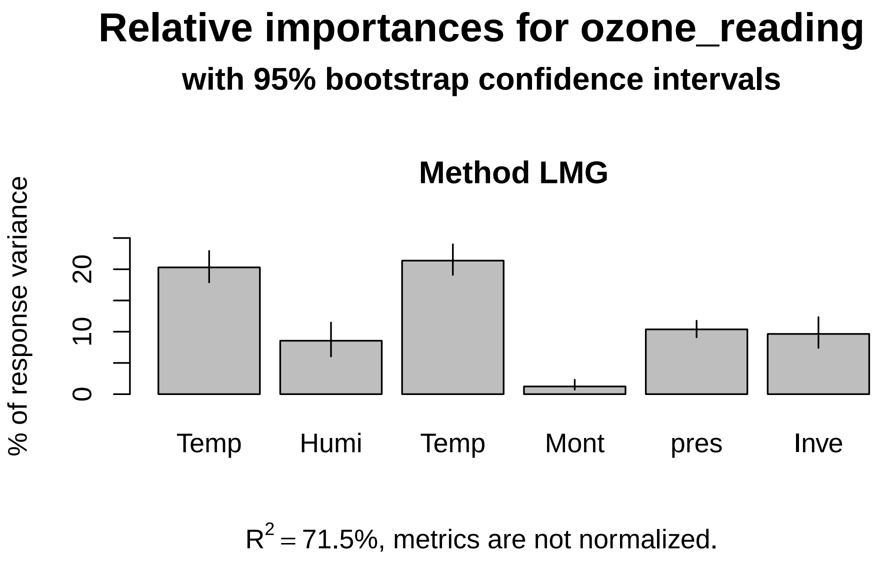
It is implemented in the `relaimpo` package. Basically, you build a linear regression model and pass that as the main argument to `calc.relimp()`. relaimpo has multiple options to compute the relative importance, but the recommended method is to use `type='lmg'`, as I have done below.

```
# install.packages('relaimpo')
library(relaimpo)
#> Loading required package: boot
#>
#> Attaching package: 'boot'
#> The following object is masked from 'package:lattice':
#>
#>     melanoma
#> The following object is masked from 'package:survival':
#>
#>     aml
#> Loading required package: survey
#> Loading required package: grid
#>
#> Attaching package: 'survey'
#> The following object is masked from 'package:graphics':
```

```
#>
#>     dotchart
#> Loading required package: mitools
#> This is the global version of package relaimpo.
#> If you are a non-US user, a version with the interesting additional metric pmvd is available
#> from Ulrike Groempings web site at prof.beuth-hochschule.de/groemping.

# Build linear regression model
model_formula = ozone_reading ~ Temperature_Sandburg + Humidity + Temperature_ElMonte + Month + pressure
lmMod <- lm(model_formula, data=trainData)

# calculate relative importance
relImportance <- calc.relimp(lmMod, type = "lmg", rela = F)

# Sort
cat('Relative Importances: \n')
#> Relative Importances:
sort(round(relImportance$lmg, 3), decreasing=TRUE)
#>   Temperature_ElMonte  Temperature_Sandburg       pressure_height
#>                 0.214                 0.203                 0.104
#> Inversion_base_height              Humidity                 Month
#>                 0.096                 0.086                 0.012
```

Additionally, you can use bootstrapping (using `boot.relimp`) to compute the confidence intervals of the produced relative importances.

```
bootsub <- boot.relimp(ozone_reading ~ Temperature_Sandburg + Humidity + Temperature_ElMonte + Month + p
                       b = 1000, type = 'lmg', rank = TRUE, diff = TRUE)

plot(booteval.relimp(bootsub, level=.95))
```



3.8 Recursive Feature Elimination (RFE)

Recursive feature elimnation (rfe) offers a rigorous way to determine the important variables before you even feed them into a ML algo.

It can be implemented using the `rfe()` from caret package.

The rfe() also takes two important parameters.

- `sizes`
- `rfeControl`

So, what does `sizes` and `rfeControl` represent?

The sizes determines the number of most important features the rfe should iterate. Below, I have set the size as 1 to 5, 10, 15 and 18.

Secondly, the `rfeControl` parameter receives the output of the `rfeControl()`. You can set what type of variable evaluation algorithm must be used. Here, I have used random forests based rfFuncs. The `method='repeatedCV'` means it will do a repeated k-Fold cross validation with `repeats=5`.

Once complete, you get the accuracy and kappa for each model size you provided. The final selected model subset size is marked with a * in the rightmost selected column.

```
str(trainData)
#> 'data.frame':    366 obs. of  13 variables:
#>  $ Month                 : int  1 1 1 1 1 1 1 1 1 1 ...
#>  $ Day_of_month          : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ Day_of_week           : int  4 5 6 7 1 2 3 4 5 6 ...
#>  $ ozone_reading         : num  3 3 3 5 5 6 4 4 6 7 ...
#>  $ pressure_height       : num  5480 5660 5710 5700 5760 5720 5790 5790 5700 5700 ...
#>  $ Wind_speed            : int  8 6 4 3 3 4 6 3 3 3 ...
#>  $ Humidity              : num  20 41 28 37 51 ...
#>  $ Temperature_Sandburg : num  40.5 38 40 45 54 ...
#>  $ Temperature_ElMonte  : num  39.8 46.7 49.5 52.3 45.3 ...
#>  $ Inversion_base_height: num  5000 4109 2693 590 1450 ...
#>  $ Pressure_gradient    : num  -15 -14 -25 -24 25 15 -33 -28 23 -2 ...
#>  $ Inversion_temperature: num  30.6 48 47.7 55 57 ...
#>  $ Visibility           : int  200 300 250 100 60 60 100 250 120 120 ...
```

```
tic()
set.seed(100)
options(warn=-1)

subsets <- c(1:5, 10, 15, 18)

ctrl <- rfeControl(functions = rfFuncs,
                   method = "repeatedcv",
                   repeats = 5,
                   verbose = FALSE)

lmProfile <- rfe(x=trainData[, c(1:3, 5:13)], y=trainData$ozone_reading,
                 sizes = subsets,
                 rfeControl = ctrl)
toc()
#> 93.627 sec elapsed
lmProfile
#>
#> Recursive feature selection
#>
#> Outer resampling method: Cross-Validated (10 fold, repeated 5 times)
#>
#> Resampling performance over subset size:
```

```
#>
#>  Variables RMSE Rsquared  MAE RMSESD RsquaredSD MAESD Selected
#>          1 5.13    0.595 3.92  0.826     0.1275 0.586
#>          2 4.03    0.746 3.11  0.542     0.0743 0.416
#>          3 3.95    0.756 3.06  0.472     0.0670 0.380
#>          4 3.93    0.759 3.01  0.468     0.0683 0.361
#>          5 3.90    0.763 2.98  0.467     0.0659 0.350
#>         10 3.77    0.782 2.85  0.496     0.0734 0.393        *
#>         12 3.77    0.781 2.86  0.508     0.0756 0.401
#>
#> The top 5 variables (out of 10):
#>    Temperature_ElMonte, Pressure_gradient, Temperature_Sandburg, Inversion_temperature, Humidity
```

So, it says, Temperature_ElMonte, Pressure_gradient, Temperature_Sandburg, Inversion_temperature, Humidity are the top 5 variables in that order.

And the best model size out of the provided models sizes (in subsets) is 10.

You can see all of the top 10 variables from '`lmProfile$optVariables`' that was created using `rfe` function above.

3.9 Genetic Algorithm

You can perform a supervised feature selection with genetic algorithms using the `gafs()`. This is **quite resource expensive** so consider that before choosing the number of iterations (iters) and the number of repeats in `gafsControl()`.

```
tic()
# Define control function
ga_ctrl <- gafsControl(functions = rfGA,  # another option is `caretGA`.
                        method = "cv",
                        repeats = 3)

# Genetic Algorithm feature selection
set.seed(100)
ga_obj <- gafs(x=trainData[, c(1:3, 5:13)],
               y=trainData[, 4],
               iters = 3,   # normally much higher (100+)
               gafsControl = ga_ctrl)
toc()
#> 652.647 sec elapsed
ga_obj
#>
#> Genetic Algorithm Feature Selection
#>
#> 366 samples
#> 12 predictors
#>
#> Maximum generations: 3
#> Population per generation: 50
#> Crossover probability: 0.8
#> Mutation probability: 0.1
#> Elitism: 0
#>
```

```
#> Internal performance values: RMSE, Rsquared
#> Subset selection driven to minimize internal RMSE
#>
#> External performance values: RMSE, Rsquared, MAE
#> Best iteration chose by minimizing external RMSE
#> External resampling method: Cross-Validated (10 fold)
#>
#> During resampling:
#>   * the top 5 selected variables (out of a possible 12):
#>     Month (100%), Pressure_gradient (100%), Temperature_ElMonte (100%), Humidity (80%), Visibility (
#>   * on average, 6.8 variables were selected (min = 5, max = 9)
#>
#> In the final search using the entire training set:
#>    * 9 features selected at iteration 2 including:
#>      Month, Day_of_month, pressure_height, Wind_speed, Humidity ...
#>    * external performance at this iteration is
#>
#>       RMSE   Rsquared        MAE
#>      3.721      0.788      2.800
```

```
# Optimal variables
ga_obj$optVariables
#> [1] "Month"                 "Day_of_month"          "pressure_height"
#> [4] "Wind_speed"            "Humidity"              "Temperature_ElMonte"
#> [7] "Inversion_base_height" "Pressure_gradient"     "Inversion_temperature"
```

'Month' 'Day_of_month' 'Wind_speed' 'Temperature_ElMonte' 'Pressure_gradient' 'Visibility'

So the optimal variables according to the genetic algorithms are listed above. But, I wouldn't use it just yet because, the above variant was tuned for only 3 iterations, which is quite low. I had to set it so low to save computing time.

3.10 Simulated Annealing

Simulated annealing is a global search algorithm that allows a suboptimal solution to be accepted in hope that a better solution will show up eventually.

It works by making small random changes to an initial solution and sees if the performance improved. The change is accepted if it improves, else it can still be accepted if the difference of performances meet an acceptance criteria.

In caret it has been implemented in the `safs()` which accepts a control parameter that can be set using the `safsControl()` function.

`safsControl` is similar to other control functions in caret (like you saw in rfe and ga), and additionally it accepts an improve parameter which is the number of iterations it should wait without improvement until the values are reset to previous iteration.

```
tic()
# Define control function
sa_ctrl <- safsControl(functions = rfSA,
                        method = "repeatedcv",
                        repeats = 3,
                        improve = 5) # n iterations without improvement before a reset
```

```r
# Genetic Algorithm feature selection
set.seed(100)
sa_obj <- safs(x=trainData[, c(1:3, 5:13)],
               y=trainData[, 4],
               safsControl = sa_ctrl)
toc()
#> 111.982 sec elapsed
sa_obj
#>
#> Simulated Annealing Feature Selection
#>
#> 366 samples
#> 12 predictors
#>
#> Maximum search iterations: 10
#> Restart after 5 iterations without improvement (0.3 restarts on average)
#>
#> Internal performance values: RMSE, Rsquared
#> Subset selection driven to minimize internal RMSE
#>
#> External performance values: RMSE, Rsquared, MAE
#> Best iteration chose by minimizing external RMSE
#> External resampling method: Cross-Validated (10 fold, repeated 3 times)
#>
#> During resampling:
#>   * the top 5 selected variables (out of a possible 12):
#>     Temperature_Sandburg (80%), Month (66.7%), Pressure_gradient (66.7%), Temperature_ElMonte (63.3%
#>   * on average, 6.5 variables were selected (min = 3, max = 11)
#>
#> In the final search using the entire training set:
#>    * 6 features selected at iteration 9 including:
#>      Day_of_week, pressure_height, Wind_speed, Humidity, Inversion_base_height ...
#>    * external performance at this iteration is
#>
#>       RMSE   Rsquared        MAE
#>      4.108      0.743      3.111
```

```r
# Optimal variables
print(sa_obj$optVariables)
#> [1] "Day_of_week"           "pressure_height"       "Wind_speed"
#> [4] "Humidity"              "Inversion_base_height" "Pressure_gradient"
```

3.11 Information Value and Weights of Evidence

The Information Value can be used to judge how important a given categorical variable is in explaining the binary Y variable. It goes well with logistic regression and other classification models that can model binary variables.

Let's try to find out how important the categorical variables are in predicting if an individual will earn > 50k from the `adult.csv` dataset. Just run the code below to import the dataset.

```r
library(InformationValue)
#>
```

```
#> Attaching package: 'InformationValue'
#> The following objects are masked from 'package:caret':
#>
#>     confusionMatrix, precision, sensitivity, specificity

# online data
# inputData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/adult.csv")

inputData <- read.csv(file.path(data_raw_dir, "adult.csv"))
print(head(inputData))
#>   AGE        WORKCLASS FNLWGT EDUCATION EDUCATIONNUM      MARITALSTATUS
#> 1  39        State-gov  77516 Bachelors           13      Never-married
#> 2  50 Self-emp-not-inc  83311 Bachelors           13 Married-civ-spouse
#> 3  38          Private 215646   HS-grad            9           Divorced
#> 4  53          Private 234721      11th            7 Married-civ-spouse
#> 5  28          Private 338409 Bachelors           13 Married-civ-spouse
#> 6  37          Private 284582   Masters           14 Married-civ-spouse
#>          OCCUPATION  RELATIONSHIP  RACE    SEX CAPITALGAIN CAPITALLOSS
#> 1      Adm-clerical Not-in-family White   Male        2174           0
#> 2   Exec-managerial       Husband White   Male           0           0
#> 3 Handlers-cleaners Not-in-family White   Male           0           0
#> 4 Handlers-cleaners       Husband Black   Male           0           0
#> 5    Prof-specialty          Wife Black Female           0           0
#> 6   Exec-managerial          Wife White Female           0           0
#>   HOURSPERWEEK NATIVECOUNTRY ABOVE50K
#> 1           40 United-States        0
#> 2           13 United-States        0
#> 3           40 United-States        0
#> 4           40 United-States        0
#> 5           40          Cuba        0
#> 6           40 United-States        0
```

```
# Choose Categorical Variables to compute Info Value.
cat_vars <- c ("WORKCLASS", "EDUCATION", "MARITALSTATUS", "OCCUPATION", "RELATIONSHIP", "RACE", "SEX",

factor_vars <- cat_vars


# Init Output
df_iv <- data.frame(VARS=cat_vars, IV=numeric(length(cat_vars)), STRENGTH=character(length(cat_vars)),

# Get Information Value for each variable
for (factor_var in factor_vars){
  df_iv[df_iv$VARS == factor_var, "IV"] <- InformationValue::IV(X=inputData[, factor_var], Y=inputData$
  df_iv[df_iv$VARS == factor_var, "STRENGTH"] <- attr(InformationValue::IV(X=inputData[, factor_var], Y
}

# Sort
df_iv <- df_iv[order(-df_iv$IV), ]

df_iv
#>            VARS     IV          STRENGTH
#> 5  RELATIONSHIP 1.5356   Highly Predictive
#> 3 MARITALSTATUS 1.3388   Highly Predictive
#> 4    OCCUPATION 0.7762   Highly Predictive
#> 2     EDUCATION 0.7411   Highly Predictive
#> 7           SEX 0.3033   Highly Predictive
#> 1     WORKCLASS 0.1634   Highly Predictive
#> 8 NATIVECOUNTRY 0.0794 Somewhat Predictive
#> 6          RACE 0.0693 Somewhat Predictive
```

Here is what the quantum of Information Value means:

Less than 0.02, then the predictor is not useful for modeling (separating the Goods from the Bads)

0.02 to 0.1, then the predictor has only a weak relationship. 0.1 to 0.3, then the predictor has a medium strength relationship. 0.3 or higher, then the predictor has a strong relationship. That was about IV. Then what is Weight of Evidence?

Weights of evidence can be useful to find out how important a given categorical variable is in explaining the 'events' (called 'Goods' in below table.)

The 'Information Value' of the categorical variable can then be derived from the respective WOE values.

IV=(perc good of all goods−perc bad of all bads) *WOE

The 'WOETable' below given the computation in more detail.

```
WOETable(X=inputData[, 'WORKCLASS'], Y=inputData$ABOVE50K)
#>                  CAT GOODS  BADS TOTAL   PCT_G    PCT_B    WOE       IV
#> 1                  ?   191  1645  1836 0.02429 0.066545 -1.008 0.042574
#> 2        Federal-gov   371   589   960 0.04719 0.023827  0.683 0.015964
#> 3          Local-gov   617  1476  2093 0.07848 0.059709  0.273 0.005131
#> 4       Never-worked     7     7     7 0.00089 0.000283  1.146 0.000696
#> 5            Private  4963 17733 22696 0.63126 0.717354 -0.128 0.011006
#> 6       Self-emp-inc   622   494  1116 0.07911 0.019984  1.376 0.081363
#> 7 Self-emp-not-inc   724  1817  2541 0.09209 0.073503  0.225 0.004190
#> 8          State-gov   353   945  1298 0.04490 0.038228  0.161 0.001073
#> 9        Without-pay    14    14    14 0.00178 0.000566  1.146 0.001391
```

The total IV of a variable is the sum of IV's of its categories.

3.12 DALEX Package

The `DALEX` is a powerful package that explains various things about the variables used in an ML model.

For example, using the `variable_dropout()` function you can find out how important a variable is based on a dropout loss, that is how much loss is incurred by removing a variable from the model.

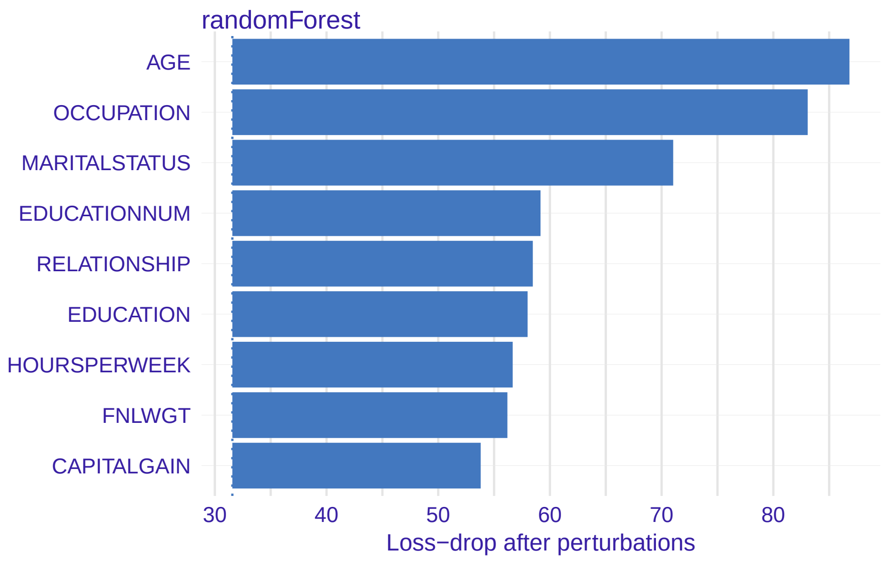
Apart from this, it also has the `single_variable()` function that gives you an idea of how the model's output will change by changing the values of one of the X's in the model.

It also has the `single_prediction()` that can decompose a single model prediction so as to understand which variable caused what effect in predicting the value of Y.

```
library(randomForest)
#> randomForest 4.6-14
#> Type rfNews() to see new features/changes/bug fixes.
#>
#> Attaching package: 'randomForest'
#> The following object is masked from 'package:dplyr':
#>
```

```
#>     combine
#> The following object is masked from 'package:ranger':
#> 
#>     importance
#> The following object is masked from 'package:ggplot2':
#> 
#>     margin
library(DALEX)
#> Welcome to DALEX (version: 0.3.0).
#> This is a plain DALEX. Use 'install_dependencies()' to get all required packages.
#> 
#> Attaching package: 'DALEX'
#> The following object is masked from 'package:dplyr':
#> 
#>     explain

# Load data
# inputData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/adult.csv")

inputData <- read.csv(file.path(data_raw_dir, "adult.csv"))

# Train random forest model
rf_mod <- randomForest(factor(ABOVE50K) ~ ., data=inputData, ntree=100)
rf_mod
#> 
#> Call:
#>  randomForest(formula = factor(ABOVE50K) ~ ., data = inputData,      ntree = 100)
#>                Type of random forest: classification
#>                      Number of trees: 100
#> No. of variables tried at each split: 3
#> 
#>         OOB estimate of  error rate: 13.6%
#> Confusion matrix:
#>       0    1 class.error
#> 0 23051 1669      0.0675
#> 1  2754 5087      0.3512

# Variable importance with DALEX
explained_rf <- explain(rf_mod, data=inputData, y=inputData$ABOVE50K)

# Get the variable importances
varimps = variable_dropout(explained_rf, type='raw')

print(varimps)
#>         variable dropout_loss        label
#> 1   _full_model_         31.6 randomForest
#> 2       ABOVE50K         31.6 randomForest
#> 3           RACE         36.6 randomForest
#> 4            SEX         39.4 randomForest
#> 5    CAPITALLOSS         39.9 randomForest
#> 6  NATIVECOUNTRY         40.3 randomForest
#> 7      WORKCLASS         51.0 randomForest
#> 8    CAPITALGAIN         53.8 randomForest
#> 9          FNLWGT          56.2 randomForest
#> 10  HOURSPERWEEK          56.7 randomForest
#> 11      EDUCATION          58.0 randomForest
#> 12   RELATIONSHIP          58.5 randomForest
#> 13   EDUCATIONNUM          59.2 randomForest
#> 14 MARITALSTATUS          71.0 randomForest
#> 15     OCCUPATION          83.1 randomForest
#> 16            AGE          86.8 randomForest
#> 17     _baseline_         304.4 randomForest
```

```
plot(varimps)
```



3.13 Conclusion

Hope you find these methods useful. As it turns out different methods showed different variables as important, or at least the degree of importance changed. This need not be a conflict, because each method gives a different perspective of how the variable can be useful depending on how the algorithms learn `Y ~ x`. So its cool.

If you find any code breaks or bugs, report the issue here or just write it below.

Chapter 4

Imputting missing values with Random Forest

4.1 Flu Prediction. `fluH7N9_china_2013` dataset

Source: https://shirinsplayground.netlify.com/2018/04/flu_prediction/

```
library(outbreaks)
library(tidyverse)
library(plyr)
library(mice)
library(caret)
library(purrr)
library("tibble")
library("dplyr")
library("tidyr")
```

Since I migrated my blog from Github Pages to blogdown and Netlify, I wanted to start migrating (most of) my old posts too - and use that opportunity to update them and make sure the code still works.

Here I am updating my very first machine learning post from 27 Nov 2016: Can we predict flu deaths with Machine Learning and R?. Changes are marked as bold comments.

The main changes I made are:

- using the tidyverse more consistently throughout the analysis
- focusing on comparing multiple imputations from the mice package, rather than comparing different algorithms
- using purrr, map(), nest() and unnest() to model and predict the machine learning algorithm over the different imputed datasets

---

Among the many nice R packages containing data collections is the `outbreaks` package. It contains a dataset on epidemics and among them is data from the 2013 outbreak of influenza A H7N9 in China as analysed by Kucharski et al. (2014):

> A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Distinguishing between reservoir exposure and human-to-human transmission for emerging

pathogens using case onset data. PLOS Currents Outbreaks. Mar 7, edition 1. doi: 10.1371/currents.outbreaks.e1473d9bfc99d080ca242139a06c455f.

A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Data from: Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. Dryad Digital Repository. http://dx.doi.org/10.5061/dryad.2g43n.

I will be using their data as an example to show how to use Machine Learning algorithms for predicting disease outcome.

4.2 The data

The dataset contains case ID, date of onset, date of hospitalization, date of outcome, gender, age, province and of course outcome: Death or Recovery.

4.2.1 Pre-processing

Change: variable names (i.e. column names) have been renamed, dots have been replaced with underscores, letters are all lower case now.

Change: I am using the tidyverse notation more consistently.

First, I'm doing some preprocessing, including:

- renaming missing data as NA
- adding an ID column
- setting column types
- gathering date columns
- changing factor names of dates (to make them look nicer in plots) and of province (to combine provinces with few cases)

```
from1 <- c("date_of_onset", "date_of_hospitalisation", "date_of_outcome")
to1   <- c("date of onset", "date of hospitalisation", "date of outcome")
from2 <- c("Anhui", "Beijing", "Fujian", "Guangdong", "Hebei", "Henan",
           "Hunan", "Jiangxi", "Shandong", "Taiwan")
to2   <- rep("Other", 10)

fluH7N9_china_2013$age[which(fluH7N9_china_2013$age == "?")] <- NA
fluH7N9_china_2013_gather <- fluH7N9_china_2013 %>%
  mutate(case_id = paste("case", case_id, sep = "_"),
         age = as.numeric(age)) %>%
  gather(Group, Date, date_of_onset:date_of_outcome) %>%
  mutate(Group = as.factor(mapvalues(Group, from = from1, to = to1)),
         province = mapvalues(province, from = from2, to = to2))

fluH7N9_china_2013 <- as.tibble(fluH7N9_china_2013)
#> Warning: `as.tibble()` is deprecated, use `as_tibble()` (but mind the new semantics).
#> This warning is displayed once per session.
fluH7N9_china_2013_gather <- as.tibble(fluH7N9_china_2013_gather)
print(fluH7N9_china_2013)
#> # A tibble: 136 x 8
#>   case_id date_of_onset date_of_hospita~ date_of_outcome outcome gender
#>   <fct>   <date>        <date>           <date>          <fct>   <fct>
#> 1 1       2013-02-19    NA               2013-03-04      Death   m
#> 2 2       2013-02-27    2013-03-03       2013-03-10      Death   m
#> 3 3       2013-03-09    2013-03-19       2013-04-09      Death   f
#> 4 4       2013-03-19    2013-03-27       NA              <NA>    f
#> 5 5       2013-03-19    2013-03-30       2013-05-15      Recover f
#> 6 6       2013-03-21    2013-03-28       2013-04-26      Death   f
#> # ... with 130 more rows, and 2 more variables: age <fct>, province <fct>
```

I'm also adding a third gender level for unknown gender

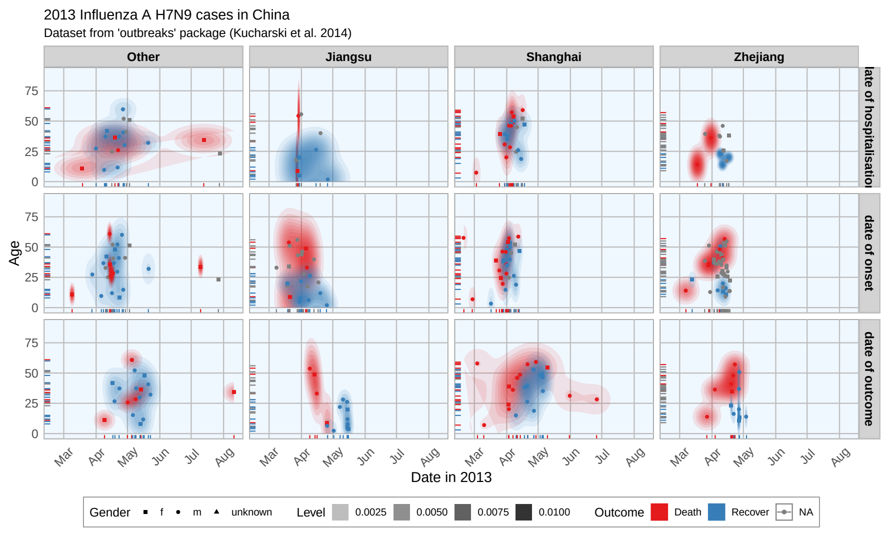
```
levels(fluH7N9_china_2013_gather$gender) <-
  c(levels(fluH7N9_china_2013_gather$gender), "unknown")
fluH7N9_china_2013_gather$gender[is.na(fluH7N9_china_2013_gather$gender)] <- "unknown"
print(fluH7N9_china_2013_gather)
#> # A tibble: 408 x 7
#>   case_id outcome gender   age province Group         Date
#>   <chr>   <fct>   <fct>  <dbl> <fct>    <fct>         <date>
#> 1 case_1  Death   m         58 Shanghai date of onset 2013-02-19
#> 2 case_2  Death   m          7 Shanghai date of onset 2013-02-27
#> 3 case_3  Death   f         11 Other    date of onset 2013-03-09
#> 4 case_4  <NA>    f         18 Jiangsu  date of onset 2013-03-19
#> 5 case_5  Recover f         20 Jiangsu  date of onset 2013-03-19
#> 6 case_6  Death   f          9 Jiangsu  date of onset 2013-03-21
#> # ... with 402 more rows
```

For plotting, I am defining a custom ggplot2 theme:

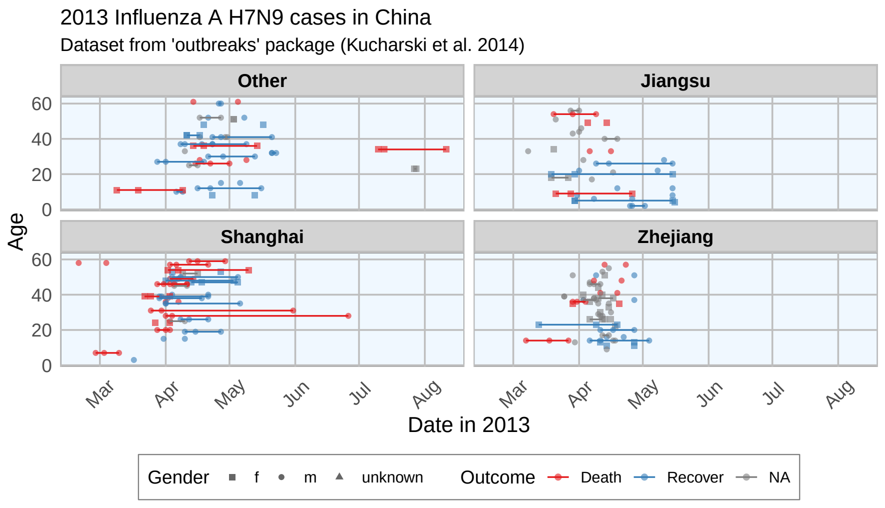
```
my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.text.x = element_text(angle = 45, vjust = 0.5, hjust = 0.5),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "lightgrey", color = "grey", size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "black"),
    legend.position = "bottom",
    legend.justification = "top",
    legend.box = "horizontal",
    legend.box.background = element_rect(colour = "grey50"),
    legend.background = element_blank(),
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}
```

And use that theme to visualize the data:

```
ggplot(data = fluH7N9_china_2013_gather, aes(x = Date, y = age, fill = outcome)) +
  stat_density2d(aes(alpha = ..level..), geom = "polygon") +
  geom_jitter(aes(color = outcome, shape = gender), size = 1.5) +
  geom_rug(aes(color = outcome)) +
  scale_y_continuous(limits = c(0, 90)) +
  labs(
    fill = "Outcome",
    color = "Outcome",
    alpha = "Level",
    shape = "Gender",
    x = "Date in 2013",
    y = "Age",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
    caption = ""
  ) +
  facet_grid(Group ~ province) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1")
#> Warning: Removed 149 rows containing non-finite values (stat_density2d).
#> Warning: Removed 149 rows containing missing values (geom_point).
```



```
ggplot(data = fluH7N9_china_2013_gather, aes(x = Date, y = age, color = outcome)) +
  geom_point(aes(color = outcome, shape = gender), size = 1.5, alpha = 0.6) +
  geom_path(aes(group = case_id)) +
  facet_wrap( ~ province, ncol = 2) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1") +
  labs(
    color = "Outcome",
    shape = "Gender",
    x = "Date in 2013",
    y = "Age",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
    caption = "\nTime from onset of flu to outcome."
  )
#> Warning: Removed 149 rows containing missing values (geom_point).
#> Warning: Removed 122 rows containing missing values (geom_path).
```



4.3 Features

In machine learning-speak features are what we call the variables used for model training. Using the right features dramatically influences the accuracy and success of your model.

For this example, I am keeping age, but I am also generating new features from the date information and converting gender and province into numerical values.

```
delta_dates <- function(onset, ref) {
    d2 = as.Date(as.character(onset), format = "%Y-%m-%d")
    d1 = as.Date(as.character(ref), format = "%Y-%m-%d")
    as.numeric(as.character(gsub(" days", "", d1 - d2)))
}
```

```
dataset <- fluH7N9_china_2013 %>%
  mutate(
      hospital = as.factor(ifelse(is.na(date_of_hospitalisation), 0, 1)),
      gender_f = as.factor(ifelse(gender == "f", 1, 0)),
      province_Jiangsu = as.factor(ifelse(province == "Jiangsu", 1, 0)),
      province_Shanghai = as.factor(ifelse(province == "Shanghai", 1, 0)),
      province_Zhejiang = as.factor(ifelse(province == "Zhejiang", 1, 0)),
      province_other = as.factor(ifelse(province == "Zhejiang"
                                        | province == "Jiangsu"
                                        | province == "Shanghai", 0, 1)),

      days_onset_to_outcome = delta_dates(date_of_onset, date_of_outcome),
      days_onset_to_hospital = delta_dates(date_of_onset, date_of_hospitalisation),
      age = age,
      early_onset = as.factor(ifelse(date_of_onset <
                                      summary(date_of_onset)[[3]], 1, 0)),
      early_outcome = as.factor(ifelse(date_of_outcome <
                                        summary(date_of_outcome)[[3]], 1, 0))
    ) %>%
  subset(select = -c(2:4, 6, 8))

rownames(dataset) <- dataset$case_id
```

```
#> Warning: Setting row names on a tibble is deprecated.
dataset[, -2] <- as.numeric(as.matrix(dataset[, -2]))
print(dataset)
#> # A tibble: 136 x 13
#>   case_id outcome   age hospital gender_f province_Jiangsu province_Shangh~
#> *   <dbl> <fct>   <dbl>    <dbl>    <dbl>            <dbl>            <dbl>
#> 1       1 Death      87        0        0                0                1
#> 2       2 Death      27        1        0                0                1
#> 3       3 Death      35        1        1                0                0
#> 4       4 <NA>       45        1        1                1                0
#> 5       5 Recover    48        1        1                1                0
#> 6       6 Death      32        1        1                1                0
#> # ... with 130 more rows, and 6 more variables: province_Zhejiang <dbl>,
#> #   province_other <dbl>, days_onset_to_outcome <dbl>,
#> #   days_onset_to_hospital <dbl>, early_onset <dbl>, early_outcome <dbl>
```

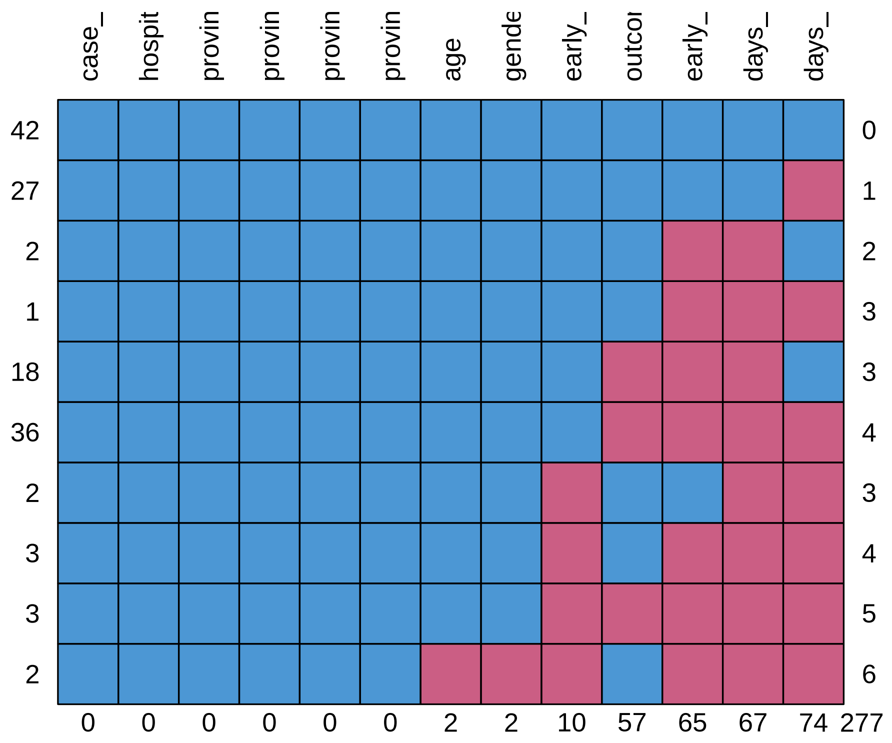
```
summary(dataset$outcome)
#>   Death Recover    NA's
#>      32      47      57
```

4.4 Imputing missing values

I am using the `mice` package for imputing missing values

Note: Since publishing this blogpost I learned that the idea behind using mice is to compare different imputations to see how stable they are, instead of picking one imputed set as fixed for the remainder of the analysis. Therefore, I changed the focus of this post a little bit: in the old post I compared many different algorithms and their outcome; in this updated version I am only showing the **Random Forest** algorithm and focus on **comparing the different imputed datasets**. I am ignoring feature importance and feature plots because nothing changed compared to the old post.

```
# plot the missing data in a matrix by variables
md_pattern <- md.pattern(dataset, rotate.names = TRUE)
```



```
dataset_impute <- mice(data = dataset[, -2],  print = FALSE)
#> Warning: Number of logged events: 150
```

4.4.1 Generate a dataframe of five imputting strategies

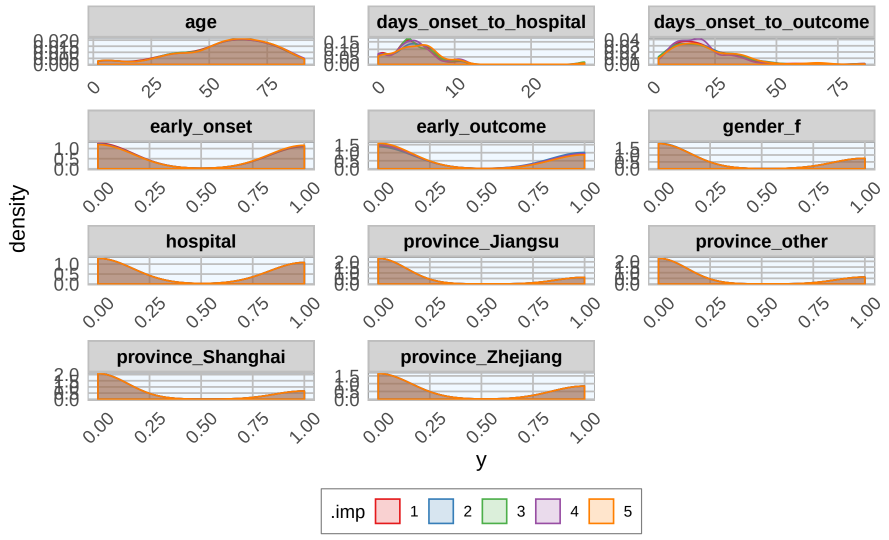
- by default, mice() calculates five (m = 5) imputed data sets
- we can combine them all in one output with the complete("long") function
- I did not want to impute missing values in the outcome column, so I have to merge it back in with the imputed data

```
# c(1,2): case_id, outcome
datasets_complete <- right_join(dataset[, c(1, 2)],
                           complete(dataset_impute, "long"),
                           by = "case_id") %>%
  mutate(.imp = as.factor(.imp)) %>%
  select(-.id) %>%
  print()
#> # A tibble: 680 x 14
#>   case_id outcome .imp    age hospital gender_f province_Jiangsu
#>     <dbl> <fct>   <fct> <dbl>    <dbl>    <dbl>            <dbl>
#> 1       1 Death   1        87        0        0                0
#> 2       2 Death   1        27        1        0                0
#> 3       3 Death   1        35        1        1                0
#> 4       4 <NA>    1        45        1        1                1
#> 5       5 Recover 1        48        1        1                1
#> 6       6 Death   1        32        1        1                1
#> # ... with 674 more rows, and 7 more variables: province_Shanghai <dbl>,
#> #   province_Zhejiang <dbl>, province_other <dbl>,
#> #   days_onset_to_outcome <dbl>, days_onset_to_hospital <dbl>,
#> #   early_onset <dbl>, early_outcome <dbl>
```

Let's compare the distributions of the five different imputed datasets:

4.4.2 plot effect of imputting on features

```
datasets_complete %>%
  gather(x, y, age:early_outcome) %>%
  ggplot(aes(x = y, fill = .imp, color = .imp)) +
    geom_density(alpha = 0.20) +
  facet_wrap(~ x, ncol = 3, scales = "free") +
    scale_fill_brewer(palette="Set1", na.value = "grey50") +
    scale_color_brewer(palette="Set1", na.value = "grey50") +
    my_theme()
```



4.5 Test, train and validation data sets

Now, we can go ahead with machine learning!

The dataset contains a few missing values in the outcome column; those will be the test set used for final predictions (see the old blog post for this).

```
length(which(is.na(datasets_complete$outcome)))
length(which(!is.na(datasets_complete$outcome)))
#> [1] 285
#> [1] 395
```

```
train_index <- which(is.na(datasets_complete$outcome))
train_data <- datasets_complete[-train_index, ]
test_data  <- datasets_complete[train_index, -2]       # remove variable outcome
```

```
print(train_data)
#> # A tibble: 395 x 14
#>   case_id outcome .imp    age hospital gender_f province_Jiangsu
#>     <dbl> <fct>   <fct> <dbl>    <dbl>    <dbl>            <dbl>
#> 1       1 Death   1        87        0        0                0
#> 2       2 Death   1        27        1        0                0
#> 3       3 Death   1        35        1        1                0
#> 4       5 Recover 1        48        1        1                1
#> 5       6 Death   1        32        1        1                1
#> 6       7 Death   1        83        1        0                1
#> # ... with 389 more rows, and 7 more variables: province_Shanghai <dbl>,
#> #   province_Zhejiang <dbl>, province_other <dbl>,
#> #   days_onset_to_outcome <dbl>, days_onset_to_hospital <dbl>,
#> #   early_onset <dbl>, early_outcome <dbl>
```

```
# outcome variable removed
print(test_data)
#> # A tibble: 285 x 13
#>   case_id .imp    age hospital gender_f province_Jiangsu province_Shangh~
#>     <dbl> <fct> <dbl>    <dbl>    <dbl>            <dbl>            <dbl>
#> 1       4 1        45        1        1                1                0
#> 2       9 1        67        1        0                0                0
#> 3      15 1        61        0        1                1                0
#> 4      16 1        79        0        0                1                0
#> 5      22 1        85        1        0                1                0
#> 6      28 1        79        0        0                0                0
#> # ... with 279 more rows, and 6 more variables: province_Zhejiang <dbl>,
#> #   province_other <dbl>, days_onset_to_outcome <dbl>,
#> #   days_onset_to_hospital <dbl>, early_onset <dbl>, early_outcome <dbl>
```

The remainder of the data will be used for modeling. Here, I am splitting the data into 70% training and 30% test data.

Because I want to model each imputed dataset separately, I am using the `nest()` and `map()` functions.

```
train_data_nest <- train_data %>%
  group_by(.imp) %>%
  nest() %>%
  print()
#> # A tibble: 5 x 2
#>   .imp  data
#>   <fct> <list>
#> 1 1     <tibble [79 x 13]>
#> 2 2     <tibble [79 x 13]>
#> 3 3     <tibble [79 x 13]>
#> 4 4     <tibble [79 x 13]>
#> 5 5     <tibble [79 x 13]>
```

```
# split the training data in validation training and validation test
set.seed(42)
val_data <- train_data_nest %>%
  mutate(val_index = map(data, ~ createDataPartition(.$outcome,
                                                     p = 0.7,
                                                  list = FALSE)),
         val_train_data = map2(data, val_index, ~ .x[.y, ]),
         val_test_data  = map2(data, val_index, ~ .x[-.y, ])) %>%
  print()
#> # A tibble: 5 x 5
#>   .imp  data               val_index        val_train_data     val_test_data
#>   <fct> <list>             <list>           <list>             <list>
#> 1 1     <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
#> 2 2     <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
#> 3 3     <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
#> 4 4     <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
#> 5 5     <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
```

4.6 Machine Learning algorithms

4.6.1 Random Forest

To make the code tidier, I am first defining the modeling function with the parameters I want.

```
model_function <- function(df) {
  caret::train(outcome ~ .,
               data = df,
               method = "rf",
               preProcess = c("scale", "center"),
               trControl = trainControl(method = "repeatedcv",
                                        number = 5,
                                        repeats = 3,
                                        verboseIter = FALSE))
}
```

4.6.2 Add model and prediction to nested dataframe and calculate

Next, I am using the nested tibble from before to map() the model function, predict the outcome and calculate confusion matrices.

4.6.2.1 add model list-column

```
val_data_model <- val_data %>%
  mutate(model = map(val_train_data, ~ model_function(.x))) %>%
  select(-val_index) %>%
  print()
#> # A tibble: 5 x 5
#>   .imp  data              val_train_data    val_test_data     model
#>   <fct> <list>            <list>            <list>            <list>
#> 1 1     <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
#> 2 2     <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
#> 3 3     <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
#> 4 4     <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
#> 5 5     <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
```

4.6.2.2 add prediction and confusion matrix list-columns

```
set.seed(42)
val_data_model <- val_data_model %>%
  mutate(
        predict = map2(model, val_test_data, ~
                               data.frame(prediction = predict(.x, .y[, -2]))),
         predict_prob = map2(model, val_test_data, ~
                               data.frame(outcome = .y[, 2],
                               prediction = predict(.x, .y[, -2], type = "prob"))),
         confusion_matrix = map2(val_test_data, predict, ~
                                        confusionMatrix(.x$outcome, .y$prediction)),
         confusion_matrix_tbl = map(confusion_matrix, ~ as.tibble(.x$table))) %>%
  print()
#> # A tibble: 5 x 9
#>    .imp  data  val_train_data val_test_data model predict predict_prob
#>    <fct> <lis> <list>         <list>        <lis> <list>  <list>
#> 1 1     <tib~ <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 2 2     <tib~ <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 3 3     <tib~ <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 4 4     <tib~ <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 5 5     <tib~ <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> # ... with 2 more variables: confusion_matrix <list>,
#> #   confusion_matrix_tbl <list>
```

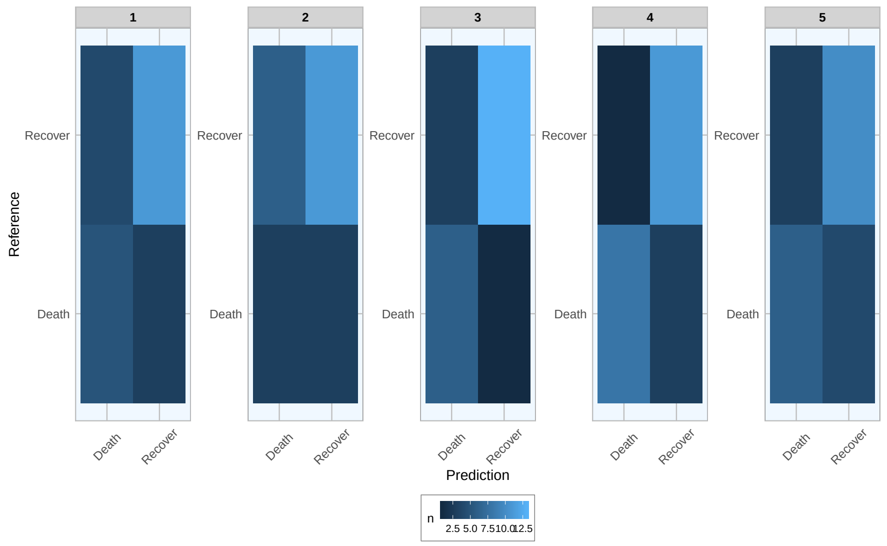
Finally, we have a nested dataframe of 5 rows or cases, one per imputting strategy with its corresponding models and prediction results.

4.7 Comparing accuracy of models

To compare how the different imputations did, I am plotting the confusion matrices:

```
val_data_model_unnest <- val_data_model %>%
  unnest(confusion_matrix_tbl) %>%
  print()
#> # A tibble: 20 x 4
#>    .imp  Prediction Reference     n
#>    <fct> <chr>      <chr>     <int>
#> 1 1     Death      Death         5
#> 2 1     Recover    Death         3
#> 3 1     Death      Recover       4
#> 4 1     Recover    Recover      11
#> 5 2     Death      Death         3
#> 6 2     Recover    Death         3
#> # ... with 14 more rows
```

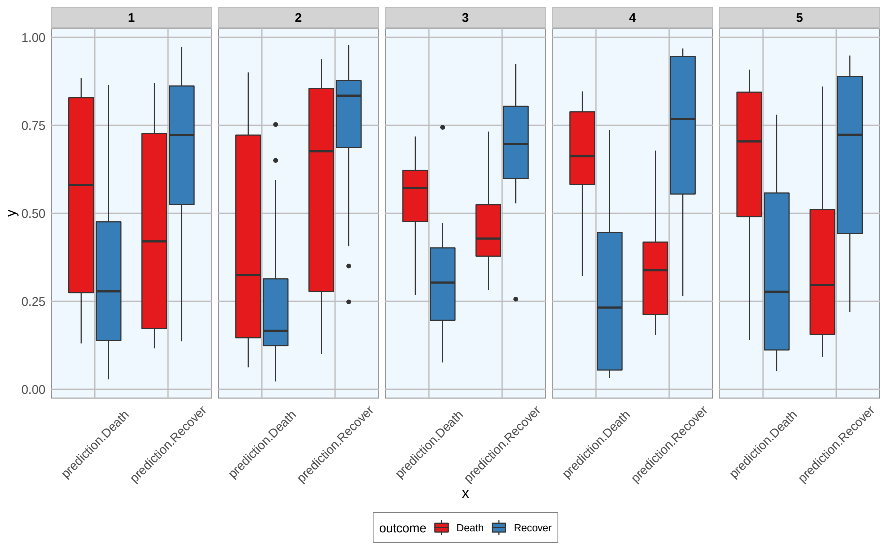
```
val_data_model_unnest %>%
  ggplot(aes(x = Prediction, y = Reference, fill = n)) +
    facet_wrap(~ .imp, ncol = 5, scales = "free") +
    geom_tile() +
    my_theme()
```



and the prediction probabilities for correct and wrong predictions:

```
val_data_model_gather <- val_data_model %>%
  unnest(predict_prob) %>%
  gather(x, y, prediction.Death:prediction.Recover) %>%
  print()
#> # A tibble: 230 x 4
#>   .imp  outcome x                       y
#>   <fct> <fct>   <chr>               <dbl>
#> 1 1     Death   prediction.Death 0.758
#> 2 1     Recover prediction.Death 0.864
#> 3 1     Death   prediction.Death 0.828
#> 4 1     Death   prediction.Death 0.828
#> 5 1     Recover prediction.Death 0.342
#> 6 1     Recover prediction.Death 0.552
#> # ... with 224 more rows
```

```
val_data_model_gather %>%
  ggplot(aes(x = x, y = y, fill = outcome)) +
    facet_wrap(~ .imp, ncol = 5) +
    geom_boxplot() +
    scale_fill_brewer(palette="Set1", na.value = "grey50") +
    my_theme()
```



Hope, you found that example interesting and helpful!

```
sessionInfo()
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> other attached packages:
#>  [1] caret_6.0-84    mice_3.4.0      lattice_0.20-38 plyr_1.8.4
#>  [5] forcats_0.4.0   stringr_1.4.0   dplyr_0.8.0.1   purrr_0.3.2
#>  [9] readr_1.3.1     tidyr_0.8.3     tibble_2.1.1    ggplot2_3.1.1
#> [13] tidyverse_1.2.1 outbreaks_1.5.0 logging_0.9-107
#>
#> loaded via a namespace (and not attached):
#>  [1] nlme_3.1-139        lubridate_1.7.4     RColorBrewer_1.1-2
#>  [4] httr_1.4.0          rprojroot_1.3-2     tools_3.6.0
#>  [7] backports_1.1.4     utf8_1.1.4          R6_2.4.0
#> [10] rpart_4.1-15        lazyeval_0.2.2      colorspace_1.4-1
#> [13] jomo_2.6-7          nnet_7.3-12         withr_2.1.2
#> [16] tidyselect_0.2.5    compiler_3.6.0      cli_1.1.0
#> [19] rvest_0.3.3         xml2_1.2.0          labeling_0.3
#> [22] bookdown_0.10       scales_1.0.0        randomForest_4.6-14
#> [25] digest_0.6.18       minqa_1.2.4         rmarkdown_1.12
#> [28] pkgconfig_2.0.2     htmltools_0.3.6     lme4_1.1-21
#> [31] rlang_0.3.4         readxl_1.3.1        rstudioapi_0.10
#> [34] generics_0.0.2      jsonlite_1.6        ModelMetrics_1.2.2
#> [37] magrittr_1.5        Matrix_1.2-17       Rcpp_1.0.1
#> [40] munsell_0.5.0       fansi_0.4.0         stringi_1.4.3
#> [43] yaml_2.2.0          MASS_7.3-51.4       recipes_0.1.5
#> [46] grid_3.6.0          parallel_3.6.0      mitml_0.3-7
#> [49] crayon_1.3.4        haven_2.1.0         splines_3.6.0
#> [52] hms_0.4.2           zeallot_0.1.0       knitr_1.22
#> [55] pillar_1.4.0        boot_1.3-22         reshape2_1.4.3
#> [58] codetools_0.2-16    stats4_3.6.0        pan_1.6
#> [61] glue_1.3.1          evaluate_0.13       data.table_1.12.2
#> [64] modelr_0.1.4        vctrs_0.1.0         nloptr_1.2.1
#> [67] foreach_1.4.4       cellranger_1.1.0    gtable_0.3.0
#> [70] assertthat_0.2.1    xfun_0.6            gower_0.2.0
#> [73] prodlim_2018.04.18  broom_0.5.2         e1071_1.7-1
#> [76] class_7.3-15        survival_2.44-1.1   timeDate_3043.102
#> [79] iterators_1.0.10    lava_1.6.5          ipred_0.9-9
```