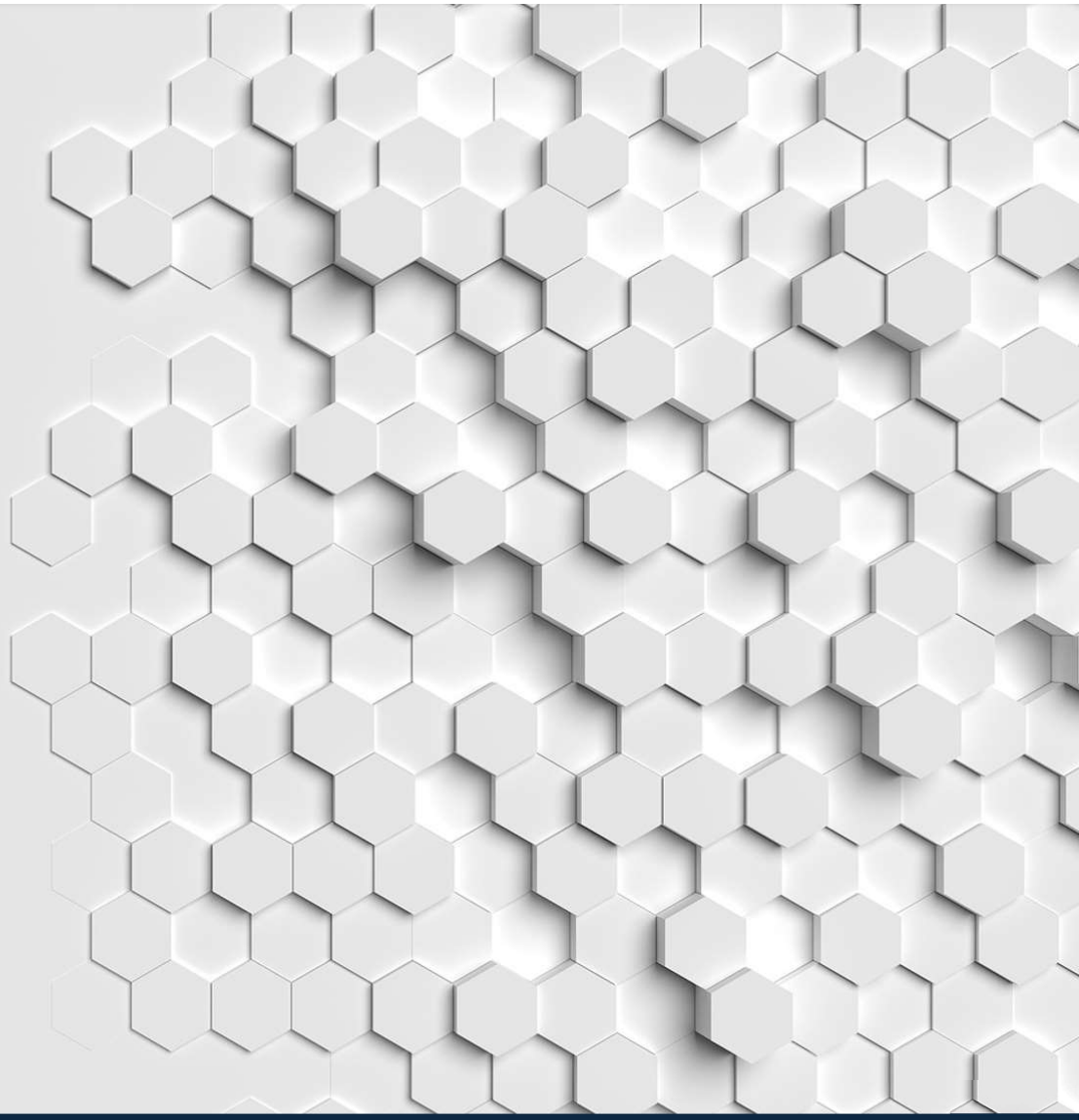
Statistical Computing

Regression Tree & KNN| August , 2022

Emmanuelle Rodrigues Nunes

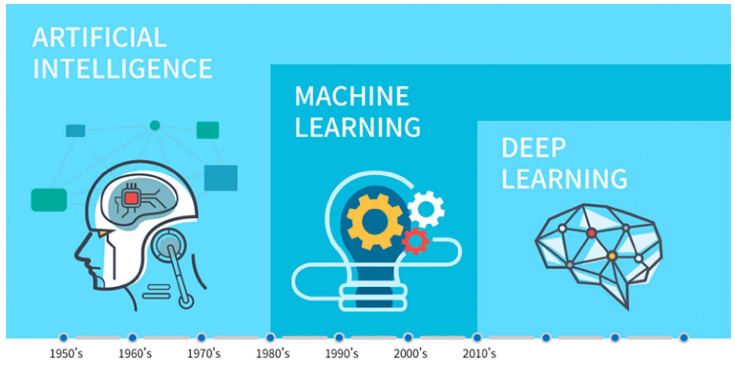


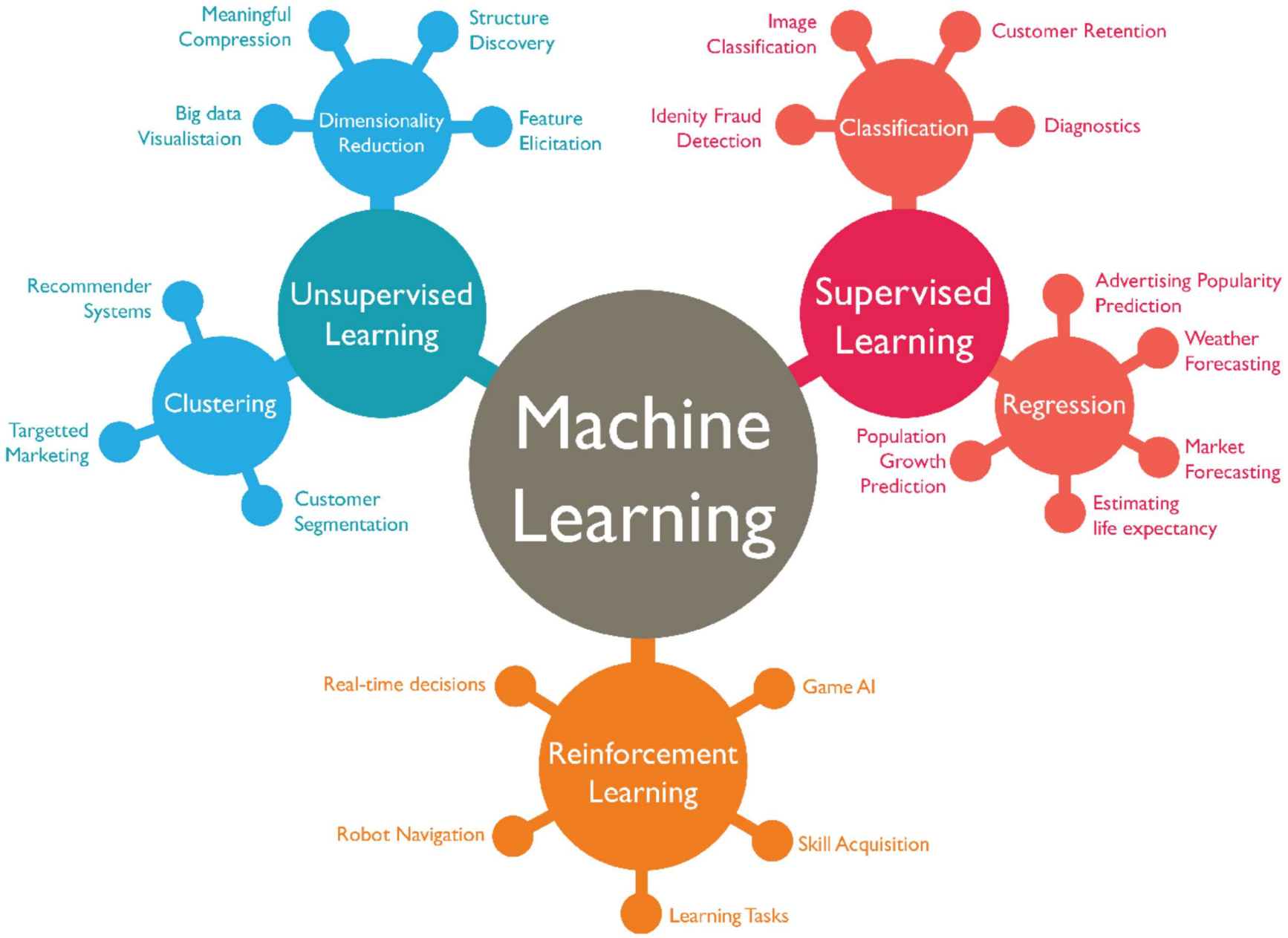
CIDRA-AIPD
African Institute for Professional Development

Machine Learning

Machine Learning - Concepts

- Machine Learning (ML) is a subset of Artificial Intelligence (AI)
- Algorithms that can improve automatically through experience and by the use of data without being explicit programmed, reason why we say that the algorithms learn.
- With ML algorithms we can build a model to make predictions or decisions.
- Machine learning algorithms are used many different applications, for example:
  - Medicine
  - Email filtering
  - Speech recognition
  - Computer vision




Machine Learning
Unsupervised Learning
Dimensionality Reduction
Meaningful Compression
Structure Discovery
Big data Visualistaion
Feature Elicitation
Clustering
Recommender Systems
Targetted Marketing
Customer Segmentation
Supervised Learning
Classification
Image Classification
Customer Retention
Idenity Fraud Detection
Diagnostics
Regression
Advertising Popularity Prediction
Weather Forecasting
Population Growth Prediction
Market Forecasting
Estimating life expectancy
Reinforcement Learning
Real-time decisions
Game AI
Robot Navigation
Skill Acquisition
Learning Tasks

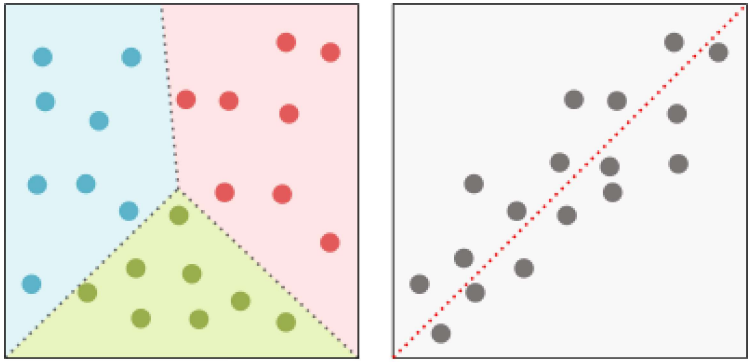
Supervised Learning

Supervised Learning

Supervised learning is where you have input variables (X) and an output variable (y) and you use an algorithm to learn the mapping function from the input to the output.

$$y = f(X)$$

- It is the most common type of Machine Learning problem
- It is called **supervised** because we have the label that tell us the correct information, and we are going to be corrected if we predict wrong.
- Supervised learning can be grouped into two problems:
  - **Regression:** The output variable is a real number, for example, weight
  - **Classification:** The output variable is a category, for example, disease and no disease



Classification

Classification

Classification is a type of **supervised** learning where we categorise data into classes. There are many different algorithms that can help us solve this kind of problems.

Classification requires a training dataset with many examples of inputs and outputs from which to learn. It can be categorised in two types of problems:

- **Binary classification:** The outcome has only **two** labels, for example, disease and not disease.
  - Some popular algorithms are: Logistic Regression, Decision Tree, K-Nearest Neighbour (KNN)
- **Multi-label classification:** The outcome has multiple labels, for example, dog, cat, bird and other.
  - Some popular algorithms: KNN, Decision Tree, Random Forest, Naive Bayes

To evaluate the model performance we can make use of ROC, confusion matrix, etc. We need to be aware of **class imbalance** problems.

Decision Tree (CART)

Decision Tree

There are various algorithms that can grow a tree.

- Differences:
  - Possible structure of the tree (e.g. number of splits per node)
  - Criteria how to find the splits
  - Criteria to stop splitting
  - How to estimate the simple models within the leaf nodes.

The **Classification and Regression Trees (CART)** algorithm is probably the most popular algorithm for tree induction.

- We will focus on CART, but the interpretation is similar for most other tree types.

> Note: Decision Trees can be used for both Regression and Classification problems

Theory

The processes behind *classification* and *regression* in tree analysis is very similar, but we need to first distinguish the two.

- **Classification:** For a response variable which has *classes*, we want to organize the dataset into groups by the response variable.
- **Regression:** When our response variable is instead numeric or continuous we wish to use the data to predict the outcome, and will use regression trees in this situation.

Essentially, a classification tree splits the data based on homogeneity by categorizing the data based on similarity, filtering out the "noise" and making the data "pure", hence the concept of **purity criterion**.

When the response variable does not have classes, a regression model is fit to each of the independent variables, isolating these variables as nodes where their inclusion decreases error.

<!--

--> <!-- -->

Theory

CART takes a feature and determines which cut-off point minimizes:

- The variance of Y for a regression task
  - The variance tells us how much the y values in a node are spread around their mean value
- The Gini index of the class distribution of Y for classification tasks
  - The Gini index tells us how "impure" a node is, e.g. if all classes have the same frequency, the node is impure, if only one class is present, it is maximally pure.

Variance and Gini index are minimized when the data points in the nodes have very similar values for Y. As a consequence, the best cut-off point makes the two resulting subsets as different as possible with respect to the target outcome.

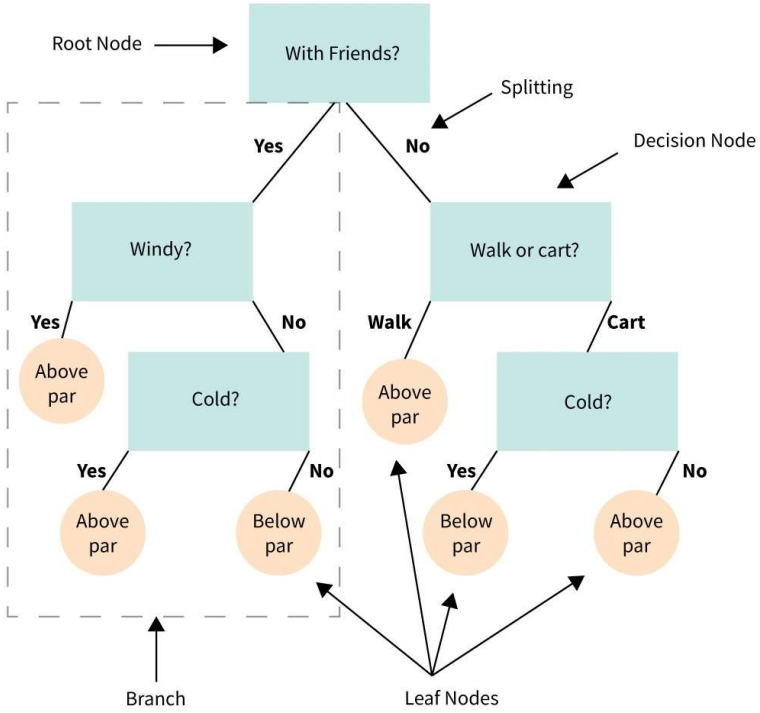
For categorical features, the algorithm tries to create subsets by trying different groupings of categories. After the best cutoff per feature has been determined, the algorithm selects the feature for splitting that would result in the best partition in terms of the variance or Gini index and adds this split to the tree. The algorithm continues this search-and-split recursively in both new nodes until a stop criterion is reached. Possible criteria are: A minimum number of instances that have to be in a node before the split, or the minimum number of instances that have to be in a terminal node.

Interpretation

Let's first define some keys terms:

- **Root node:** The base of the decision tree.
- **Splitting:** The process of dividing a node into multiple sub-nodes.
- **Decision node:** When a sub-node is further split into additional sub-nodes.
- **Leaf node:** When a sub-node does not further split into additional sub-nodes; represents possible outcomes.
- **Pruning:** The process of removing sub-nodes of a decision tree.
- **Branch:** A subsection of the decision tree consisting of multiple nodes.

Starting from the root node, you go to the next nodes and the edges tell you which subsets you are looking at. Once you reach the leaf node, the node tells you the predicted outcome. All the edges are connected by 'AND', so we are conditioning on the variable.



For example, if you are with friends *AND* it is windy, *THEN* it is above par.

In R

There are two main libraries that implement the CART model:

- tree
  - Command: *tree(formula, data, weights, subset, na.action = na.pass, control = tree.control(nobs, ...), method = "recursive.partition", split = c("deviance", "gini"), model = FALSE, x = FALSE, y = TRUE, wts = TRUE, ...)*
    - **formula** defines if it is classification or regression. It should be entered on the format $Y \sim x_1 + \ldots + x_n$
    - **data**: Dataset
- rpart
  - Command: *rpart(formula, data, weights, subset, na.action = na.rpart, method, model = FALSE, x = FALSE, y = TRUE, parms, control, cost, ...)*
    - **method** defines if it is a classification or regression

In summary, if using tree for:

- **Classification**: $\text{tree}(\text{factor}(Y) \sim \sum x, \text{data})$
- **Regression**: $\text{tree}(Y \sim \sum x, \text{data})$

If using rpart:

- **Classification**: $\text{tree}(Y \sim \sum x, \text{data}, \text{method} = ' \text{class}')$
- **Regression**: $\text{tree}(Y \sim \sum x, \text{data}, \text{method} = ' \text{anova}')$

Example

We will use a **classification** dataset for the example. The dataset will be the `Pima Indians Diabetes Database` from `mlbench`.

```
library(rsample)     # data splitting
library(dplyr)       # data wrangling
library(rpart)       # performing regression trees
library(rpart.plot)  # plotting regression trees
library(mlbench)     # ML datasets

# Pima Indians Diabetes Database
data(PimaIndiansDiabetes)

glimpse(PimaIndiansDiabetes)
```

```
## Rows: 768
## Columns: 9
## $ pregnant <dbl> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, 10, 10, 1, 5, 7, 0, 7, 1, 1…
## $ glucose  <dbl> 148, 85, 183, 89, 137, 116, 78, 115, 197, 125, 110, 168, 139,…
## $ pressure <dbl> 72, 66, 64, 66, 40, 74, 50, 0, 70, 96, 92, 74, 80, 60, 72, 0,…
## $ triceps  <dbl> 35, 29, 0, 23, 35, 0, 32, 0, 45, 0, 0, 0, 0, 23, 19, 0, 47, 0…
## $ insulin  <dbl> 0, 0, 0, 94, 168, 0, 88, 0, 543, 0, 0, 0, 0, 846, 175, 0, 230…
## $ mass     <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, 25.6, 31.0, 35.3, 30.5, 0.0, 37…
## $ pedigree <dbl> 0.627, 0.351, 0.672, 0.167, 2.288, 0.201, 0.248, 0.134, 0.158…
## $ age      <dbl> 50, 31, 32, 21, 33, 30, 26, 29, 53, 54, 30, 34, 57, 59, 51, 3…
## $ diabetes <fct> pos, neg, pos, neg, pos, neg, pos, neg, pos, pos, neg, pos, n…
```

Example

First, we will split the data into a training and test set

```
# Create training (70%) and test (30%) sets
# Use set.seed for reproducibility
set.seed(123)
pima_split ← initial_split(PimaIndiansDiabetes, prop = .7)
pima_train ← training(pima_split)
pima_test  ← testing(pima_split)
```

Example - Implementation

I'll fit the example using `rpart` and plot the tree using `rpart.plot`

```
tree ← rpart(formula = diabetes ~ .,
             data = pima_train,
             method = 'class')

tree
```

```
## n= 537
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##    1) root 537 187 neg (0.65176909 0.34823091)
##      2) glucose< 143.5 414  96 neg (0.76811594 0.23188406)
##        4) age< 28.5 226  28 neg (0.87610619 0.12389381)
##          8) mass< 29.95 107   2 neg (0.98130841 0.01869159) *
##          9) mass≥29.95 119  26 neg (0.78151261 0.21848739)
##           18) pressure≥51 107  18 neg (0.83177570 0.16822430) *
##           19) pressure< 51 12   4 pos (0.33333333 0.66666667) *
##        5) age≥28.5 188  68 neg (0.63829787 0.36170213)
##         10) mass< 27.05 43   4 neg (0.90697674 0.09302326) *
##         11) mass≥27.05 145  64 neg (0.55862069 0.44137931)
##           22) glucose< 108.5 58  15 neg (0.74137931 0.25862069) *
##           23) glucose≥108.5 87  38 pos (0.43678161 0.56321839)
##             46) pedigree< 0.7205 67  32 neg (0.52238806 0.47761194)
```

Explaining the output

The output explains the steps for the split. For example,

- We start with 537 observations at the root node (very beginning)
- The first variable we split on (the first variable that optimizes a reduction in the Gini index) is `glucose`.
- We can see that at the first node all observations with `glucose < 143.5` go to the 2nd (`2`)) branch.
- The total number of observations that follow this branch (`414`), their probability of not having diabetes is `0.768` and we have `96` negatives.
    - If you look for the 3rd branch (`3`)) you will see that `123` observations with `glucose ⩾ 143.5` follow this branch and their probability of not having diabetes is 0.26.
- Basically, this is telling us the most important variable that has the largest reduction in Gini initially is `glucose` with the majority of people having lower glucose values resulting in not having diabetes.

Example - Visualise Tree

We can visualize our model with `rpart.plot` that has many plotting options. One thing you may notice is that this tree contains 13 internal nodes resulting in 14 terminal nodes.

```
rpart.plot(tree)
```

Explaining the Partitions

You might have noticed that not all the variables were used to generate the plot.

Behind the scenes `rpart` is automatically applying a range of cost complexity to prune the tree. To compare the error for each value, rpart performs a 10-fold cross validation so that the error associated with a given value is computed on the hold-out validation data.

In this example we find diminishing returns after 14 terminal nodes as illustrated below. We could use a tree with 3 terminal nodes and reasonably expect to experience similar results within a small margin of error.

```
plotcp(tree)
```

Pruning

By default, `rpart` is performing some automated tuning, with an optimal subtree of 13 splits, 14 terminal nodes, and a cross-validated error of 0.7433. However, we can perform additional tuning to try improve model performance.

```
tree$cptable
```

```
##           CP nsplit rel error    xerror       xstd
## 1 0.31550802      0 1.0000000 1.0000000 0.05903724
## 2 0.02139037      1 0.6844920 0.6844920 0.05280048
## 3 0.01960784      2 0.6631016 0.7326203 0.05402086
## 4 0.01782531      7 0.5614973 0.7326203 0.05402086
## 5 0.01604278     10 0.5080214 0.7058824 0.05335630
## 6 0.01069519     11 0.4919786 0.7165775 0.05362609
## 7 0.01000000     13 0.4705882 0.7433155 0.05427754
```

Tuning

In addition to the cost complexity (α parameter), it is also common to tune:

- **minsplit:** the minimum number of data points required to attempt a split before it is forced to create a terminal node.
- **maxdepth:** the maximum number of internal nodes between the root node and the terminal nodes.

```
tree_control ← rpart(
    formula = diabetes ~ .,
    data    = pima_train,
    method  = "class",
    control = list(minsplit = 10, maxdepth = 12, xval = 10)
)

tree_control$cptable
```

```
##           CP nsplit rel error    xerror       xstd
## 1 0.31550802      0 1.0000000 1.0000000 0.05903724
## 2 0.02139037      1 0.6844920 0.7005348 0.05321939
## 3 0.01960784      2 0.6631016 0.7272727 0.05389058
## 4 0.01782531      7 0.5614973 0.7272727 0.05389058
## 5 0.01604278     10 0.5080214 0.7112299 0.05349187
## 6 0.01069519     11 0.4919786 0.7005348 0.05321939
## 7 0.01000000     13 0.4705882 0.7165775 0.05362609
```

Tuning

This approach requires you to manually assess multiple models.

We can perform a grid search to automatically search across a range of differently tuned models to identify the optimal hyerparameter setting.

```
hyper_grid ← expand.grid(
  minsplit = seq(5, 20, 1),
  maxdepth = seq(8, 15, 1)
)

# total number of combinations
nrow(hyper_grid)
```

```
## [1] 128
```

```
models ← list()

for (i in 1:nrow(hyper_grid)) {
  # train a model and store in the list
  models[[i]] ← rpart(
    formula = diabetes ~ .,
    data    = pima_train,
    method  = "class",
    control = list(minsplit = hyper_grid$minsplit[i], maxdepth = hyper_grid$maxdepth[i])
    )
}
```

Tuning

We can now create a function to extract the minimum error associated with the optimal cost complexity value for each model.

```
# function to get optimal cp
get_cp ← function(x) {
  min    ← which.min(x$cptable[, "xerror"])
  cp ← x$cptable[min, "CP"]
}

# function to get minimum error
get_min_error ← function(x) {
  min    ← which.min(x$cptable[, "xerror"])
  xerror ← x$cptable[min, "xerror"]
}

hyper_grid %>%
  mutate(
    cp    = purrr::map_dbl(models, get_cp),
    error = purrr::map_dbl(models, get_min_error)
    ) %>%
  arrange(error) %>%
  top_n(-5, wt = error)
```

```
##   minsplit maxdepth         cp     error
## 1        8       13 0.01069519 0.6470588
## 2       11       10 0.01604278 0.6577540
```

Example - Applying best model and predict

We can apply this final optimal model and predict on our test set.

```
optimal_tree ← rpart(
    formula = diabetes ~ .,
    data    = pima_train,
    method  = "class",
    control = list(minsplit = 8, maxdepth = 13, cp = 0.01069519)
    )

pred ← predict(optimal_tree, newdata = pima_test) %>% as.data.frame() %>%
  mutate(class = as.factor(ifelse(neg > pos, "neg", "pos")))
performanceEstimation::classificationMetrics(pima_test$diabetes, pred$class,
                                             metrics = c("rec", "prec", "F", "acc"))
```

```
##       rec      prec         F       acc
## 0.8733333 0.7359551 0.7987805 0.7142857
```

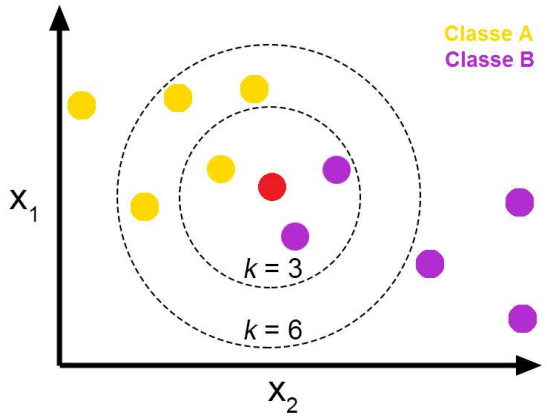
KNN

KNN

The K-Nearest Neighbours (KNN) is a **supervised learning** algorithm that can be used for regression and classification tasks, but it is more comonly used for classification.

- K represents the number of neighbours that will be used
- It assumes that similar things will be close to each other and dissimilar things will be far from each other.
- To understand the similarity between observations, we need to know the distance between them.
    - There are different distance metrics that we can use depending on the problem we are solving.

When usign KNN we need to normalise our data, as variables we different scale might have a bigger impact simply because of the scale.



The KNN Algorithm

1. Load the data
2. Initialise K 2.a) K is a user defined choice
3. For each observation in the data 3.a) Calculate the distance between the current observation and all the data points in the dataset 3.b) Sort the distances from smallest to largest 3.c) Pick the first K entries from the sorted collection
4. If regression, return the mean of the K labels and if classification return the most common label

Choosing the right value for K

If we pick the wrong K for the data, our prediction is going to be wrong. We need to run the algorithm for different K values to find the K that reduces the errors we encounter whilst maintaining a good performance for unseen data.

Example - KNN

We will continue to use the same diabetes dataset.

We will be using the library `recipes` to create workflows to apply the normalisation step with the use of the functions `step_scale` and `step_center` and we will build the model using the `nearest_neighbor` function from parsnip with the engine `kknn`.

```
library(recipes)
library(workflows)

diabetes_recipe ← recipes::recipe(diabetes ~ ., data = PimaIndiansDiabetes) ▷
  recipes::step_scale(recipes::all_predictors()) ▷
  recipes::step_center(recipes::all_predictors())

diabetes_spec ← parsnip::nearest_neighbor(neighbors = parsnip::tune()) ▷
  parsnip::set_engine("kknn") ▷
  parsnip::set_mode("classification")
```

We can notice that for the number K of neighbours we passed the function `tune`, meaning that we are going to tune the value of K.

Example - Train KNN

To train the model we will use 5 fold cross validation.

```
diabetes_cv ← rsample::vfold_cv(pima_train, v = 5)

diabetes_wkflw ← workflows::workflow() ▷
  workflows::add_recipe(diabetes_recipe) ▷
  workflows::add_model(diabetes_spec)

diabetes_wkflw
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## ── Preprocessor ────────────────────────────────────────────────────────────
## 2 Recipe Steps
##
## • step_scale()
## • step_center()
##
## ── Model ───────────────────────────────────────────────────────────────────
## K-Nearest Neighbor Model Specification (classification)
##
## Main Arguments:
##   neighbors = parsnip::tune()
##
## Computational engine: kknn
```

Example - Train KNN

Next, we run cross validation for a grid of neighbours ranging from 1 to 20.

```
grid_vals ← tibble::tibble(neighbors = seq(from = 1, to = 20, by = 1))

diabetes_results ← diabetes_wkflw ▷
  tune::tune_grid(resamples = diabetes_cv, grid = grid_vals)

diabetes_results ▷
  tune::collect_metrics()
```

```
## # A tibble: 40 × 7
##    neighbors .metric  .estimator  mean     n std_err .config
##        <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##  1         1 accuracy binary     0.698     5 0.00808 Preprocessor1_Model01
##  2         1 roc_auc  binary     0.664     5 0.00772 Preprocessor1_Model01
##  3         2 accuracy binary     0.698     5 0.00808 Preprocessor1_Model02
##  4         2 roc_auc  binary     0.724     5 0.0193  Preprocessor1_Model02
##  5         3 accuracy binary     0.698     5 0.00808 Preprocessor1_Model03
##  6         3 roc_auc  binary     0.745     5 0.0227  Preprocessor1_Model03
##  7         4 accuracy binary     0.698     5 0.00808 Preprocessor1_Model04
##  8         4 roc_auc  binary     0.758     5 0.0204  Preprocessor1_Model04
##  9         5 accuracy binary     0.721     5 0.00663 Preprocessor1_Model05
## 10         5 roc_auc  binary     0.764     5 0.0197  Preprocessor1_Model05
## # … with 30 more rows
```

KNN - Run final model with tunned parameters

We can finish the workflow by using the function `finalize_workflow` alongside the function `select_best` that selects the best parameter combination from the grid search.

```
(diabetes_wkflw ← diabetes_wkflw ▷
  tune::finalize_workflow(diabetes_results ▷
                            tune::select_best(metric = "roc_auc")))
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## ── Preprocessor ────────────────────────────────────────────────────────────
## 2 Recipe Steps
##
## • step_scale()
## • step_center()
##
## ── Model ───────────────────────────────────────────────────────────────────
## K-Nearest Neighbor Model Specification (classification)
##
## Main Arguments:
##   neighbors = 20
##
```

KNN - Evaluate the model on the test set

So far we have:

- Defined our recipe
- Defined the model
- Tuned the model's parameters

Now, we're ready to fit the final model. All the steps are contained within the `workflow` object, so we can apply the `last_fit()` function to our workflow and our train/test split object. This will automatically train the model specified by the workflow using the training data, and produce evaluations based on the test set.

```
diabetes_fit ← diabetes_wkflw ▷
  # fit on the training set and evaluate on test set
  tune::last_fit(pima_split)

diabetes_fit
```

```
## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits            id               .metrics .notes   .predictions .workflow
##   <list>            <chr>            <list>   <list>   <list>       <list>
## 1 <split [537/231]> train/test split <tibble> <tibble> <tibble>     <workflow>
```

KNN - Seeing the result of our model

To see the performance of the test set, we can use again the function `collect_metrics()`

```
diabetes_fit ▷
  tune::collect_metrics()
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.779 Preprocessor1_Model1
## 2 roc_auc  binary         0.825 Preprocessor1_Model1
```

References

Majid (2013)

Breiman et al. (1984)