# Model and Feature Selection

## CITS4009 Computational Data Analysis

Unit Coordinator: Dr Du Huynh

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2022

## Model Selection

## Overfitting

An **overfit model** looks great on the training data and then performs poorly on new data.

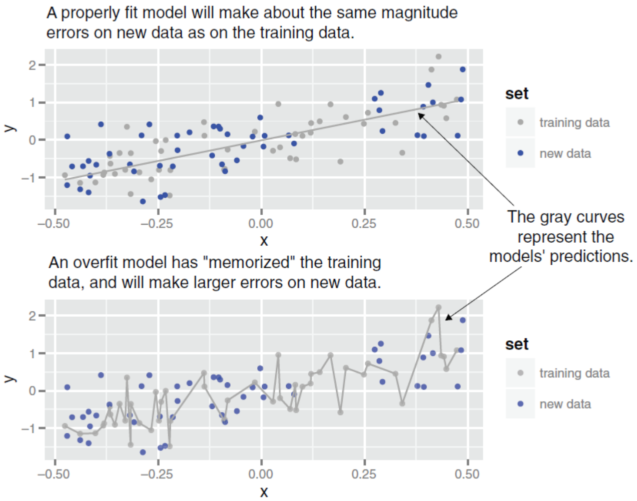
- *Training Error*: A model's prediction error for the data that it is trained on.
- *Generalisation error*: A model's prediction error for new data.

Usually, the training error will be smaller than the generalisation error (no big surprise). Ideally, though, the two errors should be close to each other.

If the generalisation error is large and your model's test performance is poor while your training error is small, then your model has probably overfit the training data.

- An overfitting model has memorised the training data instead of discovering generalisable rules or patterns.
- Simpler models are preferred as they tend to generalise better and avoid overfitting.

## Overfitting - Illustration



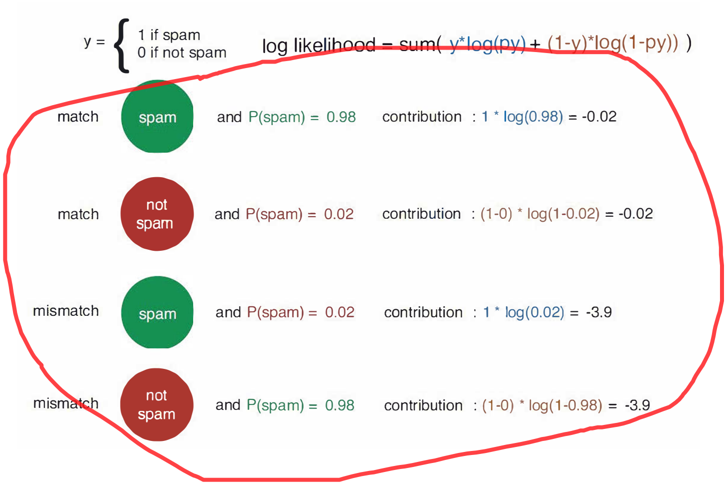
## Evaluating Probability Models

## Log Likelihood

**Log likelihood** is a measure (a non-positive number) of how well a model's predictions "match" the true class labels.

- A log likelihood of 0 means a perfect match: e.g., the model scores all the spam-emails as *spam* with a probability of 1, and all the nonspam-emails as having a probability 0 of being *spam*.
- The larger the magnitude of the log likelihood, the worse the match. So we prefer a larger log likelihood (as it is non-positive) that is close to 0.
- The log likelihood of a model's prediction on a specific instance is the logarithm of the probability that the model assigns to the instance's actual class. 模型分配给实例的实际类的概率的对数

## Log Likelihood - Illustration



y = { 1 if spam; 0 if not spam }

log likelihood = sum( y*log(py) + (1-y)*log(1-py)) )

| | | | |
|---|---|---|---|
| match | spam | and P(spam) = 0.98 | contribution : 1 * log(0.98) = -0.02 |
| match | not spam | and P(spam) = 0.02 | contribution : (1-0) * log(1-0.02) = -0.02 |
| mismatch | spam | and P(spam) = 0.02 | contribution : 1 * log(0.02) = -3.9 |
| mismatch | not spam | and P(spam) = 0.98 | contribution : (1-0) * log(1-0.98) = -3.9 |

## Deviance

Another common measure when fitting probability models is the *deviance*.

The deviance is defined as $-2 \times (\text{logLikelihood} - S)$, where $S$ is a technical constant called "the log likelihood of the *saturated model*."

In most cases, the saturated model is a perfect model that returns probability 1 for items in the class and probability 0 for items not in the class (so $S = 0$).

**The lower the deviance, the better the model**.

## Akaike Information Criterion (AIC)

*AIC* is an important *variant* of *deviance*. AIC是deviance的重要变体

AIC is defined as

$$\boxed{\textit{deviance} + 2 \times \text{numberOfParameters}}$$

The more parameters are in the model, the more complex the model is; the more complex a model is, the more likely it is to overfit.

Thus, AIC is deviance penalised for model complexity.

When comparing models (on the same test set), you will generally prefer the model with a smaller AIC.

The AIC is useful for comparing models with different measures of complexity and modelling variables with differing numbers of levels.

## Scoring models using AIC

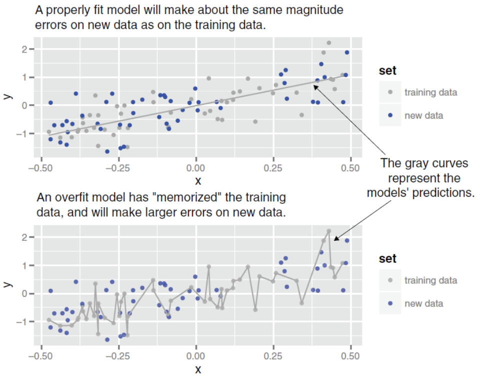
A model can be scored with

- a bonus proportional to its scaled log likelihood on the calibration data
- minus a penalty proportional to the complexity of the model.

A bit ad hoc, but tends to work well in selecting models.

## Scoring models using AIC (example)

Using the same example before, while the bottom model has a good fit to the training set, it is unlikely to fit the calibration set well. Furthermore, it is significantly penalised for the complexity of the model. The top model is more likely to have a higher AIC score and is preferred.



## Feature Selection

## Evaluation measures

If we are interested in finding the best single-variable model (or *single-feature model*), then we need to have some evaluation measure for comparing different single-variable models.

Log likelihood and deviance are both suitable measures.

- We prefer the feature that yields that largest log likelihood (non-positive) value.
- We prefer the feature that yields the smallest deviance.

We don't need to use AIC as all these single-variable models have the same model complexity.

## Step 1: compute log likelihood

For binary classification, the log likelihood is

$$\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

(for each $i$, the ground truth $y_i$ is either $0$ or $1$ and $p_i$ is the predicted probability)

```
pos <- '1'
# Define a function to compute log likelihood so that we can reuse it.
logLikelihood <- function(ytrue, ypred, epsilon=1e-6) {
  sum(ifelse(ytrue==pos, log(ypred+epsilon), log(1-ypred-epsilon)), na.rm=T)
}

# Compute the likelihood of the Null model on the calibration
# set (for the KDD dataset from previous lecture)
outcome <- 'churn'
logNull <- logLikelihood(
    dCal[,outcome], sum(dCal[,outcome]==pos)/nrow(dCal)
  )
cat(logNull)
## -1178.017
```

## Step 2: Run through categorical variables

Pick variables based on their reduction on *deviance* with respect to the Null deviance.

```
selCatVars <- c()
minDrop <- 10  # may need to adjust this number

for (v in catVars) {
  pi <- paste('pred', v, sep='')
  devDrop <- 2*(logLikelihood(dCal[,outcome], dCal[,pi]) - logNull)
  if (devDrop >= minDrop) {
    print(sprintf("%s, deviance reduction: %g", pi, devDrop))
    selCatVars <- c(selCatVars, pi)
  }
}
## [1] "predVar205, deviance reduction: 24.2321"
## [1] "predVar206, deviance reduction: 34.4432"
## [1] "predVar210, deviance reduction: 10.668"
## [1] "predVar218, deviance reduction: 13.2456"
## [1] "predVar221, deviance reduction: 12.4097"
## [1] "predVar225, deviance reduction: 22.9071"
## [1] "predVar228, deviance reduction: 15.9645"
## [1] "predVar229, deviance reduction: 24.4944"
```

## Step 2: Run through categorical variables (cont.)

From the result on the previous slide, we can see that the predicted values on the calibration set using `Var206` gives the largest reduction to the *deviance*. This agrees with last week's lecture where `Var206` gave the highest AUC (0.59).

## Step 3: Run through numerical variables

Similarly for the numerical variables.

```
selNumVars <- c()
minDrop <- 10  # may need to adjust this number
for (v in numericVars) {
  pi <- paste('pred', v, sep='')
  devDrop <- 2*(logLikelihood(dCal[,outcome], dCal[,pi]) - logNull)
  if (devDrop >= minDrop) {
    print(sprintf("%s, deviance reduction: %g", pi, devDrop))
    selNumVars <- c(selNumVars, pi)
  }
}
## [1] "predVar6, deviance reduction: 13.243"
## [1] "predVar7, deviance reduction: 18.6848"
## [1] "predVar13, deviance reduction: 10.0631"
## [1] "predVar28, deviance reduction: 11.3863"
## [1] "predVar72, deviance reduction: 12.535"
## [1] "predVar73, deviance reduction: 48.2523"
## [1] "predVar74, deviance reduction: 19.6323"
## [1] "predVar113, deviance reduction: 23.1358"
## [1] "predVar126, deviance reduction: 74.9558"
## [1] "predVar140, deviance reduction: 16.1815"
## [1] "predVar144, deviance reduction: 15.9856"
```

## Step 3: Run through numerical variables (cont.)

For the numerical variable, `Var126` is the winner, giving a drop of 74.9556 for the deviance.

From last week's lecture, `Var126` gave an AUC of 0.6288453 which was also the highest in the list.

## References

- **Practical Data Science with R** (Second Edition). *Nina Zumel and John Mount*, Manning, 2020. (Chapters 6, pages 192-195, 255)

# Multi-Variable Classification: Decision Trees

## CITS4009 Computational Data Analysis

Unit Coordinator: Dr Du Huynh

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2022

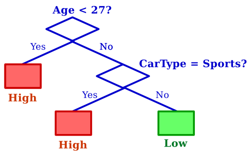
## Decision Tree Models

Decision trees are a simple model type – they make a prediction that is *piecewise constant*.

- Recall that the null models often give single constants for the whole dataset.
- The construction of a decision tree involves splitting the training data into pieces and using a simple constant on each piece.

Decision trees can be used to quickly predict categorical or numeric outcomes.

| PID | Age | CarType | Class |
|---|---|---|---|
| 0 | 23 | Family | High |
| 1 | 17 | Sports | High |
| 2 | 43 | Sports | High |
| 3 | 68 | Family | Low |
| 4 | 32 | Truck | Low |
| 5 | 20 | Family | High |



1) Age $< 27 \Rightarrow$ High

2) Age $\geq 27$ and CarType $=$ Sports $\Rightarrow$ High

3) Age $\geq 27$ and CarType $\neq$ Sports $\Rightarrow$ Low

## Decision Tree learning

Decision trees are binary trees. A decision tree is built by iteratively

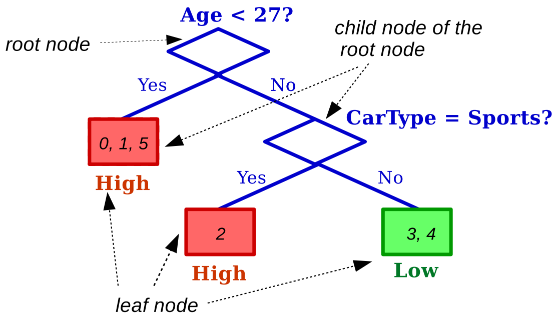
- finding the optimal feature out of all the features and the optimal threshold to **split** a *node* of the tree, e.g., at the *root node*, `Age` is the optimal feature and 27 is the optimal threshold found by the decision trees algorithm.

This splitting process results in training instances being divided and passed down the branches to the two *child nodes*. As we progress down the decision tree, there are fewer and fewer training instances in each node.

The splitting of a node can be terminated by any of the following criteria:

- the node contains only instances of the same class;
- the node is at the pre-defined maximum depth value for the tree;
- the node has too few instances for further splitting.

## Decision Tree learning (cont.)



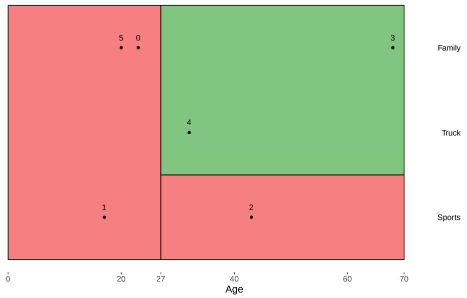
In this example, all the leaf nodes are *pure*, meaning that they contain instances of only one class. Person IDs 0, 1, and 5 are in the leaf node in the far left; person ID 2 is in the middle leaf node; person ID 3 and 4 are in the right leaf node.

## Decision Tree learning (cont.)

We can also consider that the objective behind decision tree methods is to partition the feature space into homogeneous regions (i.e. having instances belonging to one class only) as much as possible. Also, the regions should not be narrow and long.

尽可能多地将特征空间划分为同质区域(即只属于一个类的实例)。此外，区域不能太窄太长

E.g., for the decision tree example on the previous slide, the *Age-CarType* space is partitioned into 3 regions shown below:



## Decision Tree learning (cont.)

Notable Decision Trees algorithms include:

- ID3 (Iterative Dichotomiser 3)
- C4.5 (successor of ID3)
- CART (Classification And Regression Tree)
- CHAID (CHi-squared Automatic Interaction Detector).
- MARS: extends decision trees to handle numerical data better.
- Conditional Inference Trees.

## Decision Tree – Pros:

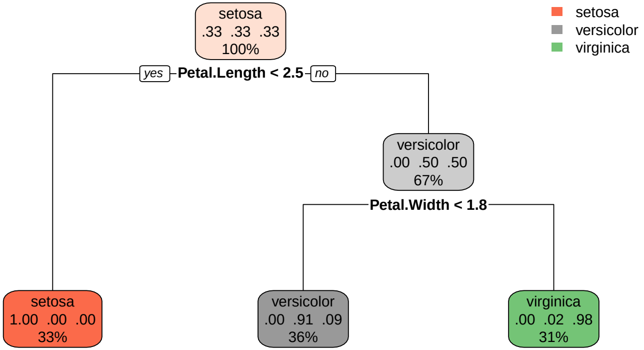
- Decision trees take any type of data – numerical or categorical – without any distributional assumptions and without preprocessing.
- Most implementations (in particular, R) handle missing data; the method is also robust to redundant and non-linear data.
- The algorithm is easy to use, and the output (the tree) is relatively easy to understand.
- They naturally express certain kinds of interactions among the input variables: those of the form "`IF x is true AND y is true, THEN...`"
- Once the model is fit, scoring is fast.

## Decision Tree – Drawbacks

- They have a tendency to overfit, especially without pruning.
- They have high training variance: samples drawn from the same population can produce trees with different structures and different prediction accuracy.
- Simple decision trees are not as reliable as other tree-based ensemble methods: e.g., random forests.

## Decision Tree using `rpart`

```r
library(rpart)
dt <- rpart(formula = Species ~ ., data = iris)
library(rpart.plot)
rpart.plot(dt)
```



(Some default hyperparameter values used by `rpart`: `minsplit=20`, `cp=0.01`, `maxdepth=30`)

## Building decision tree models using `rpart()`

```
library('rpart')
(fV <- paste(outcome,'> 0 ~ ',
             paste(c(catVars, numericVars), collapse=' + '),
             sep=''))
## [1] "churn> 0 ~ Var191 + Var192 + Var193 + Var194 + Var195 + Var196 + Var197 + V
tmodel <- rpart(fV, data=dTrain)

print(calcAUC(predict(tmodel, newdata=dTrain), dTrain[,outcome]))
## [1] 0.9241265
print(calcAUC(predict(tmodel, newdata=dTest), dTest[,outcome]))
## [1] 0.5266172
print(calcAUC(predict(tmodel, newdata=dCal), dCal[,outcome]))
## [1] 0.5126917
```

Notice that the AUC score on the training set is 0.9241265 – almost a perfect score!

## Replacing categorical values with numeric

The model looks way too good to believe on the training data and not as good as our best single-variable models on the calibration and test data.

A couple of possible sources of the failure are

- quite a few categorical variables have many levels, and
- a lot more NAs/missing data than `rpart()`'s surrogate value strategy was designed for.

What we can do to work around this is fit on our reprocessed variables, which hide the categorical levels (replacing them with numeric predictions), and remove NAs (treating them as just another level).

## A decision tree model with reprocessed variables

Rather than using the original catVars and numericVars variables, we can use the predVar variables that we obtained from our investigation using the single variable models.

```
tVars <- paste('pred', c(catVars, numericVars), sep='')
cat(tVars[1:5])
## predVar191 predVar192 predVar193 predVar194 predVar195
(fV2 <- paste(outcome,'>0 ~ ',
            paste(tVars, collapse=' + '), sep=''))
## [1] "churn>0 ~ predVar191 + predVar192 + predVar193 + predVar194 + predVar195 +
# rpart stands for "recursive partitioning and regression trees"
tmodel <- rpart(fV2, data=dTrain)
# To inspect the model, type: summary(tmodel)

print(calcAUC(predict(tmodel, newdata=dTrain), dTrain[,outcome]))
## [1] 0.928669
print(calcAUC(predict(tmodel, newdata=dCal), dCal[,outcome]))
## [1] 0.5384152
print(calcAUC(predict(tmodel, newdata=dTest), dTest[,outcome]))
## [1] 0.5390648
```

We can see an improvement to the AUC values for dTrain, dCal, and dTest.

## Performance Measures

The following displays the performance measures in terms of *accuracy*, *precision*, *recall*, and *f1 score*.

```r
# ytrue should be a vector containing TRUE and FALSE
# ypred should be a vector of probabilities
logLikelihood <- function(ytrue, ypred, epsilon=1e-6) {
  sum(ifelse(ytrue, log(ypred+epsilon), log(1-ypred+epsilon)), na.rm=T)
}

# ytrue should be a vector containing 1s (or TRUE) and 0s (or FALSE);
# ypred should be a vector containing the predicted probability values for the target class.
# Both ytrue and ypred should have the same length.
performanceMeasures <- function(ytrue, ypred, model.name = "model", threshold=0.5) {
   # compute the normalised deviance
   dev.norm <- -2 * logLikelihood(ytrue, ypred)/length(ypred)
   # compute the confusion matrix
   cmat <- table(actual = ytrue, predicted = ypred >= threshold)
   accuracy <- sum(diag(cmat)) / sum(cmat)
   precision <- cmat[2, 2] / sum(cmat[, 2])
   recall <- cmat[2, 2] / sum(cmat[2, ])
   f1 <- 2 * precision * recall / (precision + recall)
   data.frame(model = model.name, precision = precision,
              recall = recall, f1 = f1, dev.norm = dev.norm)
}
```

## Pander formatting

```r
panderOpt <- function(){
  library(pander)
  # setting up Pander Options
  panderOptions("plain.ascii", TRUE)
  panderOptions("keep.trailing.zeros", TRUE)
  panderOptions("table.style", "simple")

}
```

## Prettier Performance Table Function

```r
# A function to pretty print the performance table of a model
# on the training and test sets.
pretty_perf_table <- function(model, xtrain, ytrain,
                              xtest, ytest, threshold=0.5) {
   # Option setting for Pander
   panderOpt()
   perf_justify <- "lrrrr"

   # call the predict() function to do the predictions
   pred_train <- predict(model, newdata=xtrain)
   pred_test <- predict(model, newdata=xtest)

   # comparing performance on training vs. test
   trainperf_df <- performanceMeasures(
      ytrain, pred_train, model.name="training", threshold=threshold)
   testperf_df <- performanceMeasures(
      ytest, pred_test, model.name="test", threshold=threshold)

   # combine the two performance data frames using rbind()
   perftable <- rbind(trainperf_df, testperf_df)
   pandoc.table(perftable, justify = perf_justify)
}
```

## Pretty Print of a performance table

```
# tVars contains the reprocessed variables (defined 4 slides back)
pretty_perf_table(tmodel, dTrain[tVars], dTrain[,outcome]==pos,
                  dTest[tVars], dTest[,outcome]==pos)
##
##
## model        precision    recall        f1   dev.norm
## ---------- ----------- --------- --------- ----------
## training        0.7468   0.54348   0.62911     0.2461
## test            0.1000   0.04533   0.06238     1.0043
```

For the *precision*, *recall*, and *f1* columns, the higher is the value, the better. For the normalised deviance (*dev.norm*) column, the lower is the value, the better.

## Plotting the AUC

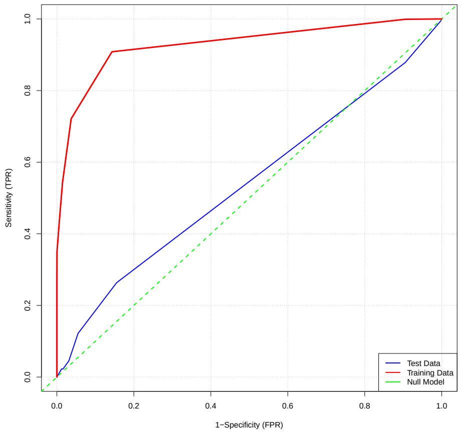
```r
library(ROCit)
plot_roc <- function(predcol1, outcol1, predcol2, outcol2){
    roc_1 <- rocit(score=predcol1, class=outcol1==pos)
    roc_2 <- rocit(score=predcol2, class=outcol2==pos)
    plot(roc_1, col = c("blue","green"), lwd = 3,
      legend = FALSE,YIndex = FALSE, values = TRUE, asp=1)
    lines(roc_2$TPR ~ roc_2$FPR, lwd = 3,
          col = c("red","green"), asp=1)
    legend("bottomright", col = c("blue","red", "green"),
       c("Test Data", "Training Data", "Null Model"), lwd = 2)
}
pred_test_roc <- predict(tmodel, newdata=dTest)
pred_train_roc <- predict(tmodel, newdata=dTrain)
```

In the code above, `plot()` plots the curves in a new figure; `lines()` does the same thing but plots the curves in the current (must exist) figure.

A ROC plot should have a square shape. We specify `asp=1` in the code to set the aspect ratio to 1. (In the R notebook, I use ` ```{r fig.asp=1}`)

## Plot to compare AUC

```
plot_roc(pred_test_roc, dTest[[outcome]],
         pred_train_roc, dTrain[[outcome]])
```



## Further improvement

The model still performs quite poorly on calibration data.

So our next suspicion is that the overfitting is because our model is too complicated. We build a new `tmodel2` by calling `rpart()` with the extra `control` argument.

```
tmodel2 <- rpart(fV2, data=dTrain,
                control=rpart.control(cp=0.001, minsplit=1000,
                                      minbucket=1000, maxdepth=5))

print(calcAUC(predict(tmodel2, newdata=dTrain[tVars]), dTrain[,outcome]))
## [1] 0.9421195
print(calcAUC(predict(tmodel2, newdata=dTest[tVars]), dTest[,outcome]))
## [1] 0.5794633
print(calcAUC(predict(tmodel2, newdata=dCal[tVars]), dCal[,outcome]))
## [1] 0.547967
```

(Type: `?rpart.control` to learn more about the parameters `cp` (complexity parameter), `minsplit`, `minbucket`, and `maxdepth`)

## Pretty-print of Performance Tables

```
# tVars contains the reprocessed variables.
# Compare tmodel and tmodel2
pretty_perf_table(tmodel, dTrain[tVars], dTrain[,outcome]==pos, dTest[tVars], dTest
##
##
## model        precision    recall        f1   dev.norm
## ---------- ----------- --------- --------- ----------
## training        0.7468   0.54348   0.62911     0.2461
## test            0.1000   0.04533   0.06238     1.0043

pretty_perf_table(tmodel2, dTrain[tVars], dTrain[,outcome]==pos, dTest[tVars], dTes
##
##
## model        precision    recall        f1   dev.norm
## ---------- ----------- --------- --------- ----------
## training       0.73768   0.31037   0.43691     0.2775
## test           0.07576   0.01416   0.02387     0.8926
```

## Using selected features

By setting the `minsplit`, `minbucket`, and `maxdepth` hyperparameters appropriately helped to improve only the AUC of the model. The next thing that we try is using only the reprocessed numerical variables that achieved high AUC scores.

```
selNumVars
##  [1] "predVar6"   "predVar7"   "predVar13"  "predVar28"  "predVar72"
##  [6] "predVar73"  "predVar74"  "predVar113" "predVar126" "predVar140"
## [11] "predVar144" "predVar189"
(f <- paste(outcome,'>0 ~ ',
         paste(selNumVars, collapse=' + '), sep=''))
## [1] "churn>0 ~ predVar6 + predVar7 + predVar13 + predVar28 + predVar72 + predVar
tmodel3 <- rpart(f, data=dTrain,
                control=rpart.control(cp=0.001, minsplit=1000,
                                      minbucket=1000, maxdepth=5))
print(calcAUC(predict(tmodel3, newdata=dTrain[selNumVars]), dTrain[,outcome]))
## [1] 0.6784064
print(calcAUC(predict(tmodel3, newdata=dTest[selNumVars]), dTest[,outcome]))
## [1] 0.6633351
print(calcAUC(predict(tmodel3, newdata=dCal[selNumVars]), dCal[,outcome]))
## [1] 0.6668087
```

## Performance comparison of tmodel, tmodel2, and tmodel3

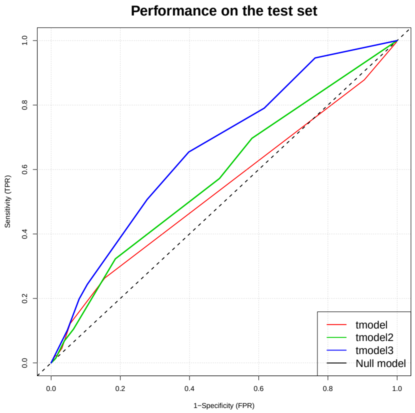
```
cat('Number of features used in tmodel and tmodel2 is:', length(tVars))
## Number of features used in tmodel and tmodel2 is: 212
cat('Number of features used in tmodel3 is:', length(selNumVars))
## Number of features used in tmodel3 is: 12
```

Compared to tmodel and tmodel2, tmodel3 used only 12 feature columns. While tmodel3 increased the AUC values to around 0.66 for both the calibration and test sets, its precision, recall, and f1 score were very poor. The output predicted probabilities on the test instances from tmodel3 were all below 0.2. Only if we make the model more sensitive to *churn* detection (by choosing a low threshold value such as 0.1) that we got the following:

```
pretty_perf_table(tmodel3, dTrain[selNumVars], dTrain[,outcome]==pos,
                  dTest[selNumVars], dTest[,outcome]==pos, threshold=0.1)
##
##
## model        precision   recall       f1   dev.norm
## ---------- ----------- -------- -------- ----------
## training        0.1338   0.5475   0.2151     0.4977
## test            0.1225   0.5071   0.1974     0.4869
```

## ROC plots of `tmodel`, `tmodel2` and `tmodel3`

Recall that both `tmodel` and `tmodel2` use the ewprocessed `predVar` features (variable `tVars`) but `tmodel2` has hyperparameters `maxdepth` and `minsplit` etc set to some predefined values to avoid overfitting, whereas `tmodel3` uses these same hyperparameter values but has only 12 features (variable `selNumVars`) to train on.



## Printing a decision tree

This is for your own inspection only. Do not present it to your client!

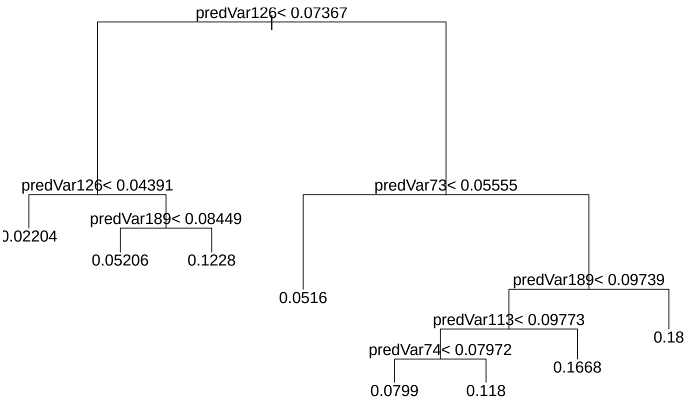
```
print(tmodel3)
## n= 40518
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 40518 2769.3550 0.07379436
##    2) predVar126< 0.07366888 18188  726.4097 0.04167583
##      4) predVar126< 0.04391312 8804  189.7251 0.02203544 *
##      5) predVar126>=0.04391312 9384  530.1023 0.06010230
##       10) predVar189< 0.08449448 8317  410.4571 0.05206204 *
##       11) predVar189>=0.08449448 1067  114.9166 0.12277410 *
##    3) predVar126>=0.07366888 22330 2008.9000 0.09995522
##      6) predVar73< 0.05554501 5872  287.3650 0.05160082 *
##      7) predVar73>=0.05554501 16458 1702.9070 0.11720740
##       14) predVar189< 0.0973886 14686 1433.4990 0.10962820
##         28) predVar113< 0.09772951 13073 1203.4430 0.10257780
##           56) predVar74< 0.0797246 5294  389.2015 0.07990178 *
##           57) predVar74>=0.0797246 7779  809.6668 0.11801000 *
##         29) predVar113>=0.09772951 1613  224.1389 0.16677000 *
##       15) predVar189>=0.0973886 1772  261.5728 0.18002260 *
```

## Interpreting a decision tree - this one has 15 nodes

- Node 1 is always called the *root*.
- A node with no children, is called a *leaf node*. Leaf nodes are marked with stars.
- Each node other than the root node has a parent, and the parent of node $k$ is node $\text{floor}(k/2)$.
- Each node other than the root is named by what condition must be true to move from the parent to the node.
  - e.g. from node 1 to node 2 if `predVar126 < 0.07379436`
  - So to score a row of data, we navigate from the root of the decision tree by the node conditions until we reach a leaf node.
- The remaining three numbers reported for each node are:
  - the number of training items that navigated to the node,
  - the deviance of the set of training items that navigated to the node (a measure of how much uncertainty remains at a given decision tree node),
  - the fraction of items that were in the positive class at the node (which is the prediction for leaf nodes).

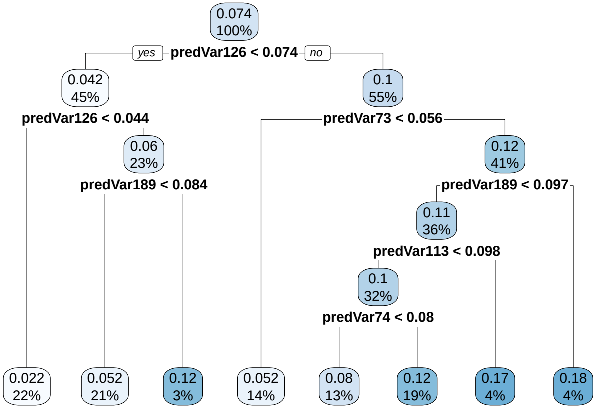
## Visualising a decision tree

```r
par(cex=1.2)
plot(tmodel3)
text(tmodel3)
```



## Visualising a decision tree

```r
par(cex=1.5)
rpart.plot(tmodel3)
```



## Poor performance of Our Decision Tree Models

The best guess is that this dataset is unsuitable for decision trees and a method that deals better with overfitting issues is needed – such as *random forests*.

## References

- **Practical Data Science with R** (Second Edition). *Nina Zumel and John Mount*, Manning, 2020. (Chapters 6, 10 (Sections 10.1, 10.1.1))
- **Decision Tree Algorithm:** https://medium.com/deep-math-machine-learning-ai/chapter-4-decision-trees-algorithms-b93975f7a1f1
- **Best Machine Learning Packages in R:** https://www.r-bloggers.com/what-are-the-best-machine-learning-packages-in-r/

# Multi-Variable Classification: K-Nearest Neighbours

## CITS4009 Computational Data Analysis

Unit Coordinator: Dr Du Huynh

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2022

# k-Nearest Neighbours (kNN)

kNN: predicting a property of a datum based on the datum or data that are most similar to it. It can be used for *regression* and *multi-class classification*. For example,

- Making product recommendations for a customer based on the purchases of other similar customers
- Predicting the final price of an auction item based on the final prices of similar products that have been auctioned in the past

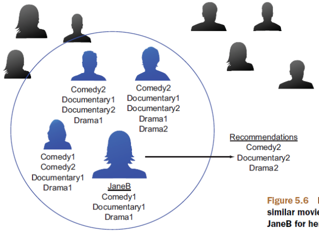


**Figure 5.6 Look to the customers with similar movie-watching patterns as JaneB for her movie recommendations.**

*Show me your friends, and I'll tell you who you are.*

## kNN – How it works

KNN注意事项：a) kNN needs to use distance values to find neighbours, so you need to normalise your features;b) you need to apply proper treatment to missing data, NAs, etc; c) use Hamming distance if you have categorical variables or convert them to numerical (not always possible).

- The value of *k* needs to be determined a priori.
  This determines the number of neighbours to be used for the prediction.
- A distance function (the default is the *Euclidean distance*) is needed as a measure of *nearness* between data points in the feature space.
- For categorical variables in the dataset, the *Hamming distance* (see later) should be used.

## kNN – A simple example (the `iris` dataset)

```
set.seed(32238)
iris <- iris
# relabel the Species column for binary classification
iris <- within(iris, Species <- ifelse(Species == "versicolor",
               "versicolor", "non-versicolor"))
# set up the response variable that we try to predict and the
# input feature columns
outcome <- 'Species'          # response variable
(features <- setdiff(colnames(iris), outcome))
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"

# split into training and calibration sets
intrain <- runif(nrow(iris)) < 0.75
train <- iris[intrain,]
calib <- iris[!intrain,]
cat('Training and calibration set sizes are:', nrow(train), 'and', nrow(calib))
## Training and calibration set sizes are: 116 and 34

library('class')
# The following is a no-no -- you always get 100% perfect predictions!
knnPred <- knn(train[features], train[features], train[,outcome], k=1, prob=T)
# exercise: inspect knnPred and compare it with train[outcome]
```

## kNN – A simple example (the `iris` dataset) (cont.)

```
# Now try the calibration set
knnPred <- knn(train[features], calib[features], train[,outcome], k=1, prob=T)
(accuracy <- sum(knnPred == calib[,outcome]) / nrow(calib))
## [1] 0.9117647
(conf_mat <- table(actual=calib[,outcome], predicted=knnPred)) # confusion matrix
##                    predicted
## actual               non-versicolor versicolor
##   non-versicolor                 20          1
##   versicolor                      2         11

# using more neighbours (usually needed for complex datasets)
knnPred <- knn(train[features], calib[features], train[,outcome], k=3, prob=T)
(accuracy <- sum(knnPred == calib[,outcome]) / nrow(calib))
## [1] 0.9411765
(conf_mat <- table(actual=calib[,outcome], predicted=knnPred))
##                    predicted
## actual               non-versicolor versicolor
##   non-versicolor                 21          0
##   versicolor                      2         11
```

## kNN – A simple example (the iris dataset) (cont.)

As we specify prob=TRUE when calling knn(), we can obtain the probabilities of the predictions from the output variable knnPred:

```
knnProb <- attributes(knnPred)$prob # prediction probability

# inspect a few cases
cases <- c(4, 14, 18, 26, 27)
calib[cases, outcome]     # ground truth
## [1] "non-versicolor" "versicolor"     "versicolor"     "non-versicolor"
## [5] "non-versicolor"
knnPred[cases]            # knn predictions
## [1] non-versicolor versicolor     versicolor     non-versicolor non-versicolor
## Levels: non-versicolor versicolor
knnProb[cases]            # predicted probabilities
## [1] 1 1 1 1 1
```

Notice that if the prediction for an instance is correct, we want the probability to be high (close to 1) – kNN is designed for multi-class problems; it doesn't output a zero (or small) probability for the non-target class.

## kNN – A simple example (the iris dataset) (cont.)

To use the same formula for the log likelihood in a previous lecture, we need to manually convert variable knnProb:

```
# for the non-target class, convert to 1-knnProb
knnProb <- ifelse(knnPred == "versicolor", knnProb, 1-knnProb)
```

Calculate the AUC:

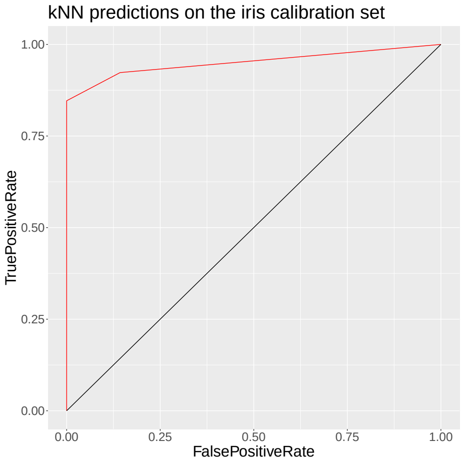
```
library(ROCR)
# ypred should be a vector of probabilities;
# ytrue should be a vector of TRUE and FALSE values or 1s and 0s.
calcAUC <- function(ypred, ytrue) {
  perf <- performance(prediction(ypred, ytrue), 'auc')
  as.numeric(perf@y.values)
}

knn_iris_AUC <- calcAUC(knnProb, calib[,outcome]=="versicolor")
cat('The AUC for kNN on the iris dataset is ', knn_iris_AUC)
## The AUC for kNN on the iris dataset is  0.9505495
```

## kNN – ROC plot for the iris dataset

```r
plotROC <- function(ypred, ytrue, titleString="ROC plot") {
  perf <- performance(prediction(ypred, ytrue), 'tpr', 'fpr')
  pf <- data.frame(FalsePositiveRate=perf@x.values[[1]],
                    TruePositiveRate=perf@y.values[[1]])
  ggplot() + geom_line(data=pf, aes(x=FalsePositiveRate, y=TruePositiveRate),
                        colour="red") +
    labs(title=titleString) +
    geom_line(aes(x=c(0,1), y=c(0,1))) +
    theme(text=element_text(size=24))
}
```

## kNN – ROC plot for the iris dataset (cont.)

```
plotROC(knnProb, calib[,outcome] == "versicolor",
        titleString="kNN predictions on the iris calibration set")
```

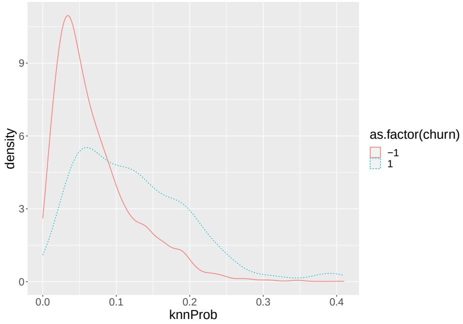


## kNN – Classification on the KDD dataset using 200 nearest neighbours

```
library(class)
nK <- 200
outcome <- 'churn'
knnTrain <- dTrain[,selVars]
knnCl <- dTrain[,outcome]==pos

knnPredict <- function(df) {
  knnDecision <- knn(knnTrain, df, knnCl, k=nK, prob=T)
  ifelse(knnDecision == TRUE,
         attributes(knnDecision)$prob,
         1 - attributes(knnDecision)$prob)
}

# create a new column in dCal and dTest to store the predicted probabilities
dCal$knnProb  <- knnPredict(dCal[,selVars])
dTest$knnProb <- knnPredict(dTest[,selVars])
print(calcAUC(dCal$knnProb, dCal[,outcome]))
## [1] 0.7205581
print(calcAUC(dTest$knnProb, dTest[,outcome]))
## [1] 0.7090571
```

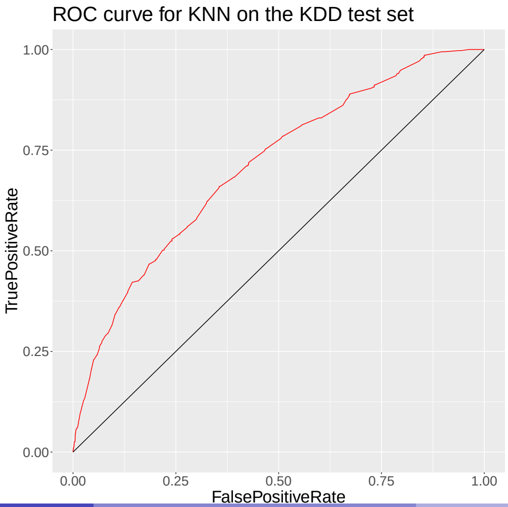
## kNN – Improved double density plot on KDD

```
ggplot(data=dCal) +
  geom_density(aes(x=knnProb, color=as.factor(churn),
                   linetype=as.factor(churn))) +
  theme(text=element_text(size=20))
```



## kNN – Plotting the ROC for KDD

```
plotROC(dTest$knnProb, dTest[,outcome]==pos,
        titleString="ROC curve for KNN on the KDD test set")
```



## kNN – Distance functions

欧几里得距离

- **Euclidean Distance** (a.k.a. **L2 Distance**)

Given two points $x_1$, $x_2 \in \mathbb{R}^n$, their Euclidean distance is defined as:

$$\text{euDist}(x_1, x_2) \equiv \sqrt{\sum_{i=1}^{n}(x_1[i] - x_2[i])^2} = \sqrt{(x_1[1] - x_2[1])^2 + ...(x_1[n] - x_2[n])^2}$$

Euclidean distance only makes sense when all the data is real-valued (quantitative). This is often referred to as straight-line distance. It is also referred to as the **L2 norm** of the difference vector $x_1 - x_2$.

Example (2D):

```
x1 <- c(1,1);  x2 <- c(5,4)
(L2dist <- sqrt(sum((x1-x2)^2)))
## [1] 5
```

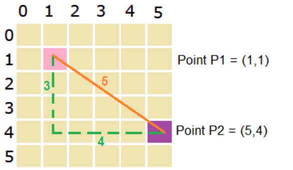
## kNN – Distance functions (cont.)

曼哈顿距离

- **Manhattan (City Block) Distance** (a.k.a. **L1 Distance**)

The Manhattan distance measures the total number of units along each dimension it takes to get from one (real-valued) point to the other (no diagonal moves). Also known as the **L1 norm** of the vector $x_1 - x_2$.

$$\text{mDist}(x_1, x_2) \equiv \sum_{i=1}^{n} |\, x_1[i] - x_2[i] \,| = \text{abs}(x_1[1] - x_2[1]) + \ldots + \text{abs}(x_1[n] - x_2[n])$$



Euclidean distance = $\sqrt{(5-1)^2 + (4-1)^2} = 5$

Manhattan distance = $|5-1| + |4-1| = 7$

```
(L1dist <- sum(abs(x1-x2)))
## [1] 7
```

## kNN – Distance functions (cont.)

L-infinity距离

- **L-infinity** distance between two points $x_1$ and $x_2$ is given by

$$\text{LinfDist} \equiv \max_{i=1,\cdots,n} |\, x_1[i] - x_2[i] \,|$$

Also known as the **L-infinity norm** of the difference vector $x_1 - x_2$.

In the diagram on the previous slide, the L-infinity distance of the two points is 4.

It is clear that $\underline{\text{LinfDist}(x_1, x_2) \leq \text{euDist}(x_1, x_2) \leq \text{mDist}(x_1, x_2)}$, for all $x_1$ and $x_2$.

## kNN – Distance functions (cont.)

汉明距离

- **Hamming Distance** – for Categorical Variables 在类型没有顺序时，同类为0异类为1

Hamming distance counts the number of mismatches. i.e., the distance is 0 if two values are in the same category (i.e., perfect match), and 1 otherwise.

If the categories are ordered (like `small/medium/large`) so that some categories are "closer" to each other than others, then you should convert them to a numerical sequence. E.g., map `small/medium/large` to `1/2/3` then use the L2, L1, or $L_\infty$ distance.

```
# suppose that x1 and x2 are categorical vectors of length 3.
x1 <- c("apple", "pear", "pear")
x2 <- c("apple", "banana", "apple")
hdist <- sum(x1 != x2)
cat("The Hamming distance between x1 and x2 is:", hdist)
## The Hamming distance between x1 and x2 is: 2
```

## kNN – Data preparation

Apart from splitting the dataset into a training set and a calibration set (assuming the test set is completely unknown), additional data preparation steps are required for kNN:

- Determine a set of $k$ values. Use the calibration set to help find the optimal $k$ value from the set.
- Numerical columns having too many NAs should be dropped from the set of input features.
- If the number of NAs and/or missing values is small in a numerical column, then they can be imputed by the median or mean value of the column; NAs in a categorical column may be treated as a separate level.
- Categorical columns whose levels can be ordered (e.g., small/medium/large) should be converted into numerical ones.
- kNN is sensitive to the different scales in the numerical columns. Normalization is usually required to ensure that there are no dominating numerical columns in the training set. Feature columns in the calibration and test set will need to be normalized the same way.
  Some libraries might perform normalization as default.

## KNN vs. Logistic Regression

KNN is expensive both in time and space. Sometimes we can get similar results with more efficient methods such as *logistic regression*.

```
f <- paste(outcome,'>0 ~ ', paste(selVars, collapse=' + '), sep='')
cat(f)
## churn>0 ~ predVar205 + predVar206 + predVar210 + predVar218 + predVar221 + predV

cat("Feature dimension = ", length(selVars))
## Feature dimension =  22

gmodel <- glm(as.formula(f), data=dTrain, family=binomial(link='logit'))

print(calcAUC(predict(gmodel, newdata=dTrain), dTrain[,outcome] > 0))
## [1] 0.7309537
print(calcAUC(predict(gmodel, newdata=dTest), dTest[,outcome] > 0))
## [1] 0.7234645
print(calcAUC(predict(gmodel, newdata=dCal), dCal[,outcome] > 0))
## [1] 0.7170824
```

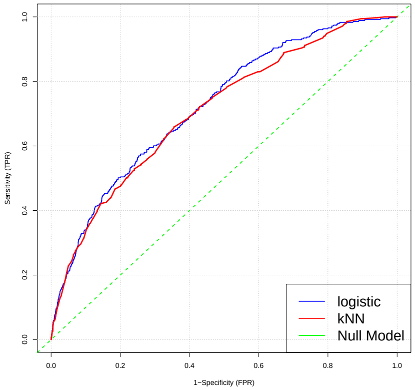
## Comparing the ROC curves

A slight modification of the ROC comparison function:

```r
library(ROCit)
plot_roc <- function(predcol1, outcol1, predcol2, outcol2){
  roc_1 <- rocit(score=predcol1, class=outcol1==pos)
  roc_2 <- rocit(score=predcol2, class=outcol2==pos)
  plot(roc_1, col = c("blue","green"), lwd = 3,
       legend = FALSE, YIndex = FALSE, values = TRUE, cex=3)
  lines(roc_2$TPR ~ roc_2$FPR, lwd = 3, col = c("red","green"), cex=3)
  legend("bottomright", col = c("blue","red", "green"),
         c("logistic", "kNN", "Null Model"), lwd = 2, cex=2)
}
```

## Plotting ROC curves of two models on the same dataset

```
pred_gmodel_roc <- predict(gmodel, newdata=dTest)
pred_knn_roc <- knnPredict(dTest[,selVars])
plot_roc(pred_gmodel_roc, dTest[,outcome],
         pred_knn_roc, dTest[,outcome])
```



## References

- **Practical Data Science with R** (Second Edition). *Nina Zumel and John Mount*, Manning, 2020. (Section 9.1.1)
- Best Machine Learning Packages in R: https://www.r-bloggers.com/what-are-the-best-machine-learning-packages-in-r/