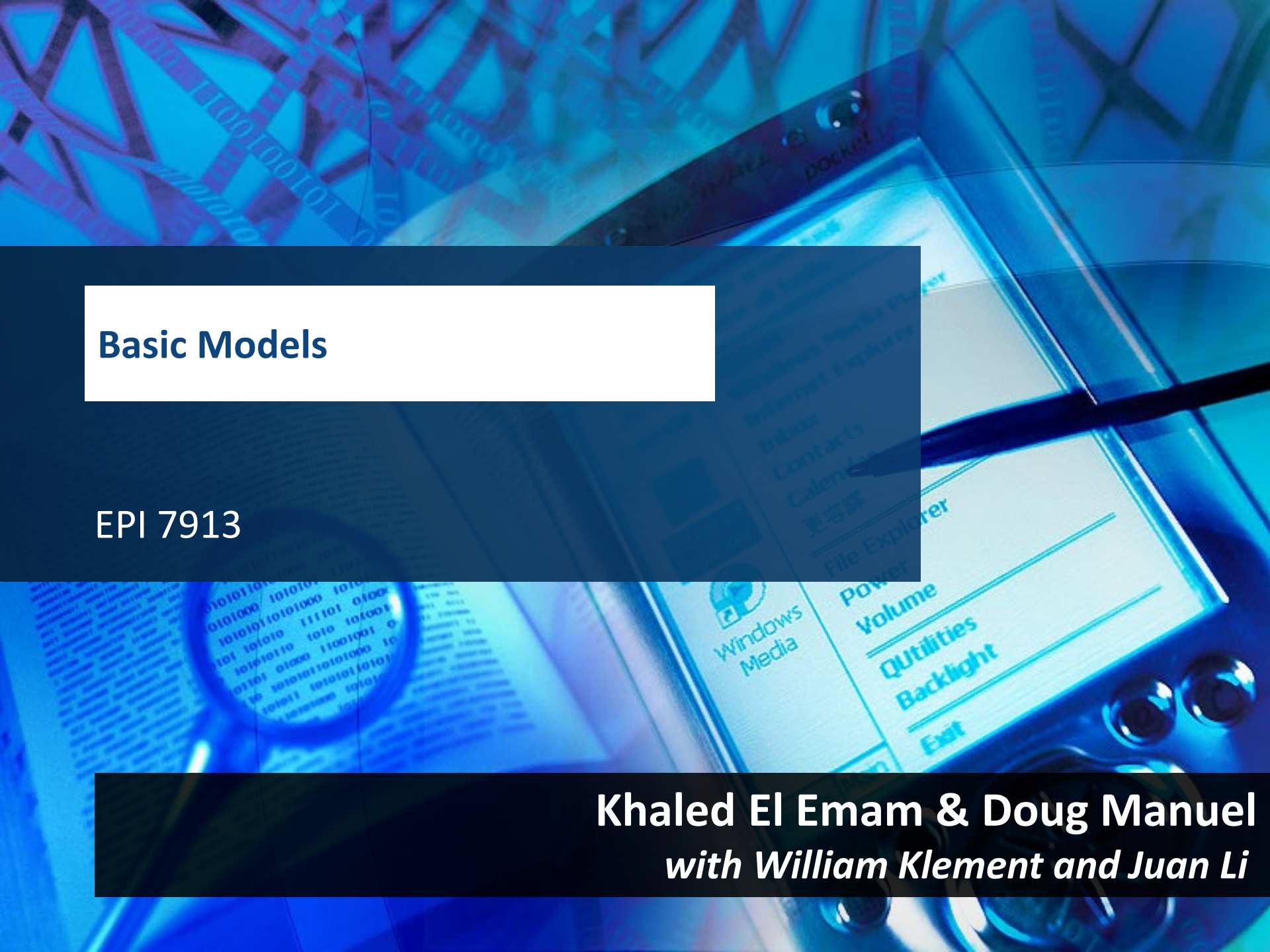



# Basic Models

EPI 7913

**Khaled El Emam & Doug Manuel**
***with William Klement and Juan Li***

# Basic machine learning techniques

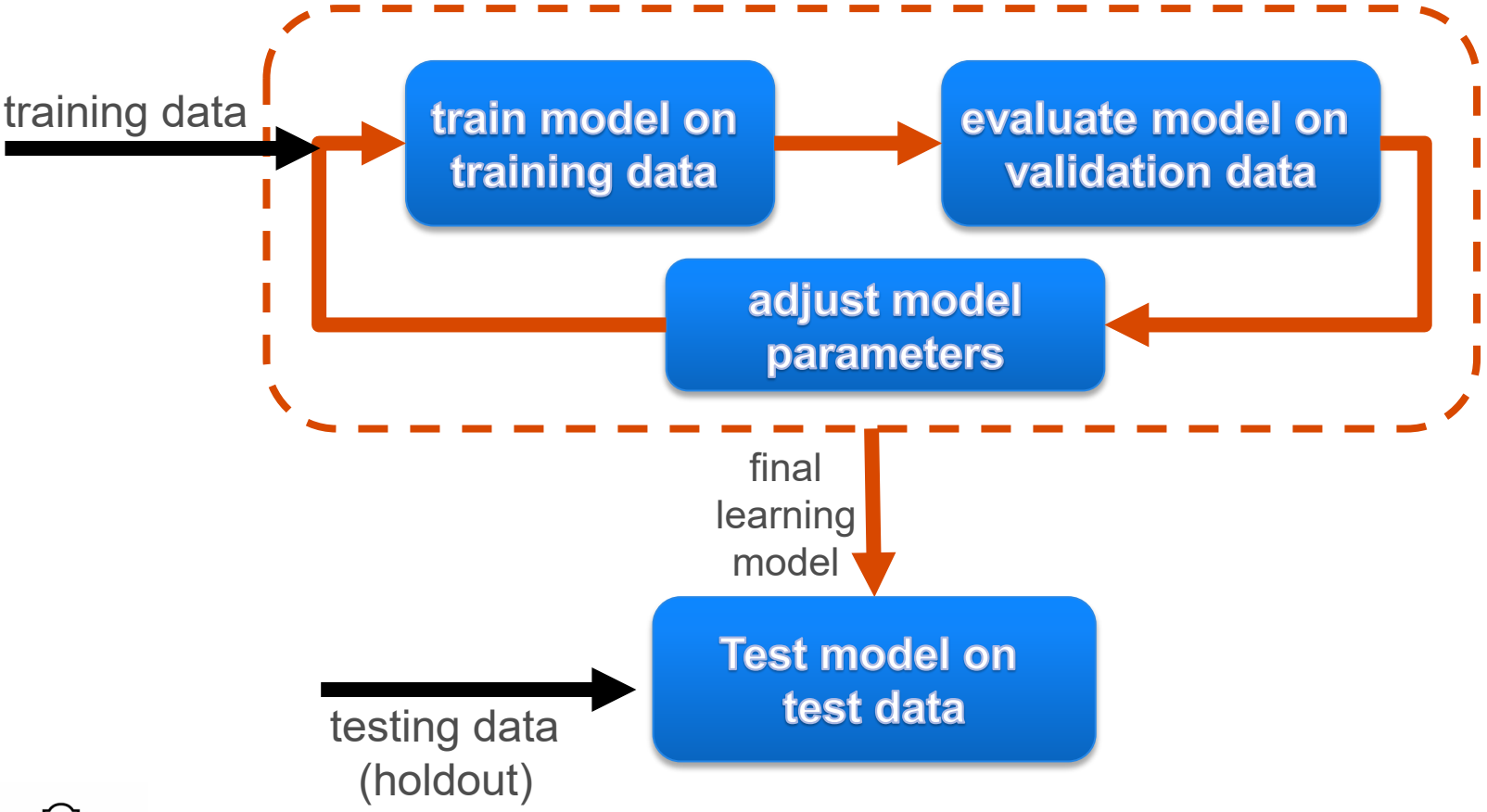
- Data splitting: training/validating/testing
- Basic learning algorithms:
  - KNN (rule based)
  - CART (tree based)
  - Rules vs. trees
- Basic hyper-parameter tuning methods:
  - Grid search

# Data Splitting

- **training** involves constructing the learning model on data points that represent the concept we wish the algorithm to learn.
  - **the algorithm uses the ground truth (labels) in data**
- **validating** allows us to refine (fine tune) the resulting model's parameters to make it perform closer to reality
  - **the model uses the ground truth for adjusting parameters**
- **testing** offers an assessment of the learning performance by testing the resulting model on a holdout data (testing)
  - **this allows the assessment of the generalization error**
- **the objective** is to maximize training (quantity and quality), as well as thoroughness of evaluation


T. J. Bradshaw, Z. Huemann, J. Hu, and A. Rahmim, “A Guide to Cross-Validation for Artificial Intelligence in Medical Imaging,” Radiol. Artif. Intell., vol. 5, no. 4, p. e220232, Jul. 2023, doi: 10.1148/ryai.220232.

# Why split the data?



# Training data

- is the data from which the model learns hidden features or patterns that predict the outcome
- training data should be diverse and cover as many scenarios as possible (broad exposure) to enable the model to predict on future, unseen cases (data points).
- in supervised learning, a representation of the gold standard (e.g., labels) is needed to allow the model to assess the relationship between features and predictions.

# Validation data

- the validation dataset is separate from the training set
- use it to validate how well the model performs during training
- the algorithm checks to verify if the model can predict reasonably well on cases for which we know the ground truth
- the algorithm adjusts parameters used by the model to improve prediction performance on this validation set (resist overfitting) – hyperparameter tuning

# Data splitting methods

- **Random:** randomly choose which row goes into which dataset without replacement
- **Stratified:** maintains the distribution of a target variable in each data split (default)
- **Grouped:** the split maintains the groups together in specified groups
  - Categorical: based on the groups
  - Continuous: based on number of bins
- **Blocked:** maintains the order of the rows when selecting the splits

# Data splitting in R

```
# first install the epi7913A package
lung <- epi7913A::lung

# install the splitTools package
install.packages("splitTools")

# load data splitting tools libraries
library(splitTools)

# initialize a random seed
set.seed(17)
inds <- splitTools::partition(lung.status,
                  p = c(train = 0.6, valid = 0.2, test = 0.2))
str(inds)

# retrieve the rows for each dataset
train <- lung[inds$train, ]
valid <- lung[inds$valid, ]
test  <- lung[inds$test, ]
```

# Example: Logistic Regression

- **Data**: lung dataset presented in previous class
- **Objective**: build a prediction model to predict if patients will live a certain number of days (based on a cut off value, 6 months= 365/2)
- **Method**: a logistic regression model predicts the probability of surviving >= cut off
- **Strategy**:
  - randomly split data into training & testing,
  - build model on training portion,
  - predict on test portion,
  - then evaluate (log loss of predicted vs actual)

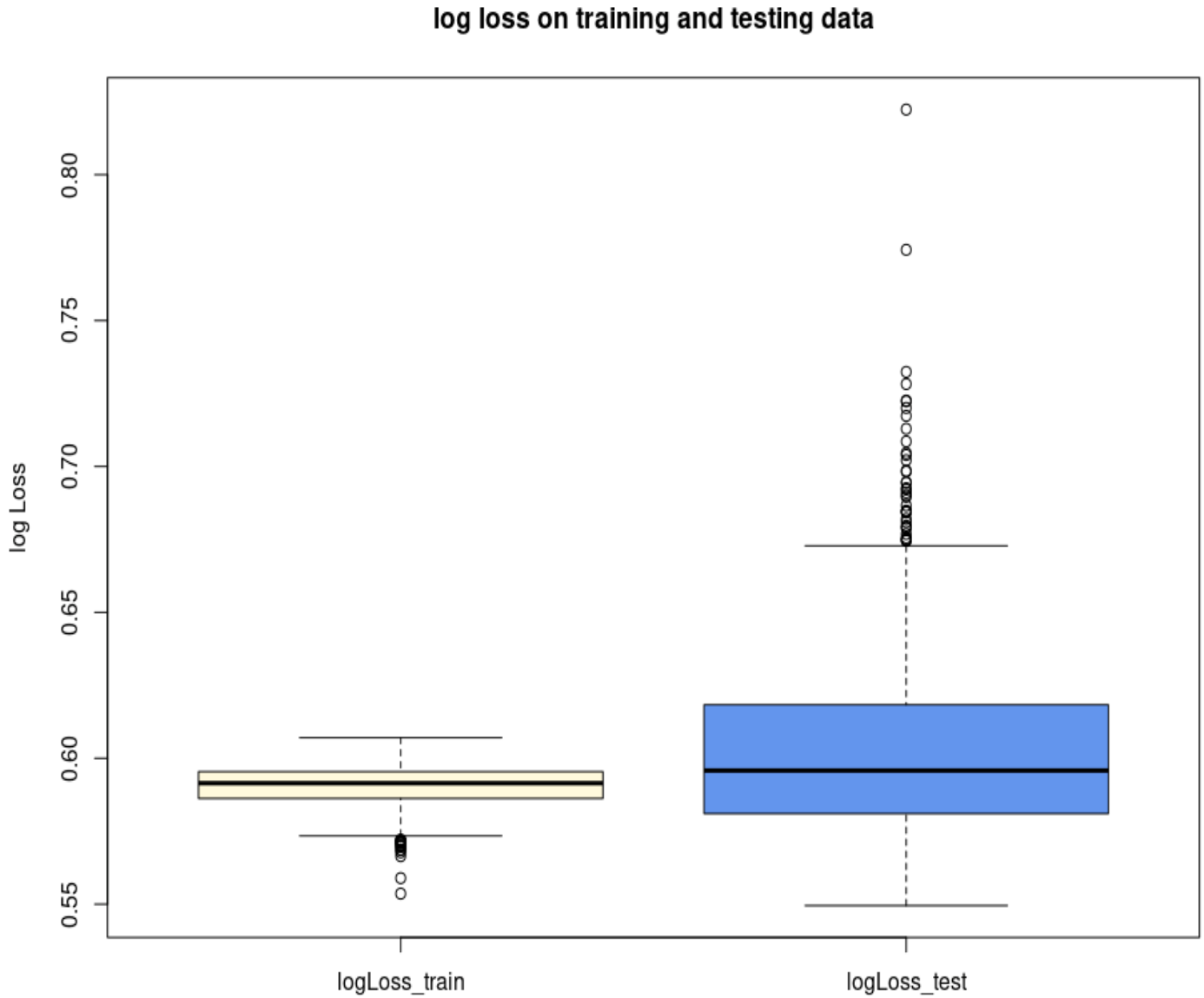
# Model performance assessment

- Goodness of fit: assess the performance of predictions on training data
- Model generalizability: assess how well predictions fit the holdout test data (generalization error)
- Use the notebook supplied with notes to experiment with the parameters

# Model development

For associated R code, please refer to notebook:

Lecture 3 – Logit Example.ipynb




Using the provided notebook, try:

- train_data_split = 0.6
- survival_cutoff = 365 or 365 * 2

# Instance-based learning

- constructs learning models form data points (instances) themselves by …
- comparing new instances to those in the training data
- it needs a similarity (or distance) metric
- complexity and memory demands increase with volume of data
- An example is K-nearest neighbors (KNN) classifiers

# Similarity & distance metrics

- distance = 1- similarity
- Euclidean distance = straight line (geometric distance) between two points
- Manhattan distance (or city block distance)
- Minkowski (general case)
- Binary (more specific case)

https://campus.datacamp.com/courses/cluster-analysis-in-r/calculating-distance-between-observations?ex=3

# General case:

- Minkowski's distance between two points:

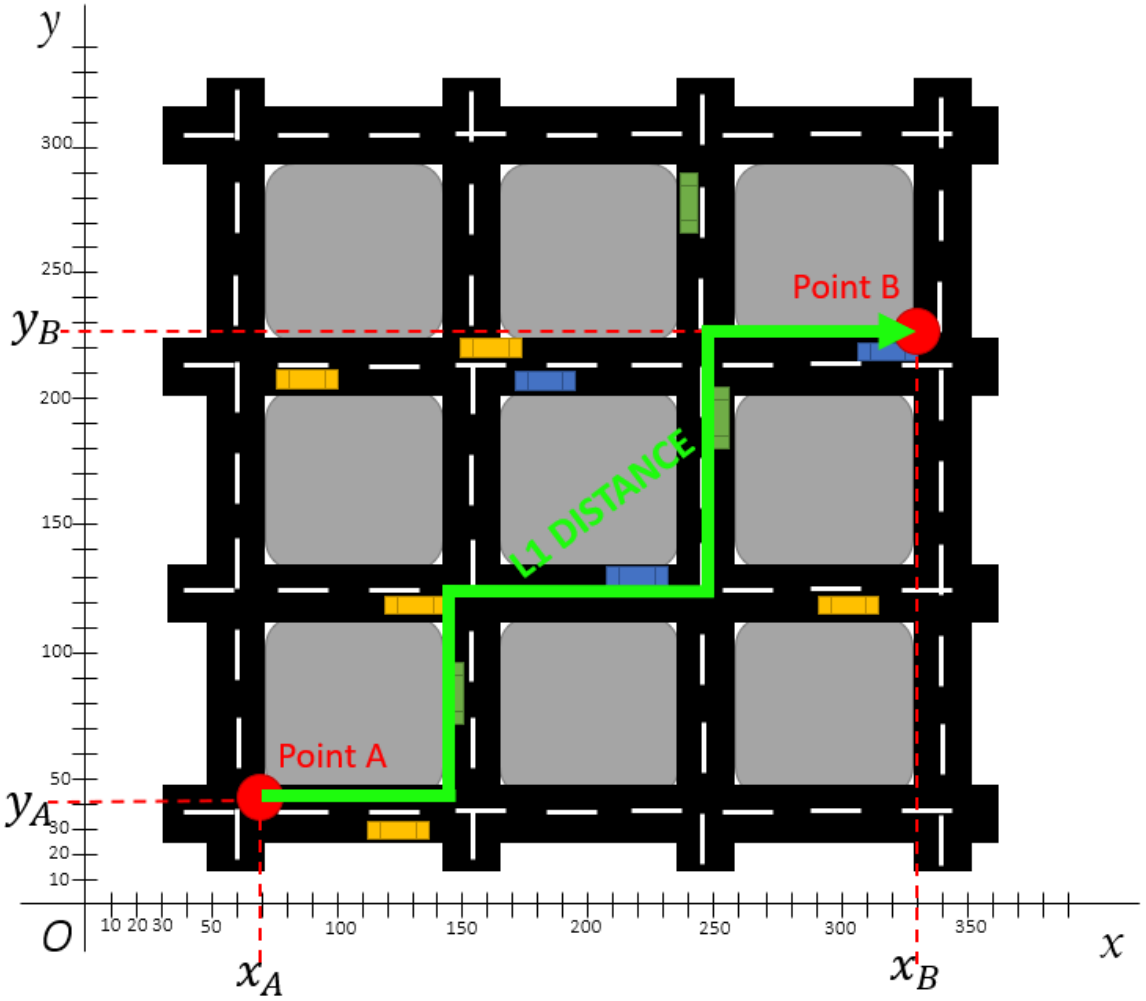
$$X = (x_1, x_2, \ldots, x_n) \text{ and } Y = (y_1, y_2, \ldots, y_n) \in \mathbb{R}^n$$

in dimension *p* is:

$$D(X, Y) = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

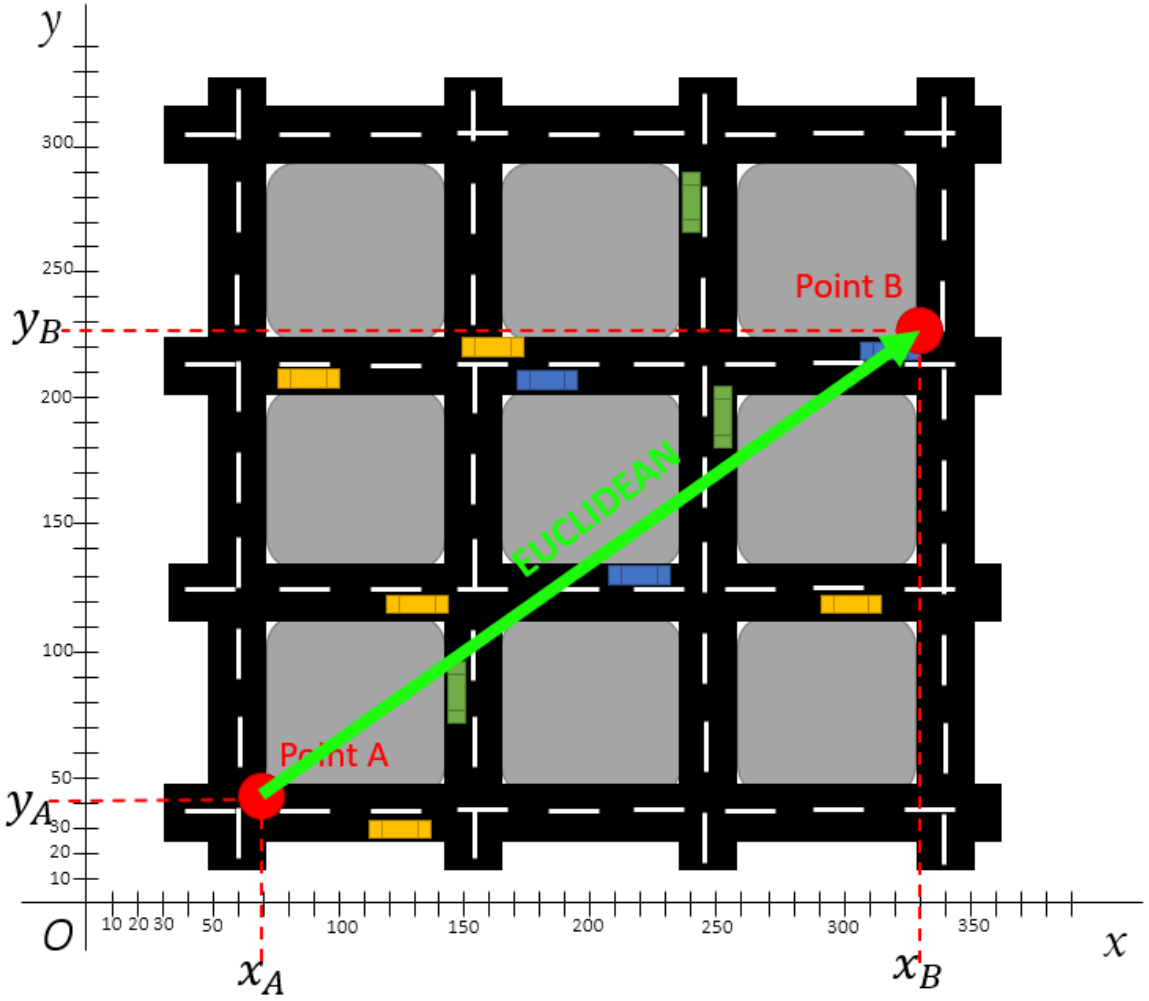
- $p = 1 \Rightarrow$ Manhattan (city block) distance
- $p = 2 \Rightarrow$ Euclidean distance

# Manhattan distance




L1 distance (when *p=1*), a.k.a. city block distance, or Manhattan distance moves forward in blocks (up/down or left/right).

# Euclidean distance




L2 distance (when *p*=2), a.k.a. Euclidean distance, moves in a straight line between points directly

# Extreme general case:

- $p$ approaches $+\infty$

$$\lim_{p\to\infty}\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}=\max_{i=1}^{n}|x_i-y_i|$$

- $p$ approaches $-\infty$

$$\lim_{p\to-\infty}\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{\frac{1}{p}}=\min_{i=1}^{n}|x_i-y_i|$$

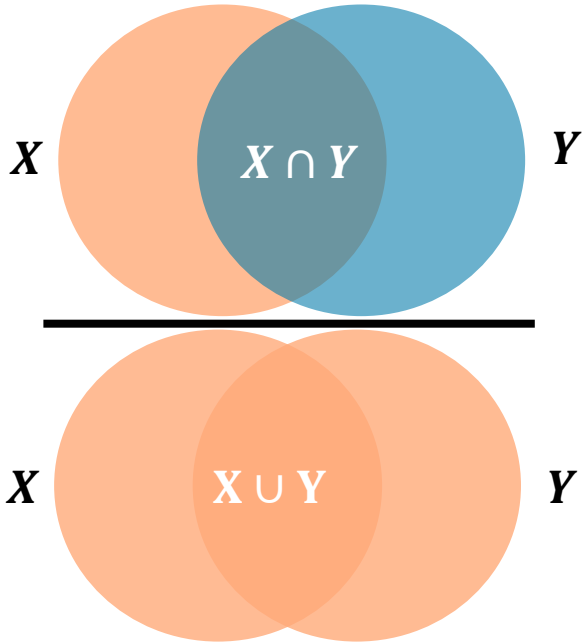
# Distance between categorical points

- Similarity between X and Y is the ratio of the number of elements in both sets to the total number of elements in either sets

$$Jaccard\ Similarity = JSim(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

- Distance = 1 – similarity

$$JDist(X,Y) = 1 - JSim(X,Y)$$



# Distance between binary variables

| | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
|---|---|---|---|---|---|---|---|
| X | T | F | F | T | F | T | T |
| Y | T | T | T | F | T | F | T |

*There are 7 individuals on two variables:*

$$Jaccard\ Similarity(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{2}{7} = 0.29$$

$$Jaccard\ Distance(X,Y) = 1 - Jaccard\ Similarity = 1 - 0.29 = 0.71$$

# Distance between categorical variables with many labels

- One-hot encode the multiclass variable
- A categorical feature has as many bits with value T as possible labels it can be assigned
- The data concatenates the binary vectors

| | F1 | | | F2 | | | F3 | | |
|---|---|---|---|---|---|---|---|---|---|
| X | T | F | F | T | F | F | T | F | F |
| Y | T | F | F | F | T | F | F | F | T |

*There are 3 individuals on two variables:*

$$Jaccard\ Similarity(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{1}{9} = 0.11$$

$$Jaccard\ Distance(X,Y) = 1 - Jaccard\ Similarity = 1 - 0.11 = 0.89$$

# Calculate distance in R

```
# define a function euclideanDist() to calculate Euclidean
# distance as the sum of squares of absolute difference between
# corresponding elements of X and Y
euclideanDistance <- function(X, Y){
  sqrt(sum((X - Y)^2))
}
# Initializing two equal length vectors
X <- c(2, 4, 4, 7)
Y <- c(1, 2, 2, 10)
print("Euclidean distance between X and Y is: ")
# euclideanDistance()
euclideanDistance(X, Y)

# The easy way is to use the built in dist() function
dist(rbind(X, Y), method="euclidean")

#try to see the list of methods and parameters
?dist
```

# What if the data was of mixed types?

- The previous distance measures were useful when all the data was of the same type (continuous or categorical)
- But how can we calculate the distance between two data points when their features (variables) have mixed types (some columns are categorical, and others are numeric)?
- Most solutions are based on calculating a weight for each variable to determine its contribution to the overall distance quantity

# Gower's distance for mixed data

The Gower distance is the complement to Gower similarity coefficient $S_{ij}$ between data points $i$ and $j$ and is determined by:

$$d_{ij} = 1 - S_{ij}$$

$$S_{ij} = \frac{\sum_{v=1}^{k} w_{ijv} s_{ijv}}{\sum_{v=1}^{k} w_{ij}} = \sum_{v=1}^{k} \frac{w_{ijv} s_{ijv}}{w_{ijv}}$$

The weight for variable $v$ is $w_{ijv} = \begin{cases} 0 & missing \\ 1 & otherwise \end{cases}$

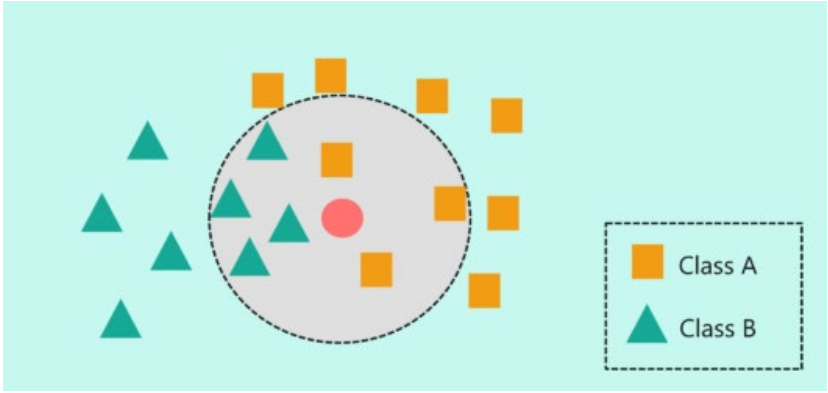
If $v$ is numeric, $S_{ijv} = \frac{|x_{iv} - x_{jv}|}{R_v}$ where $R_v$ is the range of $v$

If $v$ is binary (or categorical), $S_{ijv} \in \{0,1\}$

# K-nearest neighbors classification

- The target is to predict (Y/N) if a patient will survive a specified number of days
- We will add a new categorical variable "*alive_at_cutoff*" to indicate 1 for yes and 0 for no for every case when survival time is >= survival_cutoff value:
  - survival >= 1 year ➔ survival_cutoff = 365
  - Survival >= 6 months ➔ survival_cutoff = 365/2
- Use 1000 runs of random splitting ➔ train/test
- Compare log loss between train and test

# KNN Algorithm




- Is a simple supervised ML algorithm
- Classifies new data points into the target class based on the distance (dissimilarity) to features of its k-neighboring data points
- Is lazy: memorizes the training data instead of learning a discriminative function
- Slow and heavy on memory with large datasets

Image source: https://www.edureka.co/blog/knn-algorithm-in-r/

# Repeated train/test runs

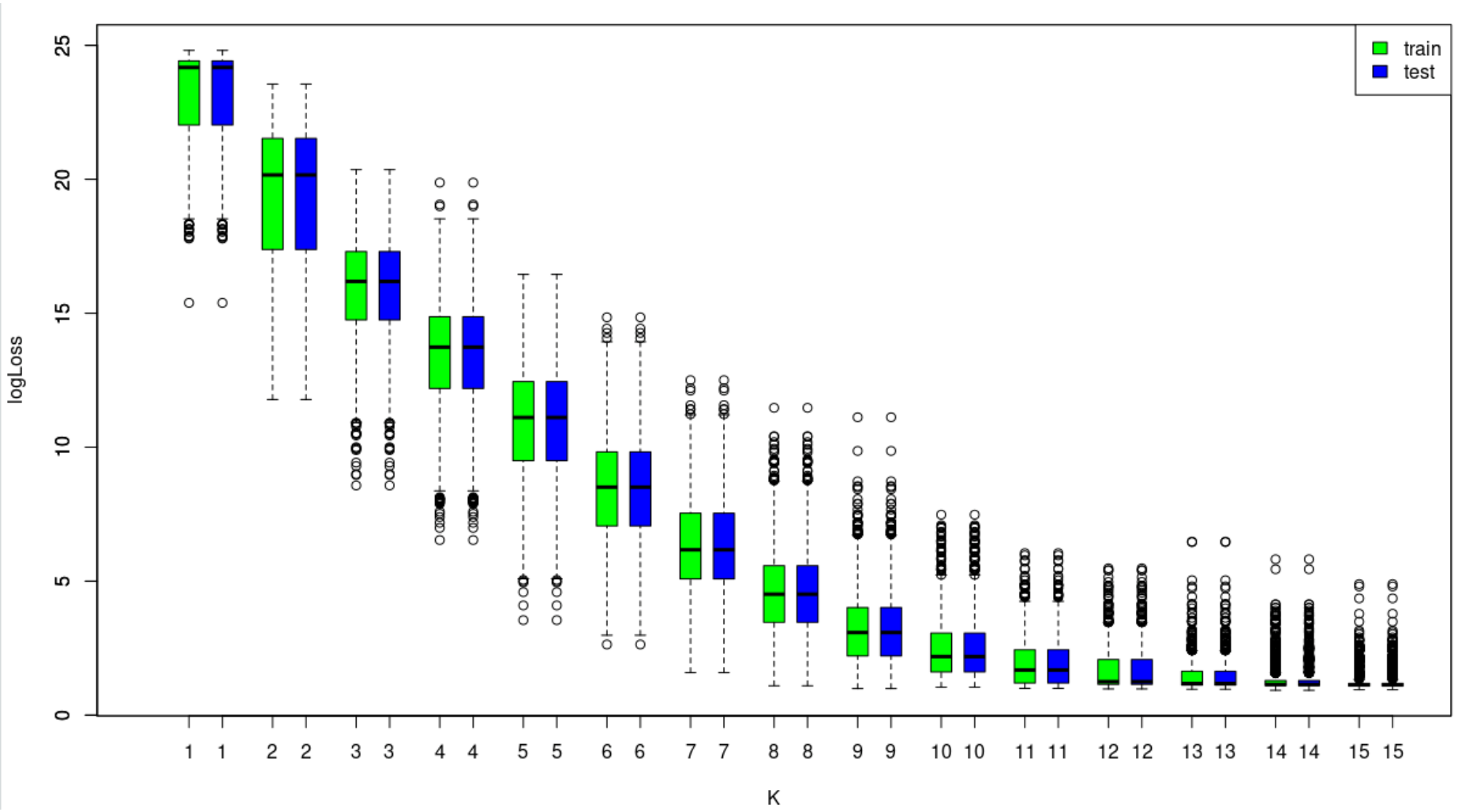
For associated R code, please refer to notebook:
Lecture 3 – KNN Example.ipynb

# Plot log loss

```
# plot box plots of log loss values on training and on testing
boxplot(evaluation_results, ylim=c(0,30),
  col = c("cornsilk","cornflowerblue"), ylab = "log Loss",
  main = paste("log loss on training and testing data k=",k))
```

- In the above case, we used K = 2
- What if we varied K? Will the results change?
- What is the optimal value for K?
- To answer the question, we repeat the experiment for various values of K and monitor the log loss values to see which yields the minimal log loss.

# Log Loss over values of K for a K-NN model in training and testing



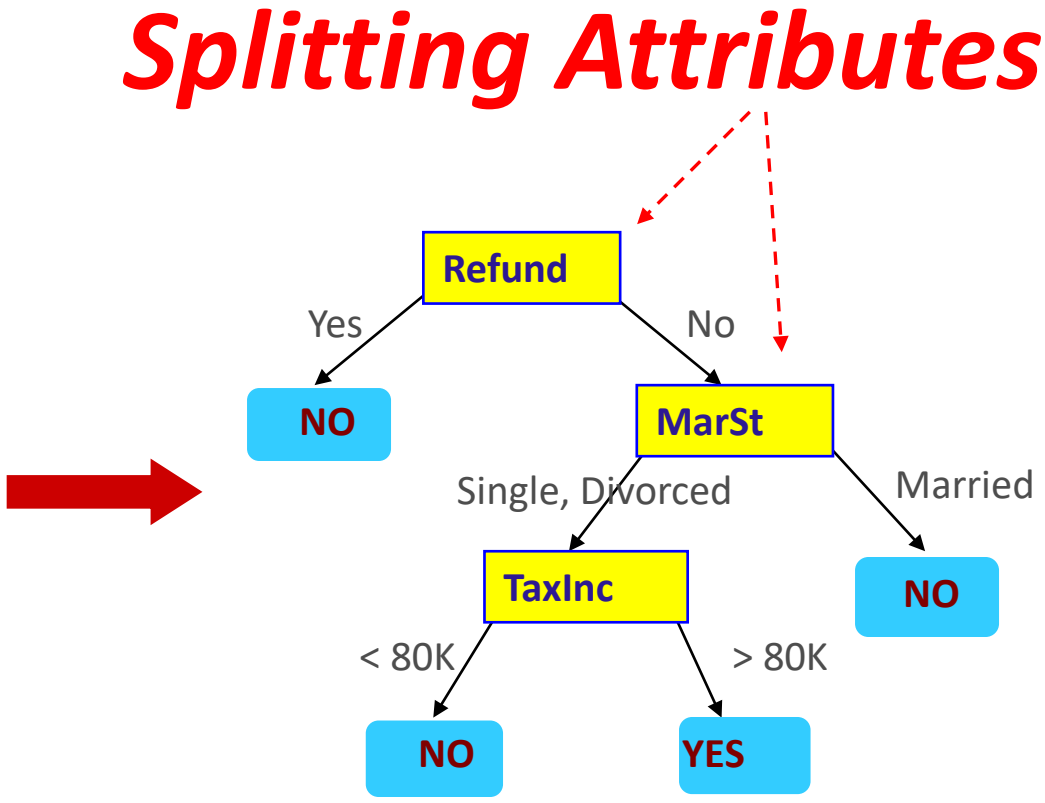
# Decision trees

- Decision tree is a supervised ML algorithm
- for both regression and classification problems
- works for categorical and/or continuous variables
- **Root node** represents the entire data sample and may be divided into more subsets
- **Splitting** is dividing a node into sub-trees to form decision-nodes (grows the tree)
- **Terminal nodes** do not divide any further
- **Pruning** is removing nodes and subtrees

# Example of a Decision Tree

| Tid | Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|---|
| | categorical | categorical | continuous | class |
| 1 | Yes | Single | 125K | No |
| 2 | No | Married | 100K | No |
| 3 | No | Single | 70K | No |
| 4 | Yes | Married | 120K | No |
| 5 | No | Divorced | 95K | Yes |
| 6 | No | Married | 60K | No |
| 7 | Yes | Divorced | 220K | No |
| 8 | No | Single | 85K | Yes |
| 9 | No | Married | 75K | No |
| 10 | No | Single | 90K | Yes |

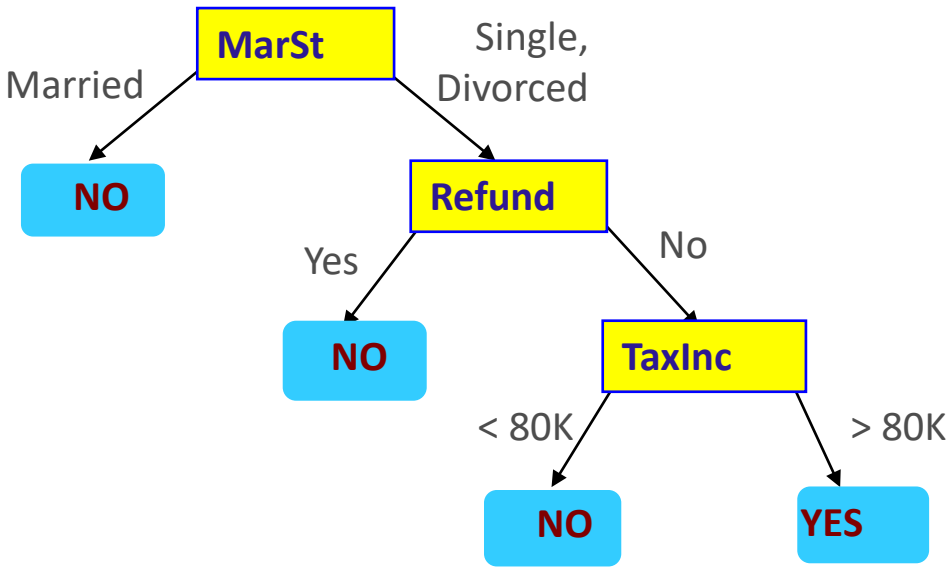
Training Data




Model: Decision Tree

# Another Example of Decision Tree

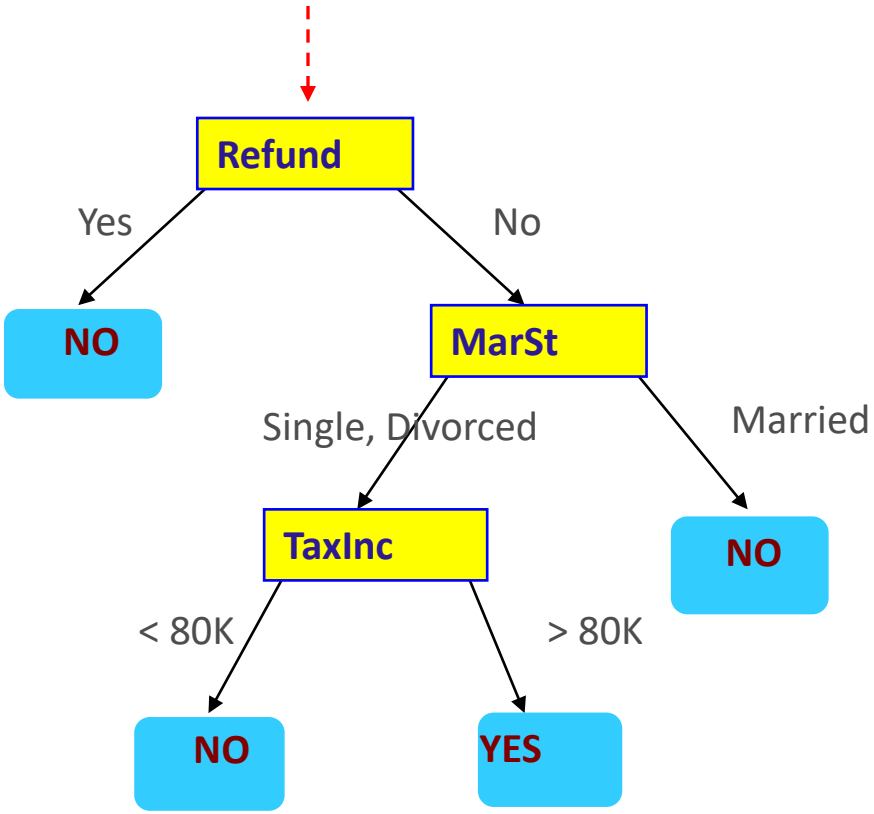
| | categorical | categorical | continuous | class |
|---|---|---|---|---|
| *Tid* | **Refund** | **Marital Status** | **Taxable Income** | **Cheat** |
| 1 | Yes | Single | 125K | No |
| 2 | No | Married | 100K | No |
| 3 | No | Single | 70K | No |
| 4 | Yes | Married | 120K | No |
| 5 | No | Divorced | 95K | Yes |
| 6 | No | Married | 60K | No |
| 7 | Yes | Divorced | 220K | No |
| 8 | No | Single | 85K | Yes |
| 9 | No | Married | 75K | No |
| 10 | No | Single | 90K | Yes |




**There could be more than one tree that fits the same data!**

# Apply Model to Test Data

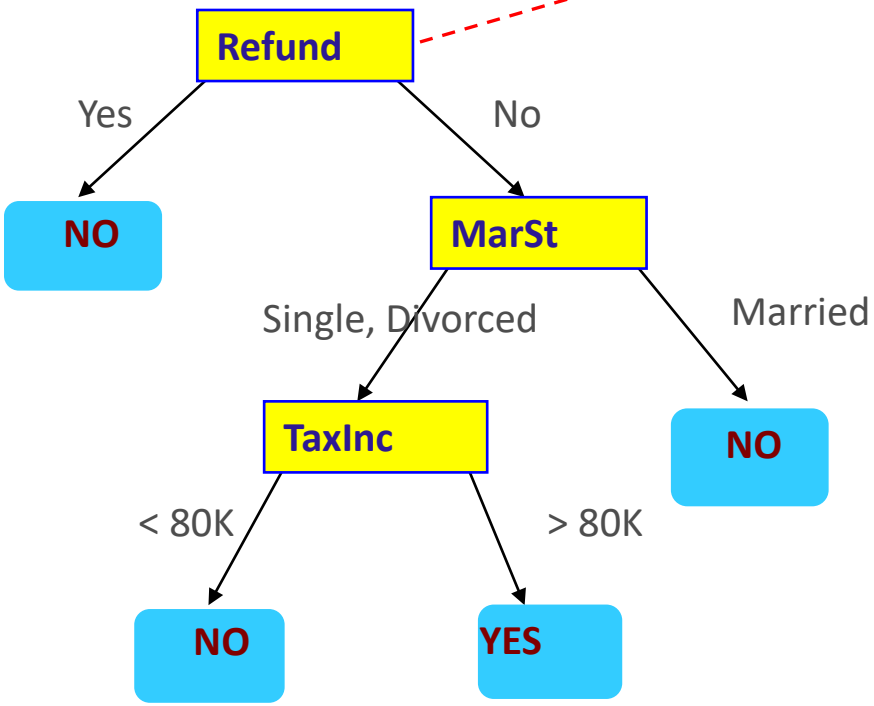
Start from the root of tree.




**Test Data**

| Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|
| No | Married | 80K | ? |

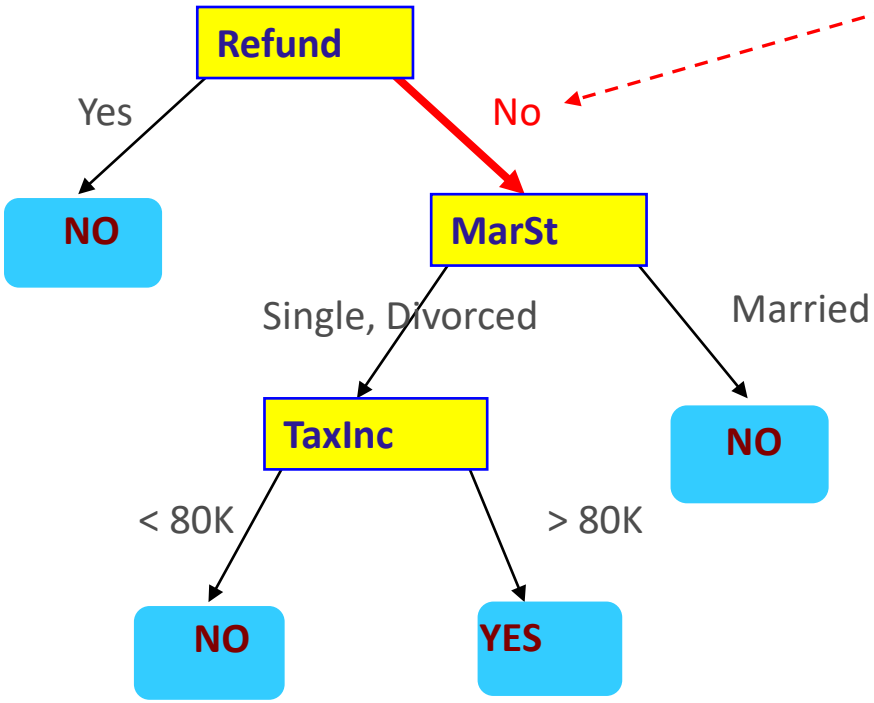
# Apply Model to Test Data




Test Data

| Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|
| No | Married | 80K | ? |

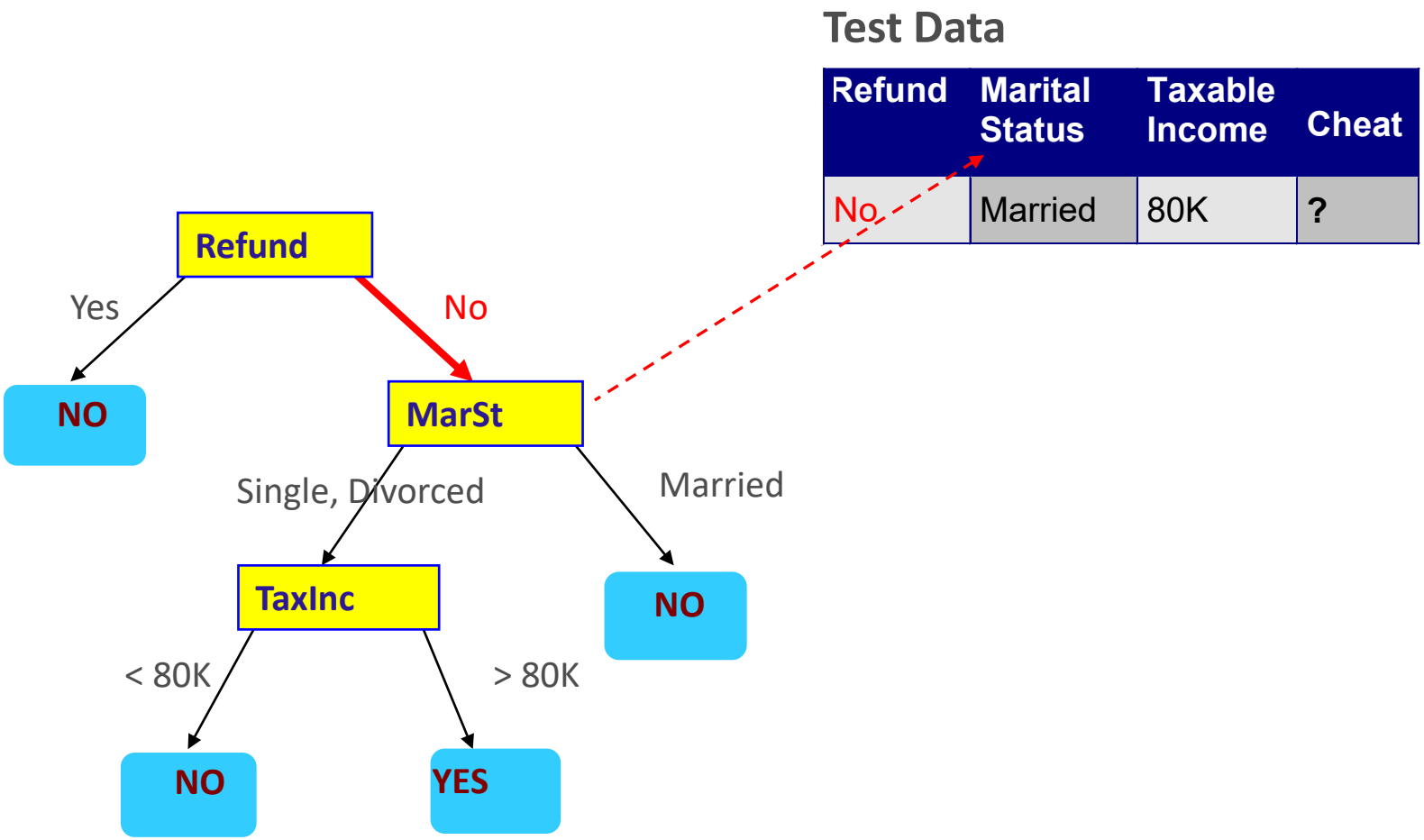
# Apply Model to Test Data




Test Data

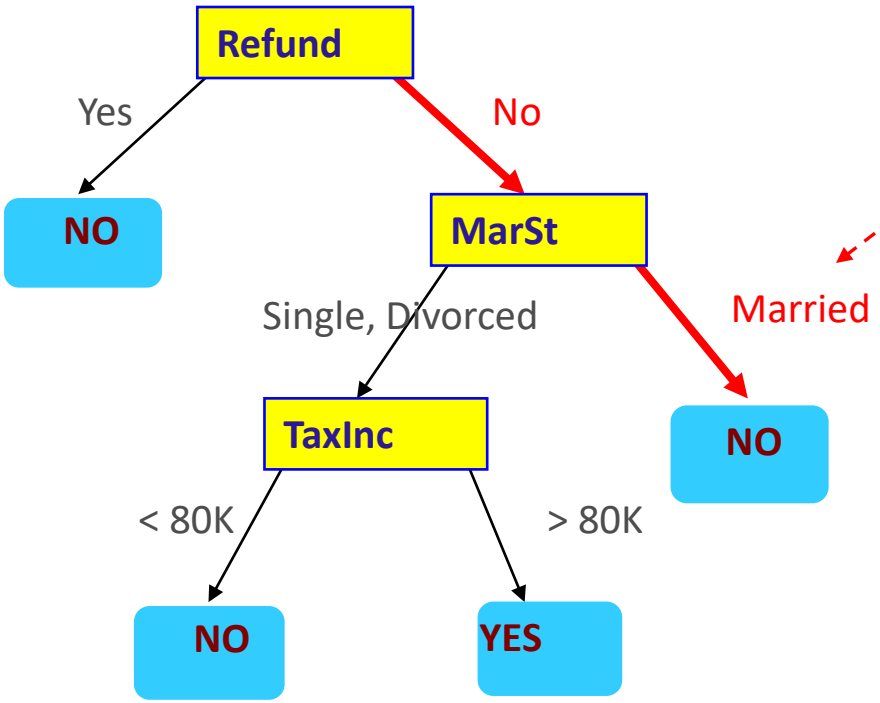
| Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|
| No | Married | 80K | ? |

# Apply Model to Test Data

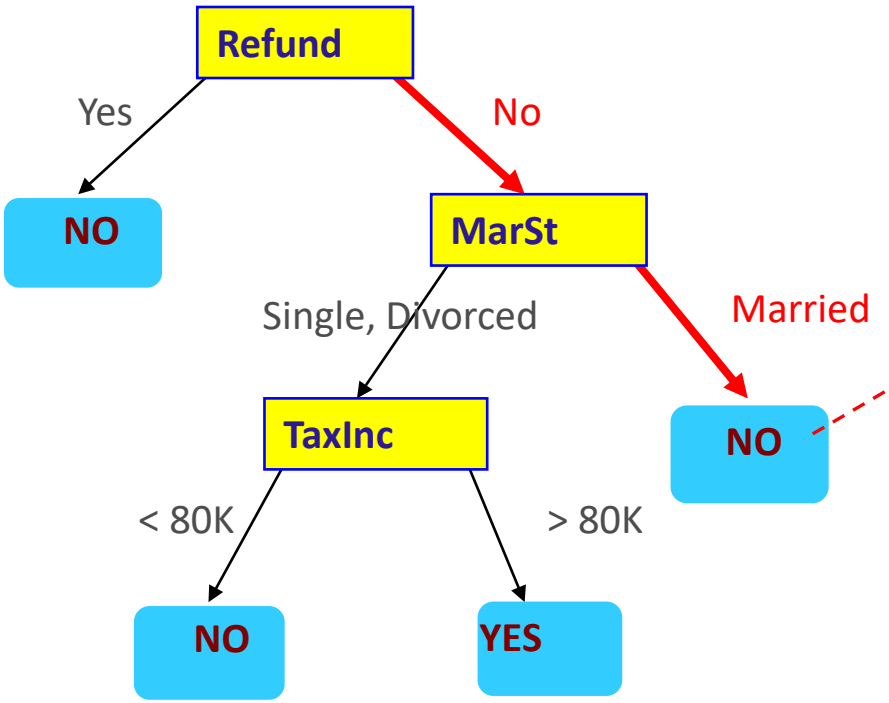


# Apply Model to Test Data

| Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|
| No | Married | 80K | ? |



# Apply Model to Test Data




Test Data

| Refund | Marital Status | Taxable Income | Cheat |
|---|---|---|---|
| No | Married | 80K | ? |

Assign Outcome to “No”

# Entropy Based Evaluation and Splitting

- Entropy at a given ***node t***:

$$Entropy(t) = -\sum_j p(j \mid t) \log p(j \mid t)$$

(NOTE: $p(j \mid t)$ is the relative frequency of class j at node t).

– **Measures impurity of a node:**

- Maximum ($-\log_2 n_c$)
  - when records are equally distributed among all classes: implying least information, where $n_c$ =the number of classes.
- Minimum (0):
  - when all records belong to one class, implying most information

# Examples for computing Entropy

$$Entropy(t) = -\sum_j p(j \mid t) \log_2 p(j \mid t)$$

| | |
|---|---|
| C1 | **0** |
| C2 | **6** |

**P(C1) = 0/6 = 0  P(C2) = 6/6 = 1**

**Entropy = – 0 log 0 – 1 log 1 = – 0 – 0 = 0**

| | |
|---|---|
| C1 | **1** |
| C2 | **5** |

**P(C1) = 1/6  P(C2) = 5/6**

**Entropy = – (1/6) $\log_2$ (1/6) – (5/6) $\log_2$ (1/6) = 0.65**

| | |
|---|---|
| C1 | **2** |
| C2 | **4** |

**P(C1) = 2/6  P(C2) = 4/6**

**Entropy = – (2/6) $\log_2$ (2/6) – (4/6) $\log_2$ (4/6) = 0.92**

# Splitting based on information gain

- Information Gain:

$$GAIN_{split} = Entropy(p) - \left( \sum_i \frac{n_i}{n} Entropy(i) \right)$$

Parent Node, p;

$n_i$ is number of records in partition i

  - Measures Reduction in Entropy achieved because of the split. Choose the split that achieves most reduction (maximizes GAIN)
  - Disadvantage: Tends to prefer splits that result in large number of partitions, each being small but pure.

# Splitting Based on GINI index

- Information Gain:

$$Gini(t) = 1 - \sum_j p(j \mid t)^2$$

  - Split made on the partition with the smallest Gini index

# Pruning the tree

- in general, trees will overfit the data.
- one approach for managing overfitting is to prune the tree after it is built by penalizing tree complexity (e.g., number of terminal leaves)
- it is also possible to manage tree growth to avoid overfitting by having more stringent stopping criteria for splitting the tree

Further reading: https://www.statmethods.net/advstats/cart.html

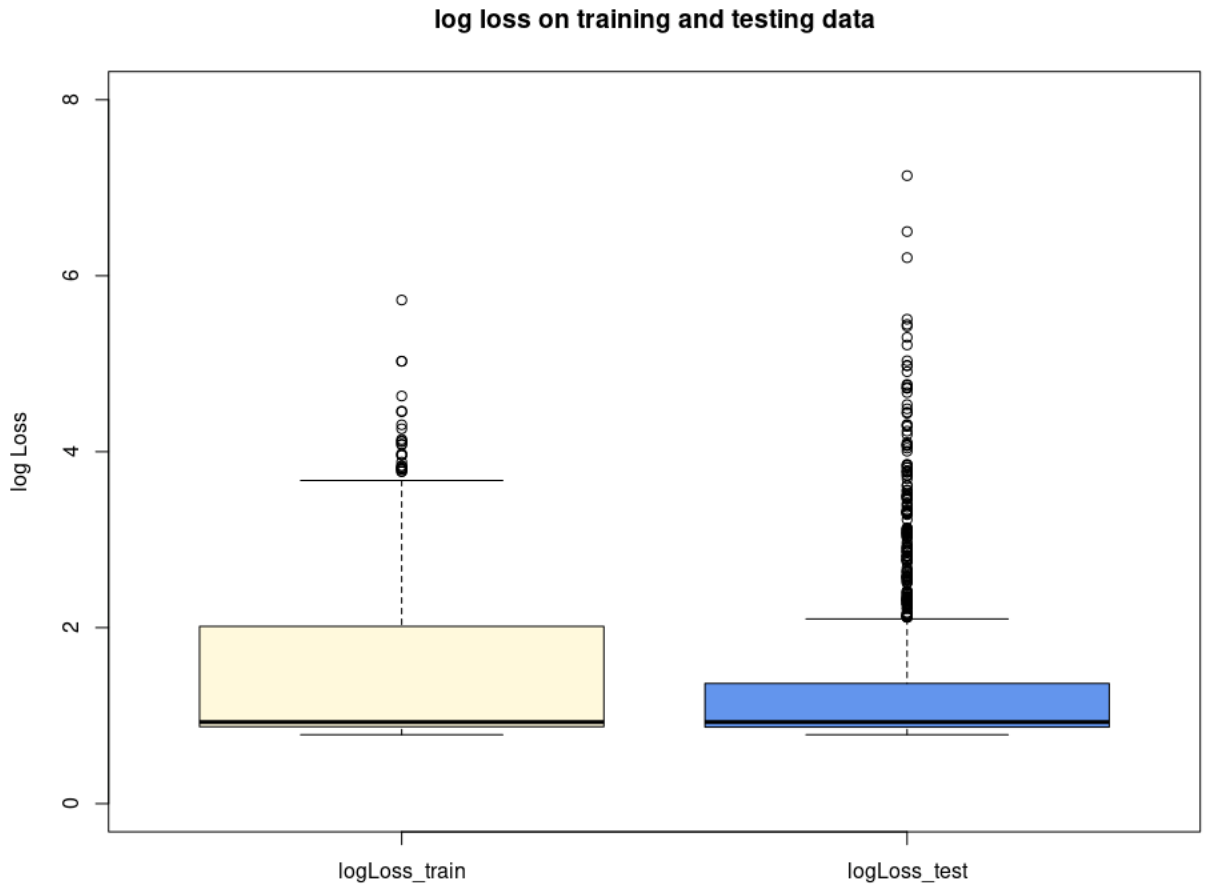
# Decision tree classification (lung)

- recall, our target is to predict (Y/N) if a patient will survive a specified number of days
- we will add a new categorical variable "*alive_at_cutoff*" to indicate 1 for yes and 0 for no for every case when survival time is >= survival_cutoff value:
  - survival >= 1 year ➔ survival_cutoff = 365
  - Survival >= 6 months ➔ survival_cutoff = 365/2
- use 1000 runs of random splitting ➔ train/test
- compare log loss between train and test

# Repeated train/test runs

For associated R code, please refer to notebook:
Lecture 3 – Tree Examples.ipynb

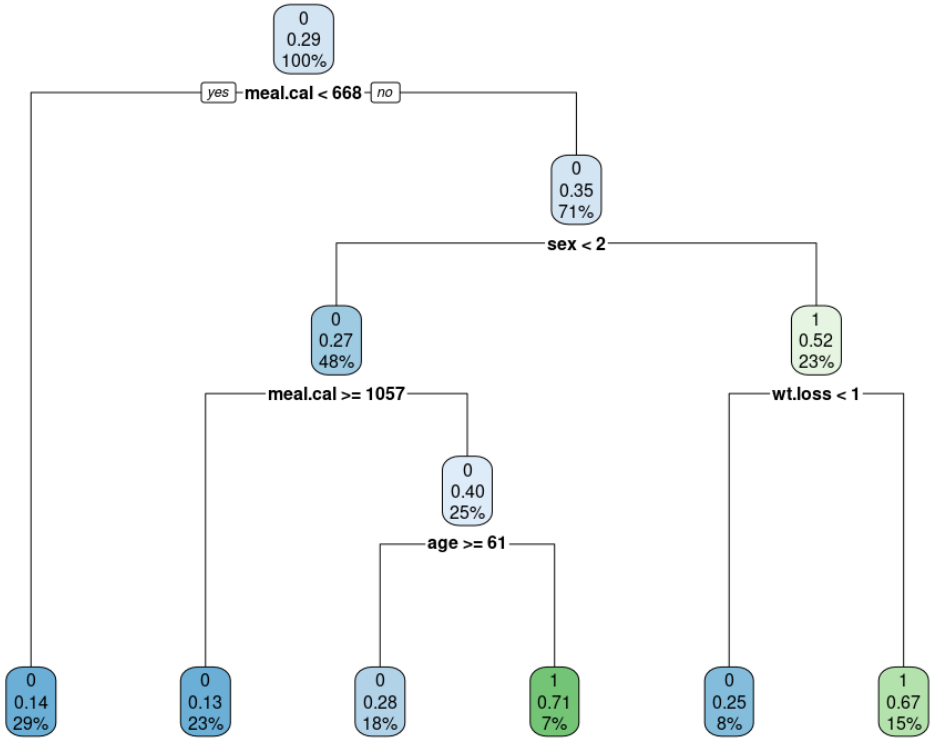
# Log loss of decision tree classification




```
# plot box plots of log los values on training and on testing
boxplot(evaluation_results, ylim=c(0,8), col =
c("cornsilk","cornflowerblue"),
        ylab = "log Loss", main = "log loss on training and testing data")
```

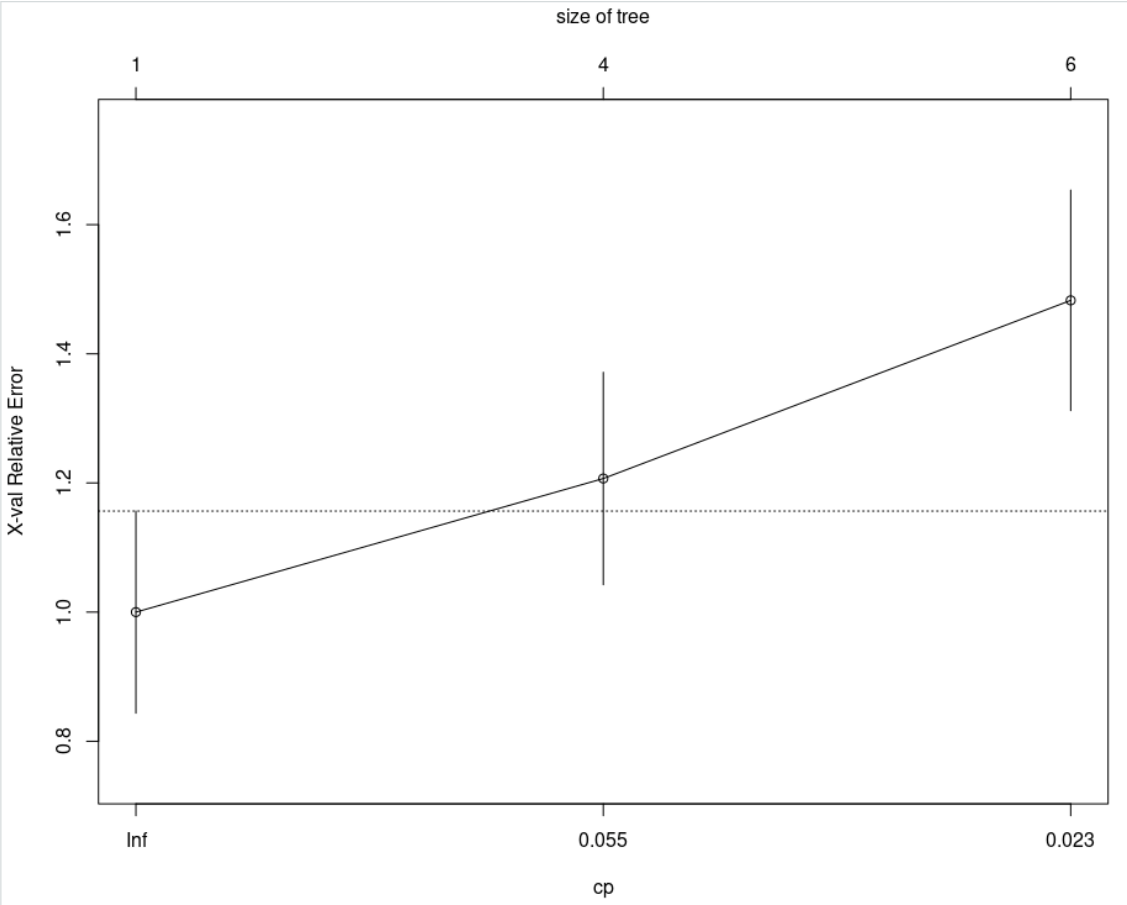
```
rpart.plot::rpart.plot(tree_model). # plot the last decision tree
```

# Resulting decision tree on lung dataset




- recall, the target is to predict 1/0 for alive_at_cutoff >= 365 days
- start at the root: 29% of patients were not alive_at_cutoff. Then, is meal.cal < 668?
- If yes, then we move to left child (a terminal node tells us the prediction is 0) and probability of survival is 14%
- If no, we move to the right child of the root where probability of survival is 35% and 71% of the training data have meal.cal >= 668.

# Complexity parameter (CP)

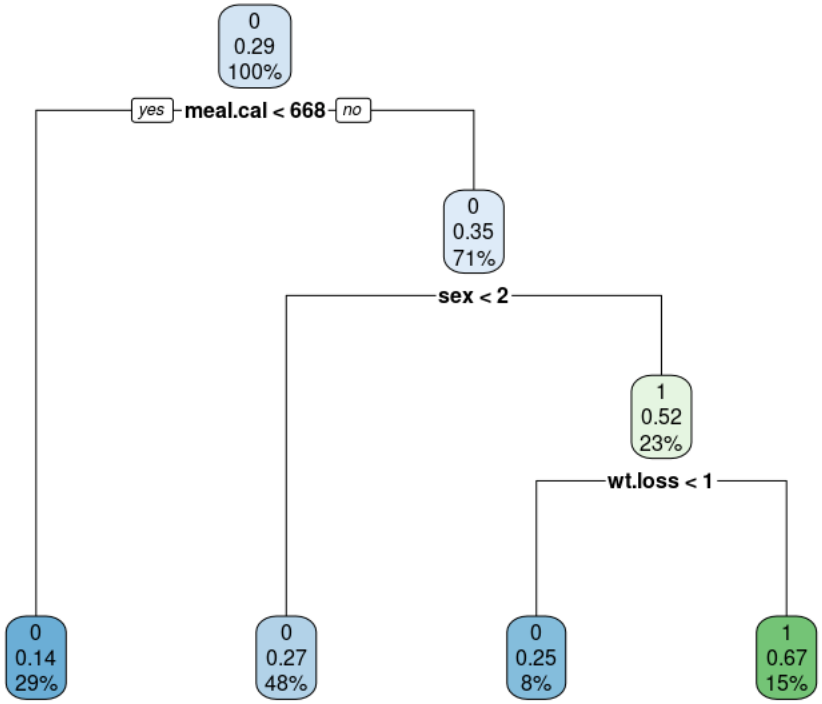



- use it to prune the tree - If the cost of adding another variable to the decision tree from the current node is above the value of cp, then tree building does not continue.
- select the optimal cp value associated with minimum error
- lower cp value result in bigger trees
- lower cp can lead to overfitting

```
> plotcp(tree_model) # plot complexity graph
```

# Pruned decision tree

## (CP= 0.052 for illustration)




```
# examine the values in cptable
printcp(tree_model)

Classification tree:
rpart::rpart(formula = alive_at_cutoff ~ age + sex + meal.cal + wt.loss,
    data = train, method = "class")
Variables actually used in tree construction:
[1] age meal.cal sex wt.loss

Root node error: 29/100 = 0.29
n= 100

       CP  Nsplit rel error xerror    xstd
1 0.057471      0   1.00000 1.0000 0.15647
2 0.051724      3   0.82759 1.2069 0.16447
3 0.010000      5   0.72414 1.4828 0.17072

pruned_tree_model <- prune(tree_model,0.052) # prune the tree at cp = 0.052
```

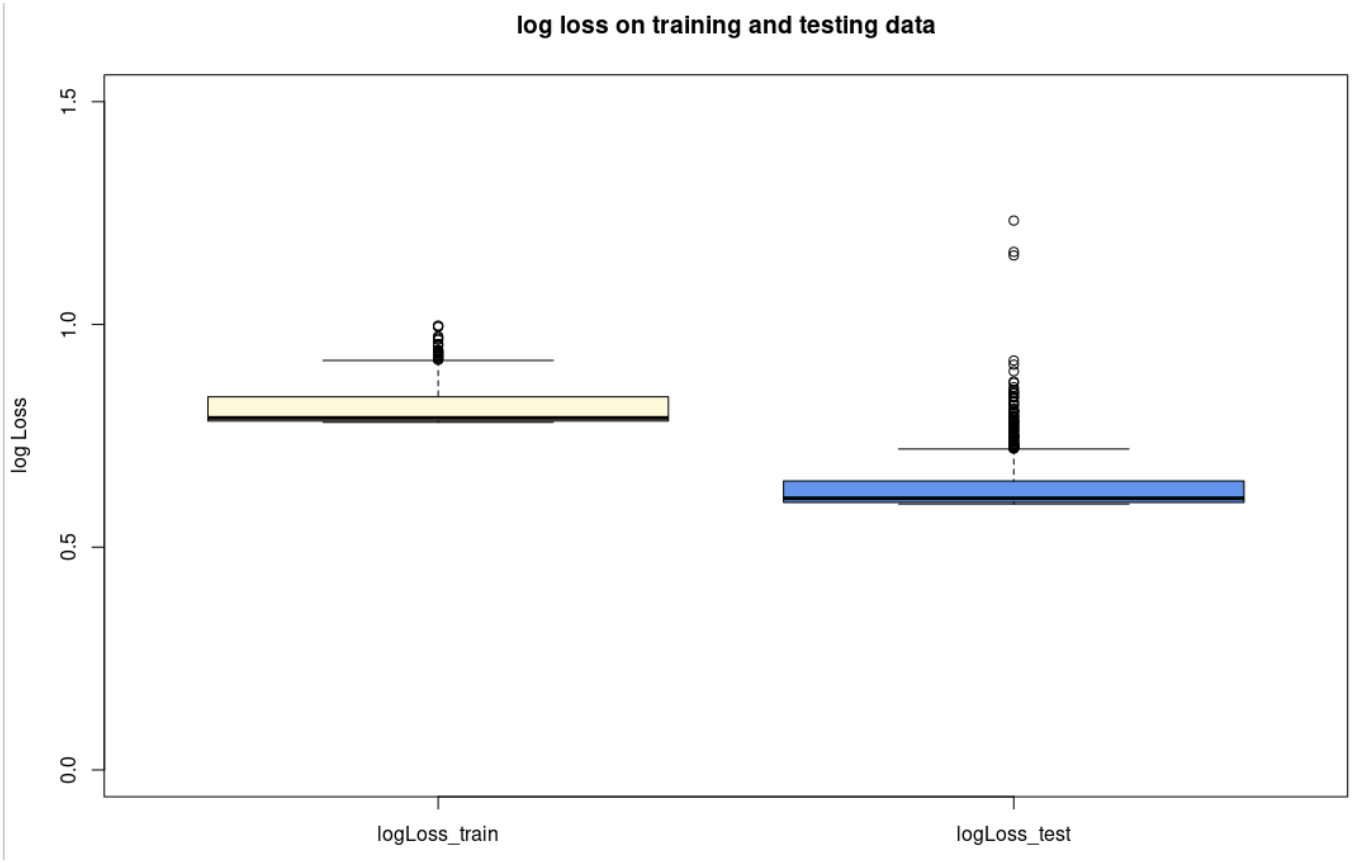
# Repeated runs on pruned tree

```
set.seed(17) # reset the random seed for reproducibility
for (i in 1:iterations) {
  # partition clung into stratified train and test
  inds <- splitTools::partition(clung$alive_at_cutoff,
          p = c(train = train_data_split, test = (1 - train_data_split)))
  train <- clung[inds$train, ] # retrieve train data
  test  <- clung[inds$test, ]  # retrieve test data

  # construct a decision tree classification model with prune cp = 0.052
  pruned_tree_model <- rpart::rpart(alive_at_cutoff ~ age + sex + meal.cal +
                       wt.loss, data = train, method = "class", cp = 0.052)
  # make predictions on training data
  pruned_tree_predictions <- predict(pruned_tree_model,train, type= 'prob')
  # calculate the log loss on training data (find goodness of fit)
  r1 <- MLmetrics::LogLoss(pruned_tree_predictions[,],
                  as.numeric(as.character(train$alive_at_cutoff)))

 # predict the probabilities on test data
  pruned_tree_predictions <- predict(pruned_tree_model,test, type = 'prob')
  # calculate log loss for test data
  r2 <- MLmetrics::LogLoss(pruned_tree_predictions[,2],
                           as.numeric(as.character(test$alive_at_cutoff)))
  # record the goodness of fit on training and logloss on test
  evaluation_results[i,] <- rbind(r1,r2)
} # repeat
```

# Log loss of pruned decision tree



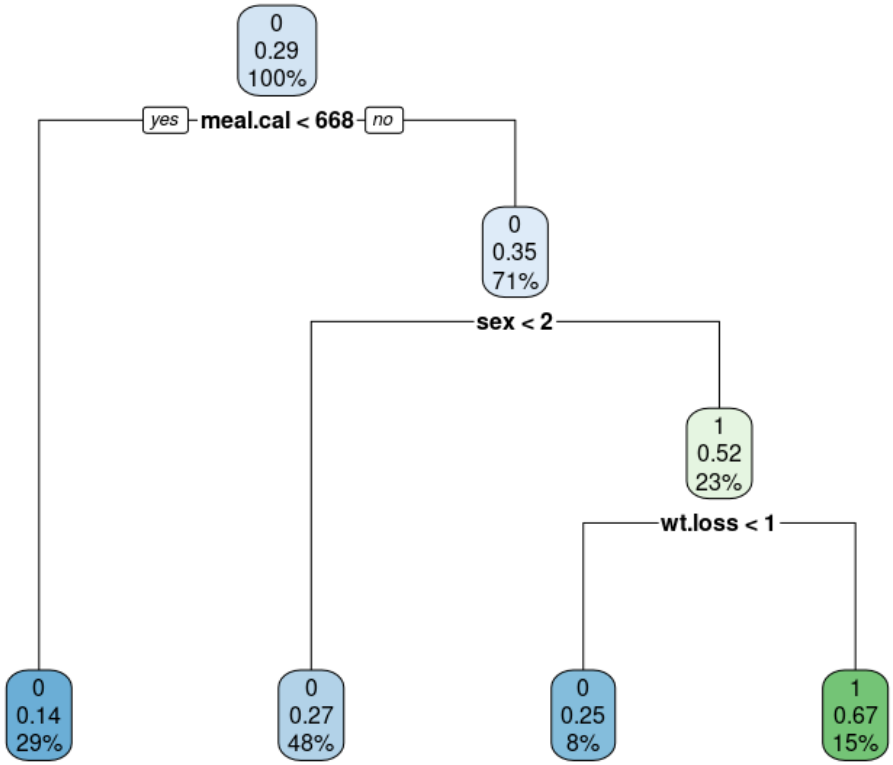

- Log loss is much lower than that obtained from the unpruned tree!

```
# plot box plots of log loss values on training and on testing
boxplot(evaluation_results, ylim=c(0,1.5),
        col =c("cornsilk","cornflowerblue"),
        ylab = "log Loss", main = "log loss on training and testing data")
```

# Converting decision trees to rules

```
> rpart.rules(pruned_tree_model, cover = TRUE)
 alive_at_cutoff                                                  cover
            0.14 when meal.cal <  668                              29%
            0.25 when meal.cal >= 668 & sex >= 2 & wt.loss <  1     8%
            0.27 when meal.cal >= 668 & sex <  2                   48%
            0.67 when meal.cal >= 668 & sex >= 2 & wt.loss >= 1    15%
>
```

- Trees can be converted into rules by following the splits form the root to a leaf node
- Rules are commonly (and intuitively evaluated by:
  - Support: how often a given rule appears in the data
  - Confidence: the number of times a given rule turns out to be true in practice.



# Common metrics to evaluate rules

- an association rule presents a relationship of the form $X \rightarrow Y$

$$Support(X) = \frac{frequency(X)}{n}$$

i.e., how often $X$ is observed, and $n$ is the number of points.

$$Confidence(X \rightarrow Y) = \frac{Support(X \cup Y)}{Support(X)}$$

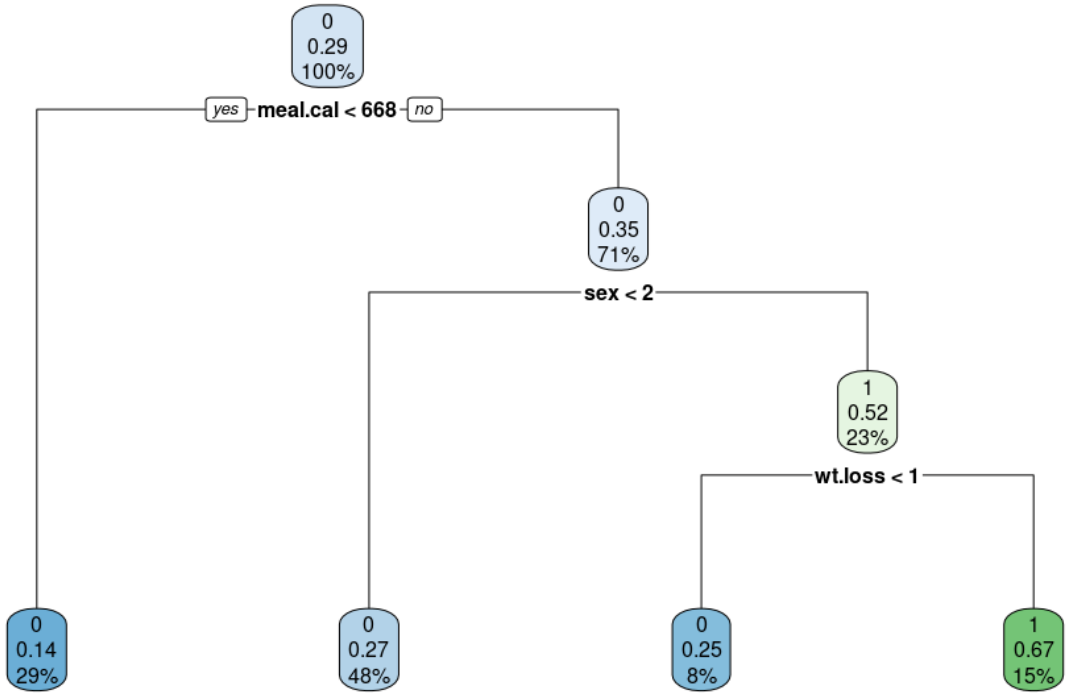
# Hyperparameters

- The tree building process has a number of hyperparameters that need to be defined a priori which will also have an impact on the accuracy of the tree
- These include things like maximum tree depth, minimum number of observations in a terminal node, and splitting criterion

# Model tuning

- is done automatically using default settings
- but it can be set manually

```
tree_model <- rpart(alive_at_cutoff ~ age + sex + meal.cal + wt.loss,
                    data = train, method = "class",
                    control = c(maxdepth = 3, cp=0.001))
```



# Hyperparameter tuning CART model

- Define the function that builds the tree and computes logloss, in which:
  - build tree function
  - predict the probabilities on test data
  - calculate log loss for validation data
- Split the Dataset (train, test, validate)
- Perform grid search hyperparameter tuning
  - define the hyperparameter grid
  - apply the tree function to all combinations
  - select the best parameter settings (min logLoss)
  - Show results on test data

`See notebook Lecture 3 - tree_hyper.ipynb`