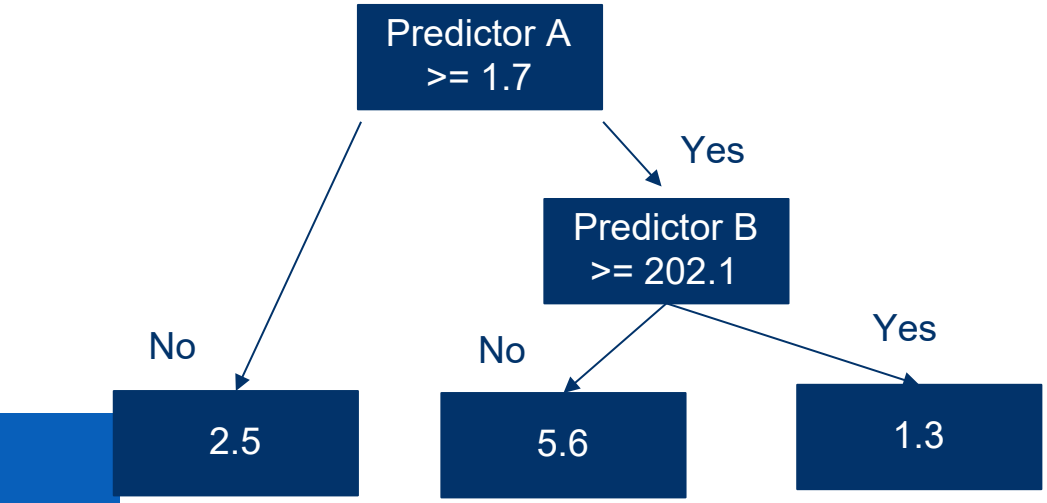
Tree -Based Models

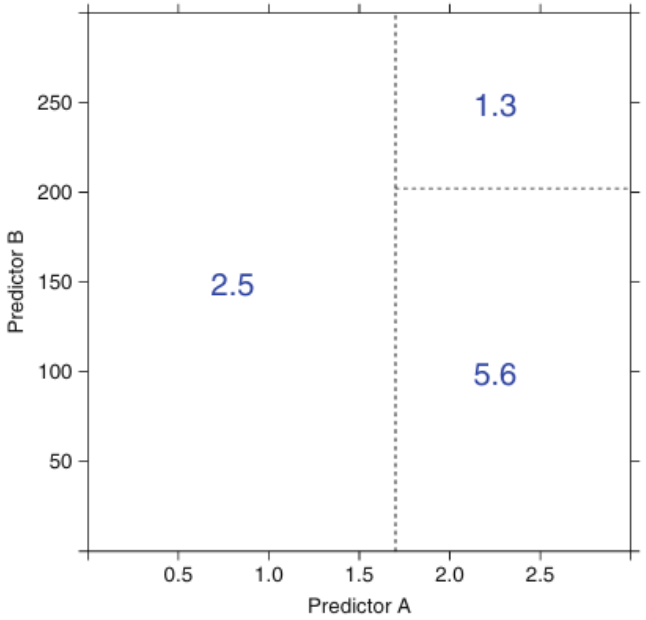
Brandon Chung, Chi Hang (Philip), Jiaxin Zheng, Ron Balaban, Yanyi Li

Basic Structure

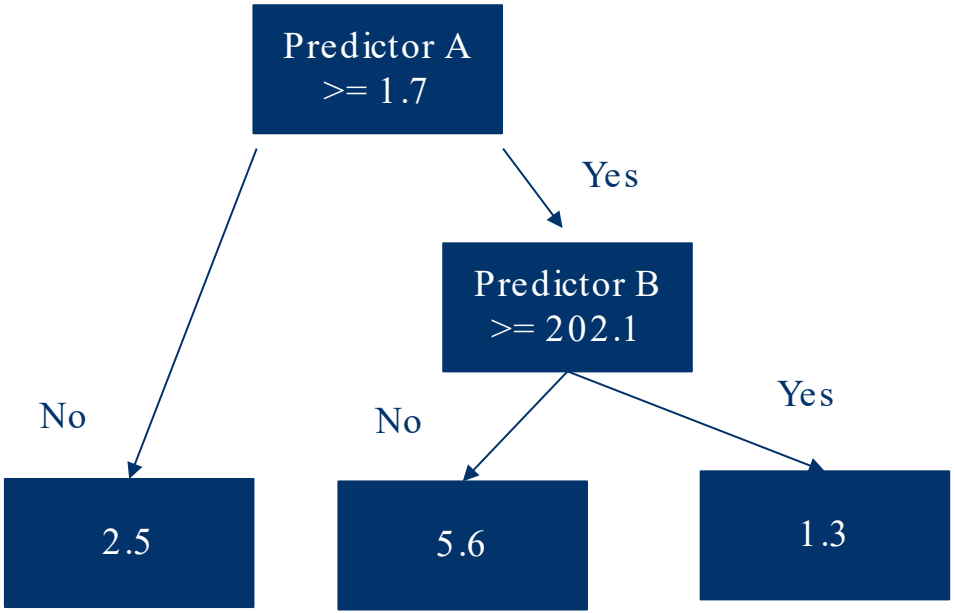
- One or more nested if-then statements for predictors that partition the data

```
if Predictor A >= 1.7 then
|   if Predictor B >= 202.1 then Outcome = 1.3
|   else Outcome = 5.6
else Outcome = 2.5
```





Terminology



- Nodes = Decision Points
- Root Node = First decision/ First node
- Terminal Nodes (leaves) = Outcomes
  - Can be function of predictors
- Branches = Segments of the trees that connect the nodes
- Split = Area in a tree where one node leads to more than one branch

Single Tree Model Pros/Cons

Pros:

- Highly interpretable and easy to implement
- Handle many types of predictors (sparse, skewed, continuous, categorical, etc.) without preprocessing
- Model can handle missing data - conduct feature selection implicitly

Cons:

- Model Instability (slight changes in data drastically changes the structure of the tree or rules)
- Less-than-optimal predictive performance (rectangular regions of outcome values)

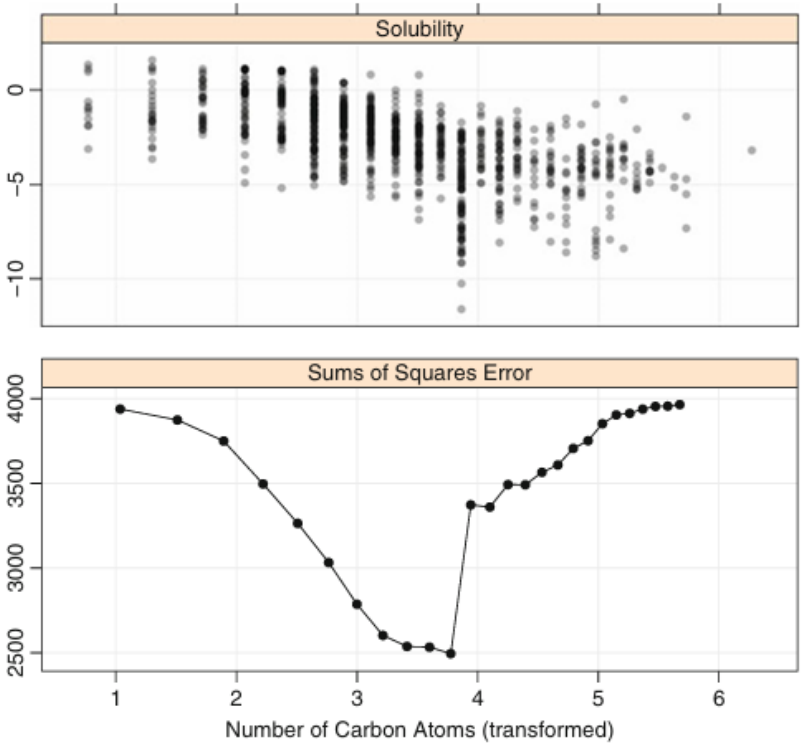
- Cons addressed by ensemble methods that combine many trees

Constructing (Growing the tree)

Classification and Regression Tree (CART)

- Start with the entire data set, S, and searches every distinct value of every predictor to find the predictor and value where the overall sums of squares error is minimized to partition the data into two groups (S1 and S2)
- When done multiple times this is known as "Recursive Partitioning"
- Recursive partitioning typically continues until a threshold, such as 20 partitions

$$\text{SSE} = \sum_{i \in S_1} (y_i - \bar{y}_1)^2 + \sum_{i \in S_2} (y_i - \bar{y}_2)^2,$$



Pruning

- Cost-complexity tuning - find the “right-sized tree” with the smallest error rate.
- Penalize error rate using size of the tree:

$$\mathrm{SSE}_{c_p} = \mathrm{SSE} + c_p \times (\#\ \text{Terminal Nodes}),$$

Cp is complexity parameter - used to penalize the SSE, smaller values produce more complex models, larger values produce less complex models.

- To find the best pruned tree:
    - Evaluate the SSE values for a sequence of Cp values -> generates multiple SSE values that we can apply a cross validation or one-standard-error rule on.
        - One-Standard-Error Rule = Find the smallest tree that is within one standard error of the tree with smallest absolute error.

Pruning (cont.)

- Alternatively to the one -standard-error rule:
  - Choose the value of the Cp associated with the smallest possible RMSE value from the cross-validation profile.

Regression Model Trees

Combines decision tree with linear regression at the nodes. This allows for better predictive accuracy as the outcome is more dynamic and able to capture extremes better.

Splits similarly done, but are created using the expected reduction the node's error rate:
This metric determines if the total variation in the splits, weighted by sample size is lower than in presplit data

The split associated with the greatest reduction is chosen, and linear models are created within the partitions.

$$\text{reduction} = \text{SD}(S) - \sum_{i=1}^{P} \frac{n_i}{n} \times \text{SD}(S_i),$$

Regression Model Trees (Cont.)

- Smoothing to prevent overfitting: “Recursive Shrinking”
  - When producing an output, starting at the appropriate terminal node, the linear regression used is a combination of all the regression models passed on the path to the terminal node.

- Pruning: Starting at the terminal nodes, adjusted error rates with and without the sub-tree is computed, and if the sub-tree does not decrease the adjusted error rate it is printed from the model.

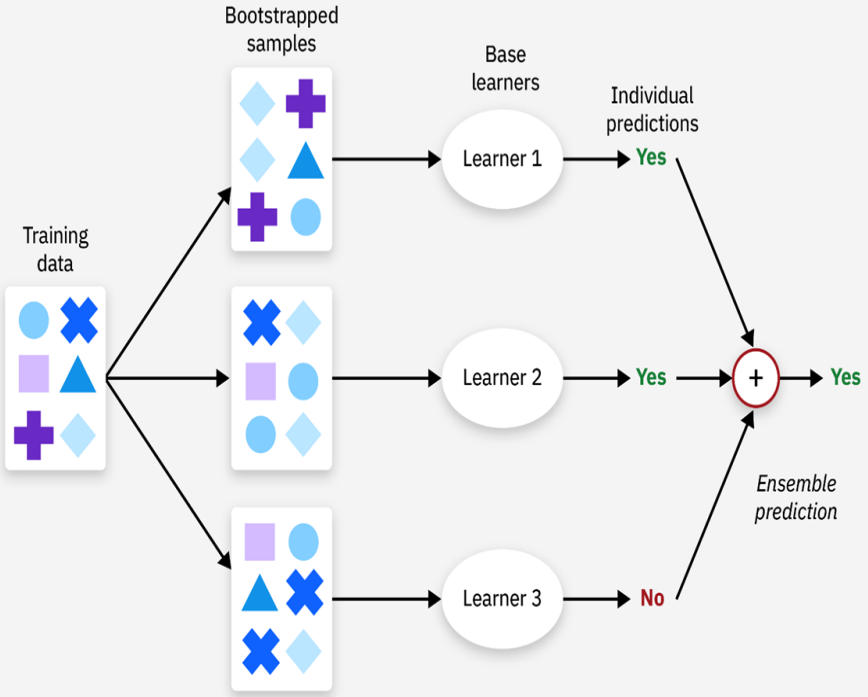
Bootstrap Aggregating (Bagging)

- Use Bootstrapping to create N number subset of training data with **replacement**
- **Parallel** model training
- Creates an ensemble by averaging all the models predictions to reduce overfitting
- Advanced version = randomForest

Advantages:

1) Reduces variance

2) Reduce overfitting



Bootstrapping

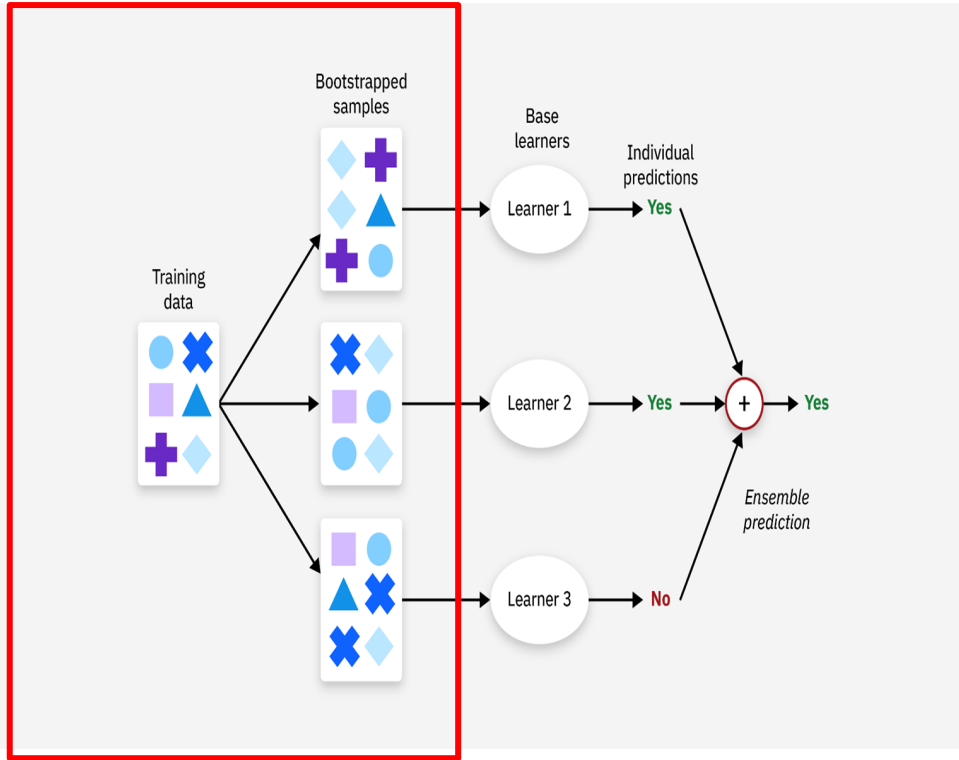
Bootstrap Sampling (Bootstrapping)

- Create training data from original data using a random sampling **with** replacement.

- Each random selection could occur any number of times.

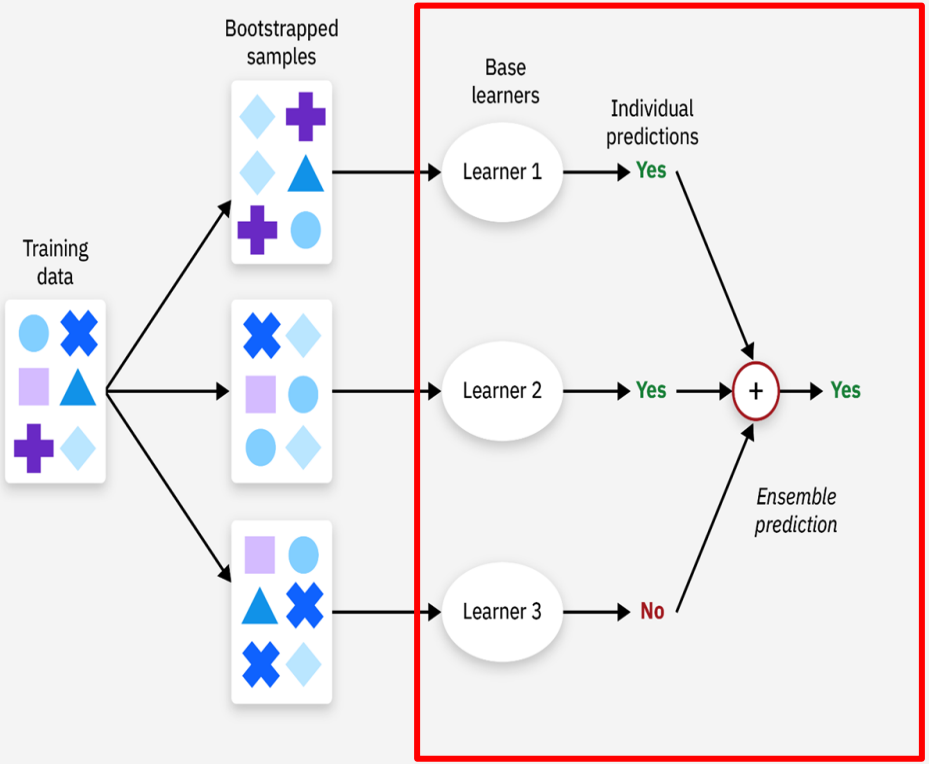
- New dataset used for training, while the original data, un-selected for training, is used for CV.

- Compared to CV, this method is faster/simpler, and is better at estimating model performance for small datasets.



Aggregating

- Each bag is used to train a separate model
- Each model is trained in parallel
- All the prediction results are averaged to become the final prediction



Bagging

Coding Example:

Data source:
ChemicalManufacturingProcess

```
library(ipred)

bagged_Tree<- ipredbagg(

chem_train_y,

chem_train_x, nbagg=25)
```

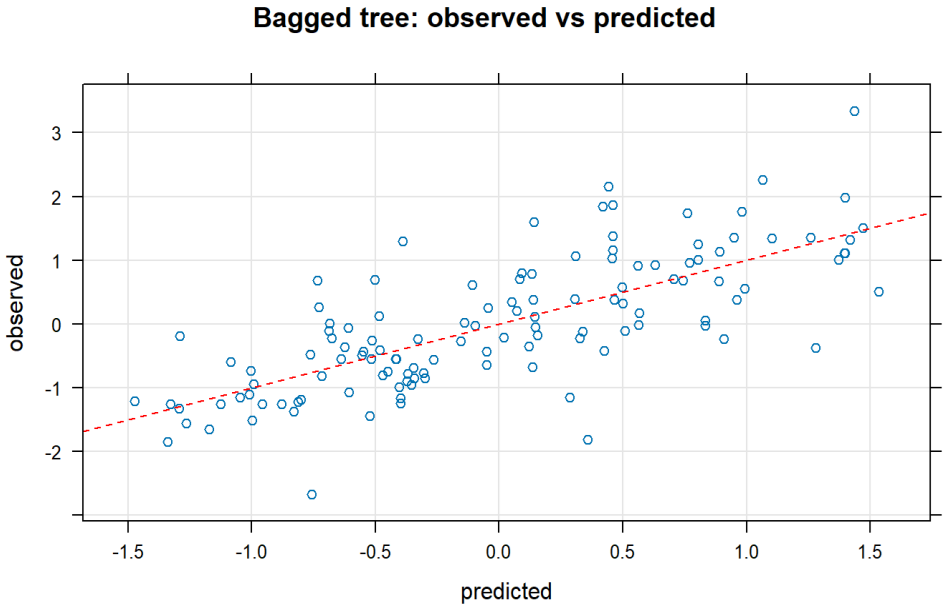
Indices for rows that are selected

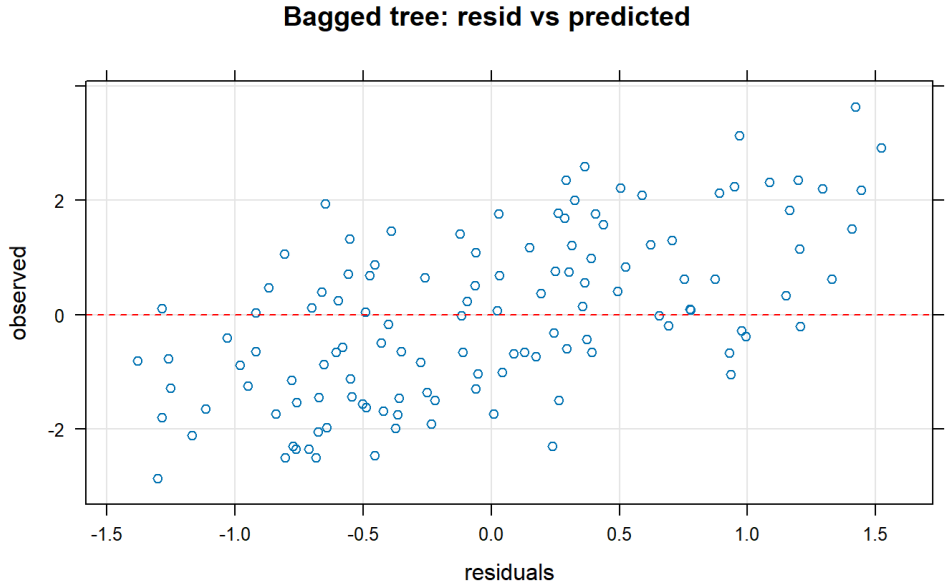
Splitting parameters for Tree #1

```
Bagging regression trees with 25 bootstrap replications

[[1]]
$bindx
  [1] 103  27  47  53 114  41  31 108  47  42  54  18  68  19 106  76  38  42  57  96   2  48
 [23]  22  54  12 102  73   6  49 102  70  97  27  37 106  65  78 120 107  72  87 106  93  21
 [45]  55  89 109  46  28  58  52  47  50  54  90  30  88 113  72  62 101  73  39 118   5  39
 [67]  68  28  96 104 102  75  50  80  88  72  55   7 113   4  99 106  38  99  10  60  17 124
 [89]  18  69  70  60  23 113  53  28  30  39  29 105  49  94  73  87 122  25  74  75  27  64
[111]  82   7  50  66  10  77  37  94  38  53  26   5  41  99

$btree
n= 124

node), split, n, deviance, yval
      * denotes terminal node

 1) root 124 133.6156000  0.03329409
   2) ManufacturingProcess32< 0.191596 73  38.1907600 -0.52627530
     4) BiologicalMaterial03< -0.9122573 23   5.6732080 -1.20659300
       8) BiologicalMaterial11>=-0.8231939 13   0.7945119 -1.47226100 *
       9) BiologicalMaterial11< -0.8231939 10   2.7683700 -0.86122500 *
     5) BiologicalMaterial03>=-0.9122573 50  16.9756300 -0.21332900
      10) ManufacturingProcess30< 0.6030193 38   6.6029030 -0.43827120
        20) BiologicalMaterial11>=0.4194234 12   0.2476649 -0.78555230 *
        21) BiologicalMaterial11< 0.4194234 26   4.2400280 -0.27798770
          42) BiologicalMaterial11< -0.2755295 16   0.9262610 -0.49083040 *
          43) BiologicalMaterial11>=-0.2755295 10   1.4292040  0.06256055 *
      11) ManufacturingProcess30>=0.6030193 12   2.3612100  0.49898830 *
   3) ManufacturingProcess32>=0.191596 51  39.8494700  0.83424630
     6) ManufacturingProcess28< 0.8683945 39  18.6169900  0.55327340
      12) ManufacturingProcess25>=0.09056566 19   7.6575660  0.11622820 *
      13) ManufacturingProcess25< 0.09056566 20   3.8825600  0.96846640
        26) ManufacturingProcess19>=-0.5638366 8   2.1238570  0.64053610 *
        27) ManufacturingProcess19< -0.5638366 12   0.3248584  1.18708700 *
     7) ManufacturingProcess28>=0.8683945 12   8.1472170  1.74740800 *
```

Prediction diagnostic results:

RMSE	Rsquared	MAE
0.5689900	0.6076770	0.4339393



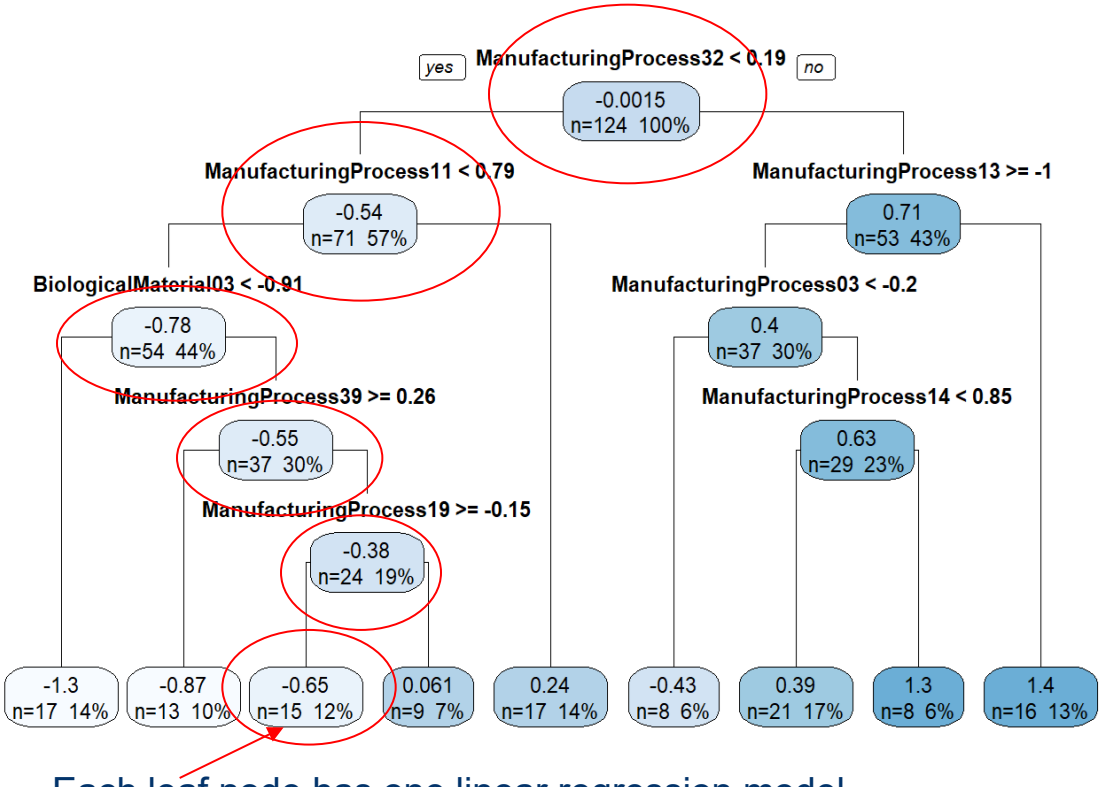


Rule -Based tree model:

The “Separate and Conquer” methodology proposed by Holmes et al. (1993)

- Step 1: Build a tree model
- Step 2: Extract a single rule that has the most **coverage**, which is the number of samples from the training data that follow this path in the tree model.
- Step 3: Remove the training data covered by the rule selected. Repeat step 1-2.
- Step 3: Collect and rank each rule sequentially..
- Step 4: These steps repeat until all training data are covered by at least one rule.
- Step 5: When a new test sample is applied to this model, the linear regression model of the first sequentially collected set of rule that matches the best is used to evaluate the new test sample.

The more advance version of rule-base is Cubist.



Each leaf node has one linear regression model

Rule -Based model

Coding example:

M5Rules from Weka package uses similar approach for tree modeling:

- It builds a decision tree using the “Divide-and-Conquer” methodology to recursively splits the data based on feature conditions (e.g. pressure, temperature)
- It assigns linear regression models at the leaves (terminal points of a tree)
- Extracts the rules from the tree model using the “Separate-and-Conquer” methodology
- When predicting for new data and multiple rules apply to the new data, the one with the lowest estimated error is used.

Rule -Based model

Coding example:

```
m5rules<- train(x= chem_train_x,
                y= chem_train_y,
                method = 'M5Rules',
                trControl = trainControl(method='cv'),
                control = Weka_control(M=10))
```

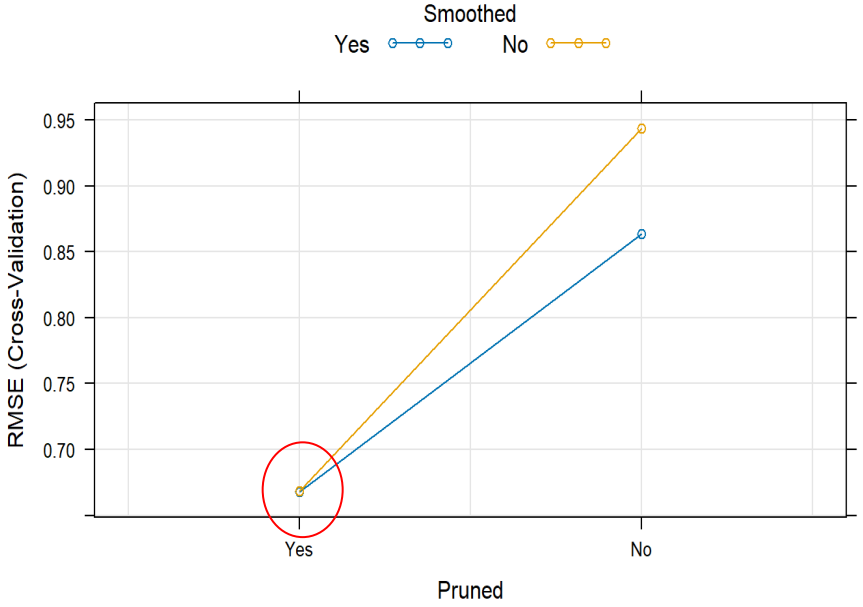
*The M parameter controls the minimal number of instances(data points) needed to reach a node in order for it to keep splitting. If the number of instances do not meet the M value, the node becomes a leaf node (terminal node), stopping to split further.

Small M will make the mode too specific–overfitting (too many splittings)

Large M will make the mode too general–underfitting (too many leaf nodes)

Rule -Based model:

Model Selection




```
Model Rules

124 samples
 57 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 112, 112, 110, 112, 112, 112, ...
Resampling results across tuning parameters:

  pruned  smoothed  RMSE       Rsquared   MAE
  Yes     Yes       0.6678171  0.5881014  0.5379196
  Yes     No        0.6681735  0.5877197  0.5375549
  No      Yes       0.8633989  0.3299680  0.6734082
  No      No        0.9440006  0.2833767  0.7129772

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were pruned = Yes and smoothed = Yes.
```

Rule -Based model:

What is inside the model?

- The final model has only one rule with the largest coverage for the new test data.
- The rule is associated with a linear regression equation, which is used to predict the outcome for the response variable:

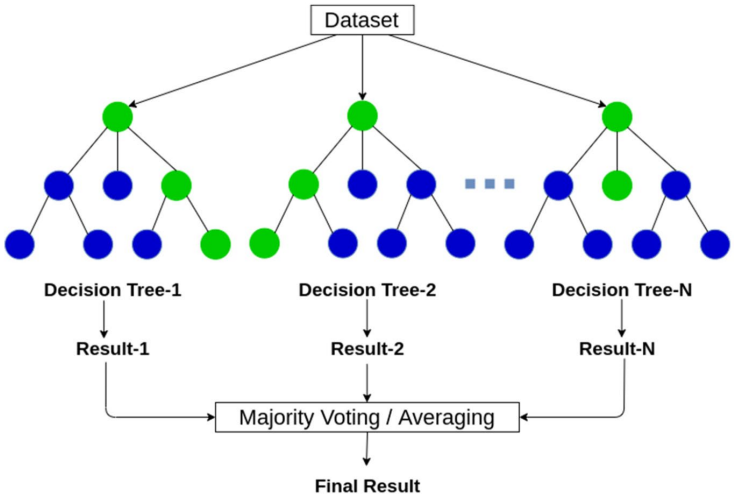
RMSE	Rsquared	MAE
0.6111017	0.5568828	0.4828267

```
M5 pruned model rules
(using smoothed linear models) :
Number of Rules : 1

Rule: 1

.outcome =
        0.1415 * BiologicalMaterial03
        + 0.1048 * ManufacturingProcess11
        - 0.4813 * ManufacturingProcess13
        + 0.2172 * ManufacturingProcess14
        + 0.678 * ManufacturingProcess32
        - 0.2392 * ManufacturingProcess33
        - 0.0365 [124/57.541%]
```

57.5% of the test data is covered by this rule.

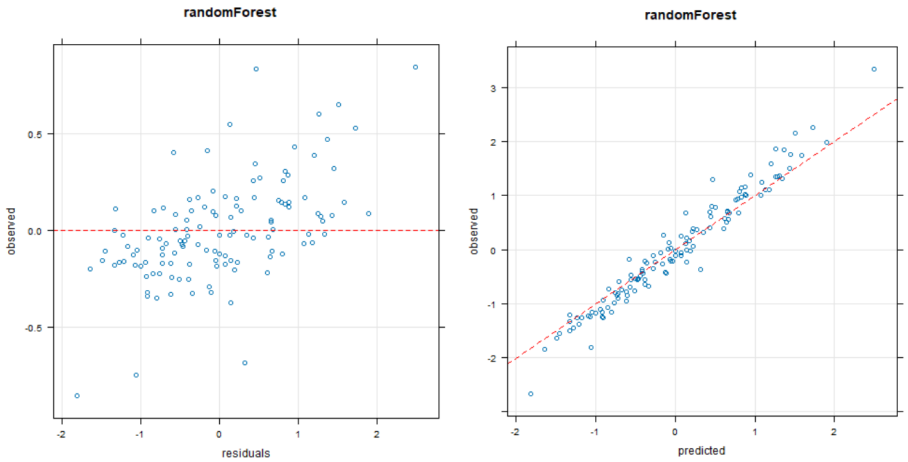
Random Forest

- A **Random Forest** is an ensemble learning method that builds multiple decision trees and combines the output of multiple decision trees to reach a single result.
- Handles both classification and regression problems.



Why Add Randomness?

- Random Forest reduce tree correlation by selecting a random subset of predictors at each split.
- By lowering the correlation, it leads to better variance reduction,and improved accuracy.
- It reduces overfitting and improves predictive accuracy, particularly when the individual trees are uncorrelated with each other.



How Random Forests Work

01

Bootstrap Sampling

Each decision tree in the forest is trained on an unique bootstrap sample of the original data.

One-third of sample will set as test data, known as the out- of- bag(OOB) sample.

02

Random Feature Selection

At each node, a random subset of predictors will be selected.

Among the predictors, RF will determine the ones with the less MSE(or error) as the criterion for a split at that node.

03

Ensemble Prediction

After each tree has been trained, the Random Forest averages the predictions from all the trees to make a final prediction:

- For classification: The final prediction comes from all the trees' majority votes.
- For regression: The final prediction is the average of all tree predictions.

Random Forests Pros/Cons

Pros:

- High predictive accuracy and reduced risk of overfitting and bias compared to a single decision tree.
- Provides flexibility: can handle both regression and classification.
- It maintains accuracy even when a portion of the data is missing.
- Works well with large and complex datasets.

Cons:

- Time- consuming: Can be slow to process data when it is computing data for each individual decision tree.
- More complex when compared to a single decision tree.
- Require more resources to store the data.

Example

- **mtry:** Number of predictors randomly selected at each split.
- **ntree :** Number of trees created. Usually use at least 1000.

```r
library(randomForest)

rf_model<- randomForest(chem_train_x, chem_train_y,importance = TRUE,
                        ntrees= 1000)
#To tune the mtry if wanted:

grid <- expand.grid(mtry = seq(1, ncol(chem_train_x), by = 50))

chem_rf<- train(chem_train_x,
                chem_train_y,
                method = 'rf',
                trControl = trainControl(method='cv', number=10),
                ntree=1000,
                tuneGrid = grid
                )

chem_rf_pred<- predict(chem_rf, chem_test_x)

postResample(chem_rf_pred, chem_test_y)
```

Cubist

- Cubist is a machine learning technique that extends regression trees by incorporating rule-based models and instance-based learning to improve predictive accuracy.
  - **Rule-based:** Create explicit "if-then" rules to make predictions.
  - **Instance-based:** Make predictions based on past examples (instances) rather than general rules.
- It was developed by Ross Quinlan, the same researcher who created C4.5 and M5 model trees.

Key Characteristics

- **Regression Tree with Linear Models**
  - Unlike traditional decision trees that predict a single value per leaf, Cubist builds a regression model at each leaf.
  - Each rule (or tree split) corresponds to a linear equation instead of a constant value.
- **Rule-Based Refinement**
  - After constructing the regression tree, Cubist extracts rules from it.
  - Each rule consists of conditions (if-then statements) and an associated linear regression model for prediction.

Key Characteristics

- **Instance-Based Correction (Neighbors)**
  - After making an initial prediction, Cubist adjusts it based on similar past cases.
  - This nearest neighbor correction helps reduce errors and improve generalization.
- **Handles Missing Data Well**
  - Uses surrogate splits to deal with missing values.

How the Cubist Works

01

Builds a Regression Tree

Uses recursive partitioning to create splits based on features that minimize error.

At each leaf node, instead of assigning a single value, it fits a linear regression model.

02

Converts the Tree into Rules

Each path from the root to a leaf becomes an if-then rule.

The corresponding linear model is stored for that rule.

03

Applies Nearest Neighbor Adjustments

During prediction, it finds similar instances in the training set.

The final prediction is adjusted based on these neighbors.

Strengths and Weaknesses

- **Pros:**
  - Interpretability (clear rules).
  - Good performance with small-to-medium datasets.
  - Robust to irrelevant features.
- **Cons:**
  - Not ideal for very high-dimensional data (like deep learning).
  - Limited to regression (no native classification support).

When to Use Cubist?

- **Best for:**
  - Structured datasets with mixed linear & non-linear relationships.
  - Situations where interpretability is important.
  - Forecasting tasks that benefit from instance-based corrections.

- **Avoid If:**
  - Need a pure deep learning approach.
  - Have very high-dimensional sparse data (e.g., text mining).
  - Want a model that is natively available in Python (Cubist is mainly in R).

Example

- The **Cubist** package in **R** directly implements the Cubist algorithm, making it easy to use.
- Python does not have a direct Cubist package.

```
# Load dataset (using Boston Housing data)
data <- MASS::Boston

# Define predictors and target variable
predictors <- data[, -14]  # All columns except median house price
target <- data$medv        # House price

# Train a Cubist model
cubist_model <- cubist(x = predictors, y = target, committees = 10)
# Using 10 committees (boosting)

# Make predictions
predictions <- predict(cubist_model, predictors)

# Print model summary
summary(cubist_model)
```

Estimating Future Model Performance

- Goal: Use observed data to develop a model that best estimates the relationship between a set of predictor variables X and corresponding response values Y.
- Relationship between X and Y is ***goodness-of-fit***.
- The difference between a model's predicted response and observed response is ***resubstitution error.***

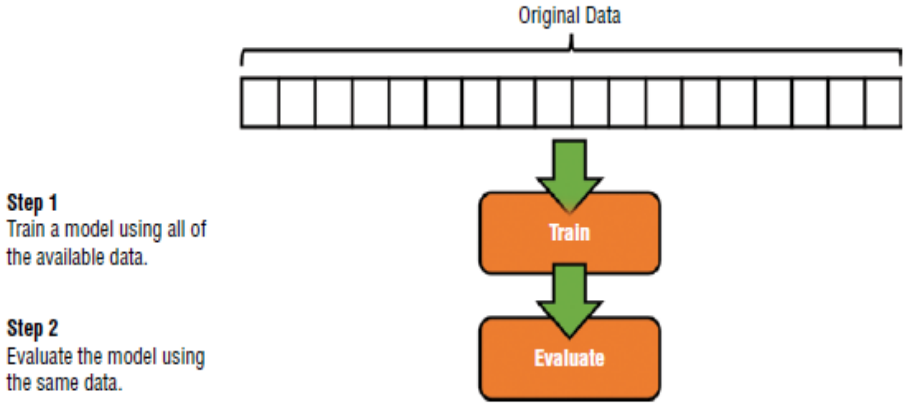


Figure 9.1 Model build and evaluation process using all of the observed data

Holdout Method: Train-Test Split Partitions (1/3 to 1/4)

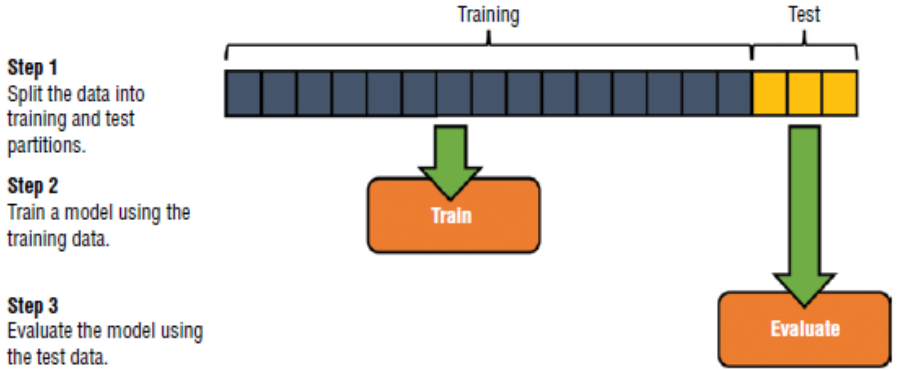


Figure 9.2 Model build and evaluation process using subsets of the observed data for training and for test (the holdout method)

Estimating Future Model Performance

- Partition the Training/Validation/Test sets as 50/25/25% respectively.
- With small enough dataset, the random sampling approach used to generate training, validation, and test partitions could result in samples that don't evenly represent class distribution of original dataset.

Step 1
Split the data into training, validation, and test partitions.

Step 2
Train and tune a model using the training and validation data.

Step 3
Evaluate the final model using the test data.

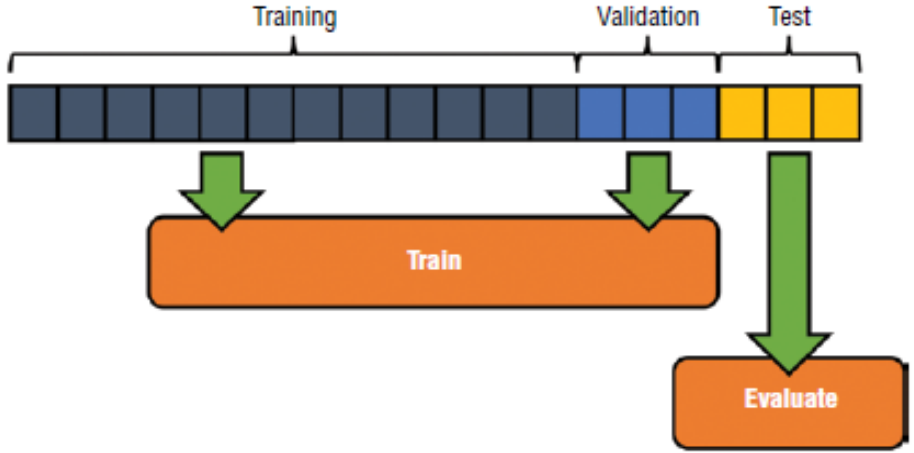


Figure 9.3 Model build and evaluation process using the training and validation data to optimize and choose a model. The test data is used to estimate the future performance of the final model.

k-Fold CV

Repeatedly using different samples of the original data to train and validate a model. The performance of the model across the different iterations is averaged to yield an overall performance estimate for the model.

k-Fold Cross-Validation

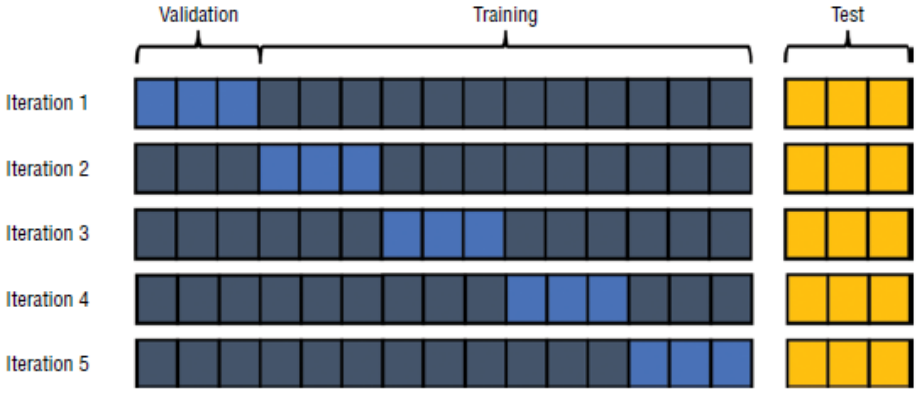


Figure 9.4 The *k*-fold cross-validation approach with *k*=5 (5-fold cross validation). A set of *n* examples is split into five independent folds.

- After the TTS, partition the training data equally

- For the first iteration, all instances labeled as fold1 are held out, while the remainder of the data is used to train the model. The performance of the model is then evaluated against the unseen data (fold1)

- ***Stratified CV***: Ensure class distribution of each fold is representative of the overall distribution.

Code (K -Fold)

Import & view data

```
> library(tidyverse)
> income <- read_csv("income.csv", col_types = "nffnfffffnff")
> glimpse(income)

Observations: 32,560
Variables: 12
$ age                <dbl> 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30, 23, 32,...
$ workClassification <fct> Self-emp-not-inc, Private, Private, Private, Privat...
$ educationLevel     <fct> Bachelors, HS-grad, 11th, Bachelors, Masters, 9th, ...
```

Train/Test Split partition with caret(). Set seed for reproducibility

```
> library(caret)
> set.seed(1234)
> sample_set <- createDataPartition(y = income$income, p = .75, list = FALSE)
> income_train <- income[sample_set,]
> income_test <- income[-sample_set,]
```

Balance imbalanced dataset with SMOTE()

```
> library(DMwR)
> set.seed(1234)
> income_train <-
    SMOTE(income ~ .,
          data.frame(income_train),
          perc.over = 100,
          perc.under = 200)
```

Train & validate the model. 5-Fold CV

```
> library(rpart)
> set.seed(1234)
> income_mod <- train(
    income ~ .,
    data = income_train,
    metric = "Accuracy",
    method = "rpart",
    trControl = trainControl(method = "cv", number = 5)
  )
```

View Accuracy

```
> income_mod$resample %>%
    arrange(Resample)

   Accuracy     Kappa Resample
1 0.7963868 0.5927808    Fold1
2 0.7861395 0.5722789    Fold2
3 0.7333192 0.4666383    Fold3
4 0.7309245 0.4618247    Fold4
5 0.7774235 0.5548469    Fold5
```

```
> income_mod$resample %>%
    arrange(Resample) %>%
    summarise(AvgAccuracy = mean(Accuracy))
  AvgAccuracy
1   0.7648387
```

LOOCV

Leave-One-Out Cross-Validation (LOOCV)

Figure 9.5 The leave-one-out cross-validation approach (LOOCV). A set of *n* examples with only one instance is used for validation in each iteration.

- Functionally a *k*-Fold model with *k* sets.

- After the last iteration, we end up with *n* estimates of the model's performance from each of the iterations. The average of these estimates is the estimate of model performance.

- Greatest amount of data trained, high accuracy.

- Deterministic: Same performance each run.

- Very high or infeasible computational cost.

Code (LOOCV)

Import & view data

```
> library(tidyverse)
> income <- read_csv("income.csv", col_types = "nffnfffffnff")
> glimpse(income)

Observations: 32,560
Variables: 12
$ age                <dbl> 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30, 23, 32,...
$ workClassification <fct> Self-emp-not-inc, Private, Private, Private, Privat...
$ educationLevel     <fct> Bachelors, HS-grad, 11th, Bachelors, Masters, 9th, ...
```

Train/Test Split partition with caret(). Set seed for reproducibility

```
> library(caret)
> set.seed(1234)
> sample_set <- createDataPartition(y = income$income, p = .75, list = FALSE)
> income_train <- income[sample_set,]
> income_test <- income[-sample_set,]
```

Balance imbalanced dataset with SMOTE()

```
> library(DMwR)
> set.seed(1234)
> income_train <-
   SMOTE(income ~ .,
         data.frame(income_train),
         perc.over = 100,
         perc.under = 200)
```

Train & validate the model. *k*-Fold CV

```
> library(rpart)
> set.seed(1234)
> income_mod <- train(
    income ~ .,
    data = income_train,
    metric = "Accuracy",
    method = "rpart",
    trControl = trainControl(method = "LOOCV")
  )
```

View Accuracy

```
> income_mod$resample %>%
   arrange(Resample)

   Accuracy     Kappa Resample
1 0.7963868 0.5927808    Fold1
2 0.7861395 0.5722789    Fold2
3 0.7333192 0.4666383    Fold3
4 0.7309245 0.4618247    Fold4
5 0.7774235 0.5548469    Fold5
```

```
> income_mod$resample %>%
   arrange(Resample) %>%
   summarise(AvgAccuracy = mean(Accuracy))
  AvgAccuracy
1   0.7648387
```

Random CV

Random (Monte Carlo) Cross-Validation

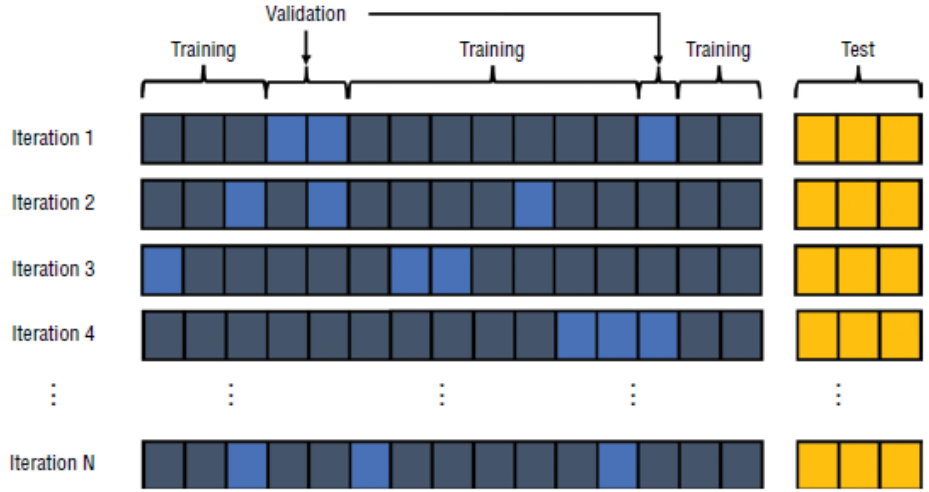


Figure 9.6 The random cross-validation approach. The training and validation sets are created independently in each iteration.

- Instead of creating a set number of folds, the random sample that determines validation is created each iteration.

- Random sampling without replacement

- Each random selection could occur any number of times.

- Major advantage over *k*-CV: size of the training and validation sets is independent of the number of CV iterations.

Code (RCV)

Import & view data

```
> library(tidyverse)
> income <- read_csv("income.csv", col_types = "nffnfffffnff")
> glimpse(income)

Observations: 32,560
Variables: 12
$ age                <dbl> 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30, 23, 32,...
$ workClassification <fct> Self-emp-not-inc, Private, Private, Private, Privat...
$ educationLevel     <fct> Bachelors, HS-grad, 11th, Bachelors, Masters, 9th, ...
```

Train/Test Split partition with caret(). Set seed for reproducibility

```
> library(caret)
> set.seed(1234)
> sample_set <- createDataPartition(y = income$income, p = .75, list = FALSE)
> income_train <- income[sample_set,]
> income_test <- income[-sample_set,]
```

Balance imbalanced dataset with SMOTE()

```
> library(DMwR)
> set.seed(1234)
> income_train <-
   SMOTE(income ~ .,
         data.frame(income_train),
         perc.over = 100,
         perc.under = 200)
```

Train & validate the model. Randomly select 90% as training data, 10% as validation over 10 different iterations.

```
> library(rpart)
> set.seed(1234)
> income_mod <- train(
   income ~ .,
   data = income_train,
   metric = "Accuracy",
   method = "rpart",
   trControl = trainControl(method = "LGOCV", p = .1, number = 10)
)
```

View Accuracy

```
> income_mod$resample %>%
   arrange(Resample)

   Accuracy     Kappa Resample
1 0.7963868 0.5927808    Fold1
2 0.7861395 0.5722789    Fold2
3 0.7333192 0.4666383    Fold3
4 0.7309245 0.4618247    Fold4
5 0.7774235 0.5548469    Fold5
```

```
> income_mod$resample %>%
   arrange(Resample) %>%
   summarise(AvgAccuracy = mean(Accuracy))
  AvgAccuracy
1   0.7648387
```

Code (Bootstrap)

Import & view data

```
> library(tidyverse)
> income <- read_csv("income.csv", col_types = "nffnfffffnff")
> glimpse(income)

Observations: 32,560
Variables: 12
$ age                <dbl> 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30, 23, 32,...
$ workClassification <fct> Self-emp-not-inc, Private, Private, Private, Privat...
$ educationLevel     <fct> Bachelors, HS-grad, 11th, Bachelors, Masters, 9th, ...
```

Train/Test Split partition with caret(). Set seed for reproducibility

```
> library(caret)
> set.seed(1234)
> sample_set <- createDataPartition(y = income$income, p = .75, list = FALSE)
> income_train <- income[sample_set,]
> income_test <- income[-sample_set,]
```

Balance imbalanced dataset with SMOTE()

```
> library(DMwR)
> set.seed(1234)
> income_train <-
   SMOTE(income ~ .,
         data.frame(income_train),
         perc.over = 100,
         perc.under = 200)
```

Train & validate the model.

```
> library(rpart)
> set.seed(1234)
> income_mod <- train(
   income ~ .,
   data = income_train,
   metric = "Accuracy",
   method = "rpart",
   trControl = trainControl(method = "boot632", number = 3)
```

View Accuracy

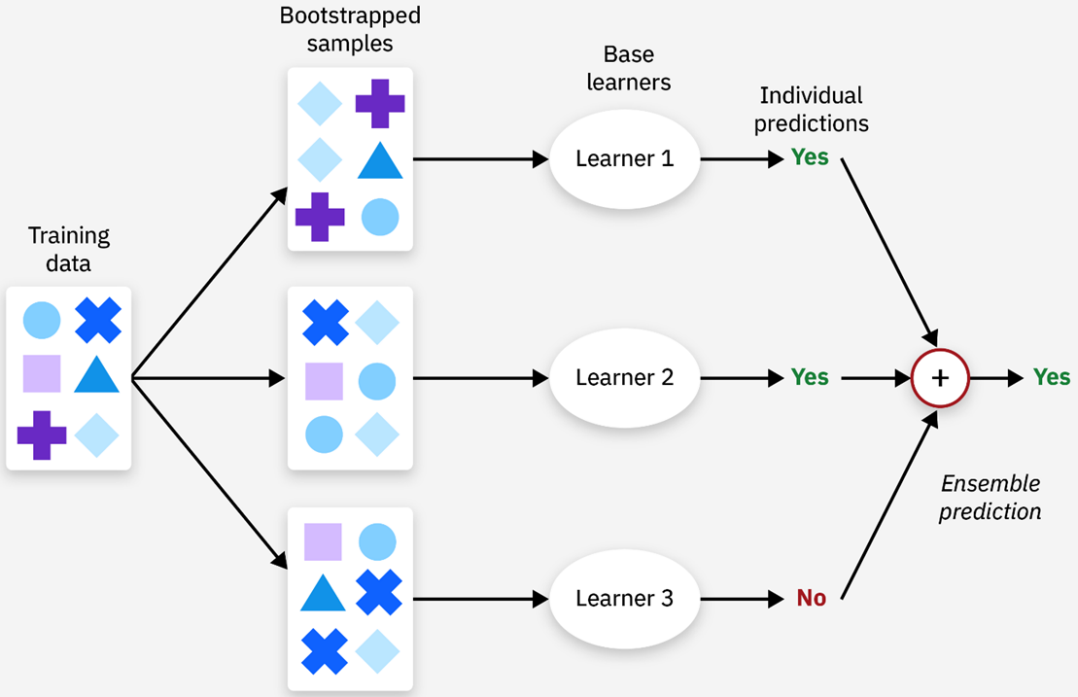
```
> income_mod$resample %>%
   arrange(Resample)

   Accuracy     Kappa Resample
1 0.7963868 0.5927808    Fold1
2 0.7861395 0.5722789    Fold2
3 0.7333192 0.4666383    Fold3
4 0.7309245 0.4618247    Fold4
5 0.7774235 0.5548469    Fold5
```

```
> income_mod$resample %>%
   arrange(Resample) %>%
   summarise(AvgAccuracy = mean(Accuracy))

  AvgAccuracy
1   0.7648387
```

Ensemble Methods

We might not find the optimal set of hyperparameters for a single model and even if we did, the model may not capture all the underlying patterns.

Instead of optimizing the performance of one model, use several complementary weak models to build a much more effective and powerful model.



Allocation Function :
Algorithm that decides how much of the training data is assigned to each model in the ensemble.

Can assign all or part of the data to an ensemble model.

By varying a model's input, we can control the learning/bias components.

Combination Function :
Algorithm that decides how to reconcile and weigh different answers from different learners.

If the ensemble method consists of multiple of the same model, it's ***homogenous.***

If we combine varied models, it is a ***heterogeneous*** ensemble.

Code (Ensemble)

Train & validate the model: Random forest

```
> set.seed(1234)
> rf_mod <- train(
    income ~ .,
    data = income_train,
    metric = "Accuracy",
    method = "rf",
    trControl = trainControl(method = "none"),
    tuneGrid = expand.grid(.mtry = 3)
  )
```

```
> rf_pred <- predict(rf_mod, income_test)
> confusionMatrix(rf_pred, income_test$income, positive = "<=50K")

Confusion Matrix and Statistics

          Reference
Prediction <=50K >50K
     <=50K  4981  495
     >50K   1198 1465

               Accuracy : 0.792
                 95% CI : (0.783, 0.8008)
    No Information Rate : 0.7592
    P-Value [Acc > NIR] : 1.099e-12

                  Kappa : 0.4932

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.8061
            Specificity : 0.7474
         Pos Pred Value : 0.9096
         Neg Pred Value : 0.5501
             Prevalence : 0.7592
         Detection Rate : 0.6120
   Detection Prevalence : 0.6728
      Balanced Accuracy : 0.7768

       'Positive' Class : <=50K
```

Ensemble: Boosting

Adaptive Boosting:
Homogenous sequential models. Within the sequence, each model attempts the performance of prior model by focusing on mistakes.

Each model is evaluated, scored, and shapes the future model in the ensemble. The combination functions considers a model's performance for its influence.

Gradient Boosting:
Instead of correctly predicting prior mistakes, predict the residuals.

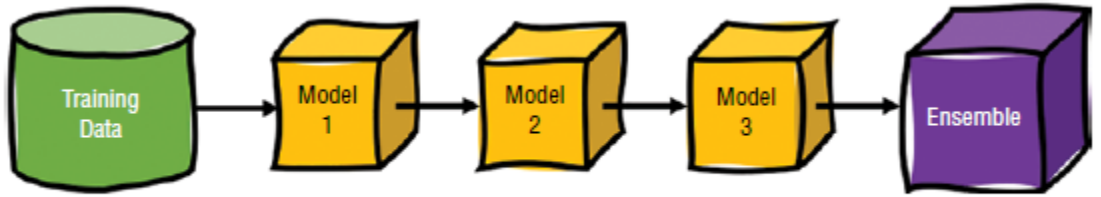
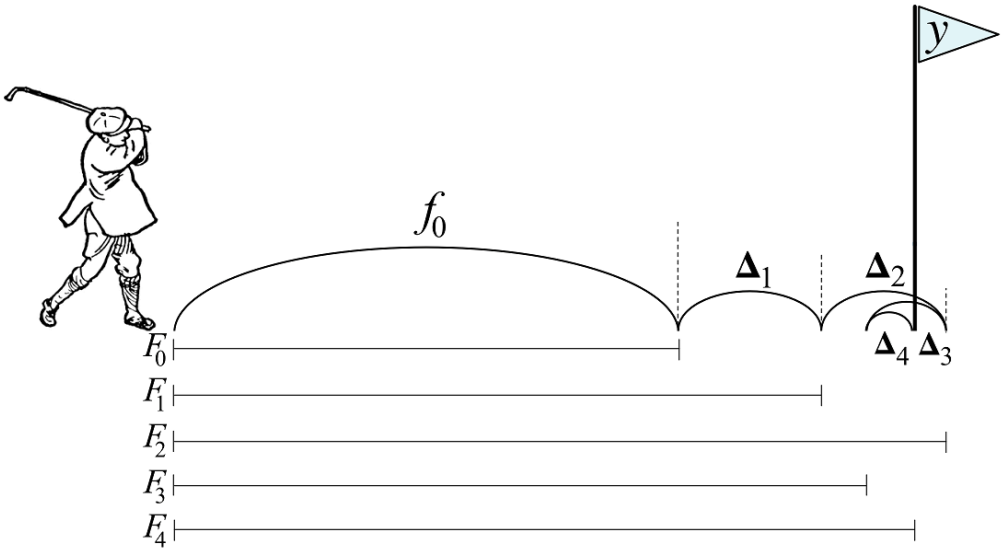


Figure 10.4 The boosting ensemble features a linear sequence of homogenous models.



Adaptive Boosting (Adaboost)

Initial Fitting

Fit a classifier to the original dataset- the foundation for future estimators.

Weight Adjustment

Misclassified observations have their weights adjusted, for future classifiers to focus on them.

Re-weighting Instances

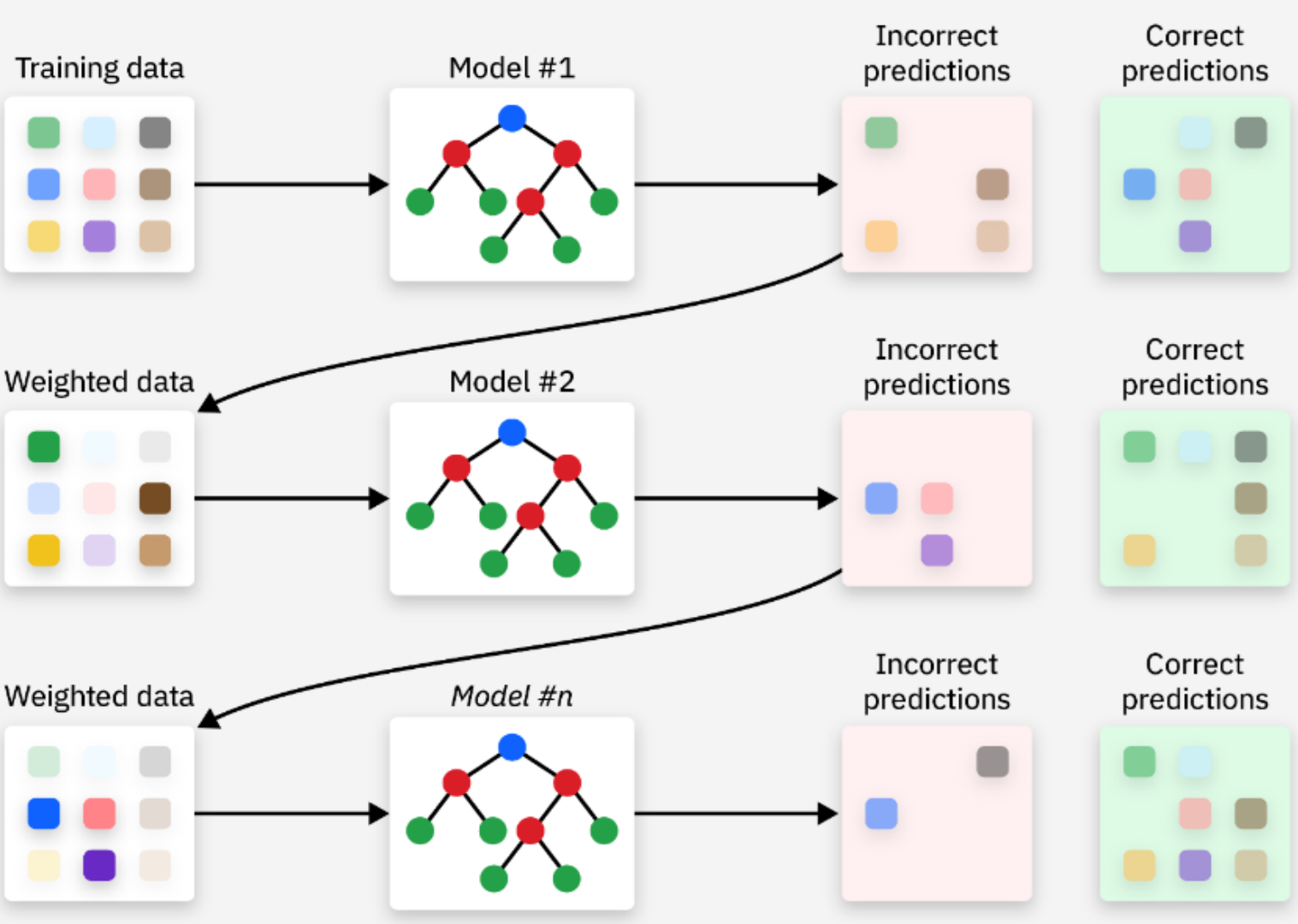
Re-weighted instances are passed to the next estimator in the sequence, until error reduction is minimized or the maximum number of estimators is reached.

Iteration Continuation

Each subsequent estimator is built based on the performance of the previous one.

The aim is to enhance accuracy by addressing the misclassified instances.

We opt out of hyperparameter tuning for our model, and thus avoid resampling.



Adaboost Pros/Cons

Speed: AdaBoost operates quickly, enhancing task efficiency, by allowing for rapid iterations to improve performance.

Simplicity: Straightforward to implement and program, requiring minimal parameter adjustments for usage.

Accuracy: Focusing on misclassified instances to boost performance, by combining many weak learners into strong ones.

Computational Efficiency: Subsequent weak learners get tweaked in favor of those instances misclassified earlier.

Versatility: Works with different data/models via Ensemble methods.

Scalability: The sequential nature limits its scalability and efficiency with large datasets, so it can't be parallelized.

Data Limitations: Poor performance occurs with insufficient data in AdaBoost.

Noise Sensitivity: Sensitive to noise and outliers, as it tries to learn from all instances, needing careful data preparation.

Transparency: Easy to interpret, but the individual weak learners in the ensemble are difficult to understand.

Feature Bias: Adaboost may give more weight to features highly correlated to the target variable.

Code (XGBoost)

```
> library(xgboost)
> modelLookup("xgbTree")
```

	model	parameter	label	forReg	forClass	probModel
1	xgbTree	nrounds	# Boosting Iterations	TRUE	TRUE	TRUE
2	xgbTree	max_depth	Max Tree Depth	TRUE	TRUE	TRUE
3	xgbTree	eta	Shrinkage	TRUE	TRUE	TRUE
4	xgbTree	gamma	Minimum Loss Reduction	TRUE	TRUE	TRUE
5	xgbTree	colsample_bytree	Subsample Ratio of Columns	TRUE	TRUE	TRUE
6	xgbTree	min_child_weight	Minimum Sum of Instance Weight	TRUE	TRUE	TRUE
7	xgbTree	subsample	Subsample Percentage	TRUE	TRUE	TRUE

Higher ‘nrounds’ = better performance, but overfit on training data.

```
> set.seed(1234)
> xgb_mod <- train(
    income ~ .,
    data = income_train,
    metric = "Accuracy",
    method = "xgbTree",
    trControl = trainControl(method = "none"),
    tuneGrid = expand.grid(
      nrounds = 100,
      max_depth = 6,
      eta =  0.3,
      gamma = 0.01,
      colsample_bytree = 1,
      min_child_weight = 1,
      subsample = 1
    )
  )
```

Evaluate model against test data.

```
> xgb_pred <- predict(xgb_mod, income_test)
> confusionMatrix(xgb_pred, income_test$income, positive = "<=50K")
Confusion Matrix and Statistics

          Reference
Prediction <=50K >50K
     <=50K  5168  477
     >50K   1011 1483

               Accuracy : 0.8172
                 95% CI : (0.8086, 0.8255)
    No Information Rate : 0.7592
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5425

 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.8364
            Specificity : 0.7566
         Pos Pred Value : 0.9155
         Neg Pred Value : 0.5946
             Prevalence : 0.7592
         Detection Rate : 0.6350
   Detection Prevalence : 0.6936
      Balanced Accuracy : 0.7965

       'Positive' Class : <=50K
```

Ensemble: Stacking

Heterogeneous models, i.e. a *k*-NN, Naive Bayes, Linear Regression

Relies on several independent learners, whose predictions go to a combination function.

The final model that learns from outputs of other is a ***meta model.***

When combining predictions of different models, ensure the base models have very low correlation- this tells us the models are good in different ways and don't approach problems similarly.

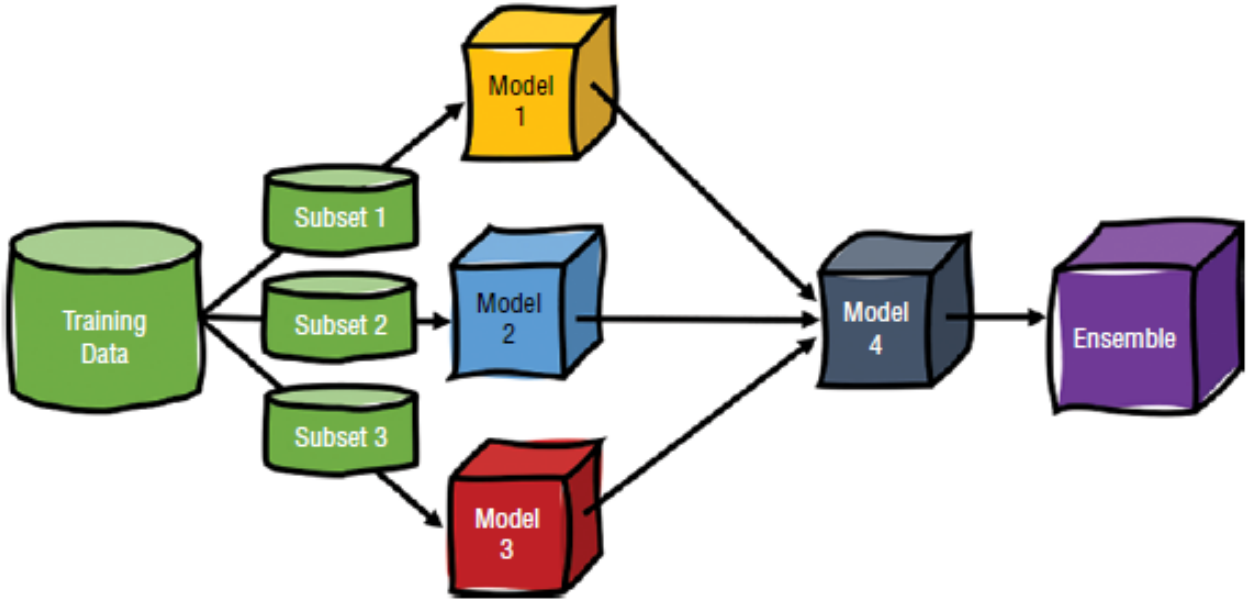


Figure 10.5 The stacking ensemble features independently trained heterogeneous models with a meta-model as the combination function.

Code (Stacking)

```
> library(caret)
> set.seed(1234)
> sample_set <-
   createDataPartition(y = income$income, p = .75, list = FALSE)
> income_train <- income[sample_set, ]
> income_test <- income[-sample_set, ]

> set.seed(1234)
> income_train <-
   SMOTE(income ~ .,
         data.frame(income_train),
         perc.over = 100,
         perc.under = 200)
```

Create learners: Decision Tree, Logistic Regression, KNN

```
> library(caretEnsemble)
> ensembleLearners <- c("rpart","glm","knn")
> library(rpart)
> library(stats)
> library(class)
```

View Accuracy

```
> results <- resamples(models)
> summary(results)
```

```
> modelCor(results)
          rpart         glm         knn
rpart  1.00000000 -0.04723051 -0.1593756
glm   -0.04723051  1.00000000  0.3920402
knn   -0.15937561  0.39204015  1.0000000
```

Train model: 10-fold CV 5 times

```
> models <- caretList(
   income ~ .,
   data = income_train,
   metric = "Accuracy",
   methodList = ensembleLearners,
   trControl = trainControl(
     method = "repeatedcv",
     number = 10,
     repeats = 5,
     savePredictions = "final",
     classProbs = TRUE
```

Build meta model.

```
> library(randomForest)
> stack_mod <- caretStack(
   models,
   method = "rf",
   metric = "Accuracy",
   trControl = trainControl(
     method = "repeatedcv",
     number = 10,
     repeats = 5,
     savePredictions = "final",
     classProbs = TRUE
```