Machine Learning Cheat Sheet

April 21, 2025

Contents

1 Linear Models

1.1 Linear Regression

Linear Regression

Formula

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon \qquad (1)$$

R Implementation

```r
# Fit
model <- lm(y ~ x1 + x2, data = train_data)
# Predict
predictions <- predict(model, newdata = test_data)
```

Useful Functions

```r
# Basic inspection
summary(model)           # Detailed summary with coefficients & p-
   values
coef(model)              # Extract coefficients
confint(model)           # Confidence intervals
plot(model)              # Diagnostic plots
residuals(model)         # Extract residuals
fitted(model)            # Fitted values
AIC(model)               # Akaike Information Criterion
BIC(model)               # Bayesian Information Criterion
```

Advantages

- Simple to implement and interpret
- Computationally efficient
- Provides explicit feature importance
- Works well for linearly separable data
- Solid statistical foundation

Disadvantages

- Assumes linear relationship between features and target
- Sensitive to outliers
- Poor performance with high-dimensional data
- Cannot model complex non-linear relationships
- Assumes independence of errors and features

1.2 Ridge Regression

Ridge Regression

Formula

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \tag{2}$$

R Implementation

```r
# Fit (glmnet package)
library(glmnet)
x_train <- model.matrix(y ~ . + 0, data = train_data)
# Simple fit with fixed lambda
model <- glmnet(x_train, train_data$y, alpha = 0, lambda = 0.1)
# Cross-validation for lambda selection
cv_model <- cv.glmnet(x_train, train_data$y, alpha = 0)
best_lambda <- cv_model$lambda.min
model <- glmnet(x_train, train_data$y, alpha = 0, lambda = best_lambda)
# Predict
x_test <- model.matrix(~ . + 0, test_data)
predictions <- predict(model, newx = x_test)
```

Useful Functions

```r
# Inspect model
coef(model, s = lambda_value)   # Extract coefficients for specific
    lambda
plot(model, xvar = "lambda")     # Plot coefficient paths
plot(cv_model)                   # Plot CV error vs lambda
cv_model$lambda.min              # Lambda with minimum CV error
cv_model$lambda.1se              # More conservative lambda (1 SE rule)
```

Advantages

- Handles multicollinearity effectively
- Reduces overfitting compared to ordinary least squares
- Stable coefficient estimates with high-dimensional data
- Computationally efficient closed-form solution
- Shrinks coefficients toward zero but never to exactly zero

Disadvantages

- Does not perform feature selection
- Still assumes linear relationship between features and target
- Requires tuning of regularization parameter
- Less interpretable than unregularized linear regression
- Less effective with truly sparse underlying models

1.3 LASSO

LASSO

Formula

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{3}$$

R Implementation

```
# Fit (glmnet package)
library(glmnet)
x_train <- model.matrix(y ~ . + 0, data = train_data)
# Simple fit with fixed lambda
model <- glmnet(x_train, train_data$y, alpha = 1, lambda = 0.1)
# Cross-validation for lambda selection
cv_model <- cv.glmnet(x_train, train_data$y, alpha = 1)
best_lambda <- cv_model$lambda.min
model <- glmnet(x_train, train_data$y, alpha = 1, lambda = best_lambda)
# Predict
x_test <- model.matrix(~ . + 0, test_data)
predictions <- predict(model, newx = x_test)
```

Useful Functions

```
# Inspect model
coef(model, s = lambda_value)    # Extract coefficients for specific
    lambda
plot(model, xvar = "lambda")     # Plot coefficient paths
# Get non-zero coefficients
coef_vector <- coef(model, s = best_lambda)
non_zero_coefs <- coef_vector[which(coef_vector != 0),]
```

Advantages

- Performs automatic feature selection
- Produces sparse models with fewer parameters
- Reduces overfitting
- Handles high-dimensional data effectively
- Works well when many features have zero coefficients

Disadvantages

- No closed-form solution
- Unstable with highly correlated features
- Can select only one from a group of correlated features
- Requires tuning of regularization parameter
- Still assumes linear relationship between features and target

2 Classification Models

2.1 Logistic Regression

Logistic Regression

Formula

$$P(Y=1|X) = \frac{e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}} \quad (4)$$

R Implementation

```
# Fit
model <- glm(y ~ x1 + x2, data = train_data, family = "binomial")
# Predict classes
class_pred <- predict(model, newdata = test_data, type = "response") >
    0.5
# Predict probabilities (for ROC/AUC)
prob_pred <- predict(model, newdata = test_data, type = "response")
```

Useful Functions

```
# Inspect model
summary(model)                  # Model summary
coef(model)                     # Coefficients
exp(coef(model))                # Odds ratios (exponentiated
    coefficients)
confint(model)                  # Confidence intervals for
    coefficients
exp(confint(model))             # Confidence intervals for odds
    ratios
```

Advantages

- Directly models probability of class membership
- Efficient training and prediction
- Less prone to overfitting than complex models
- Provides interpretable feature importance
- Handles both continuous and categorical features

Disadvantages

- Assumes linear decision boundary
- Limited to binary or multinomial classification
- Struggles with imbalanced datasets
- Assumes independence of features
- Sensitive to outliers

2.2 k-Nearest Neighbors

k-Nearest Neighbors

Formula

$$\hat{y} = \frac{1}{k} \sum_{i \in N_k(x)} y_i \qquad (5)$$

R Implementation

```
# Fit & predict in one step (class package)
library(class)
# For classification
predictions <- knn(train = train_data[, -1],
                   test = test_data[, -1],
                   cl = train_data$y,
                   k = 5)
# For probabilities
predictions <- knn(train = train_data[, -1],
                   test = test_data[, -1],
                   cl = train_data$y,
                   k = 5,
                   prob = TRUE)
probs <- attr(predictions, "prob")
```

Useful Functions

```
# kNN doesn't create a model object, so inspection is limited
table(predictions, test_data$y)   # Confusion matrix
mean(predictions == test_data$y)  # Accuracy
```

Advantages

- No training phase required
- Non-parametric and makes no assumptions about data distribution
- Simple to implement
- Works well with multi-class problems
- Can model complex decision boundaries

Disadvantages

- Computationally expensive for large datasets
- Requires feature scaling
- Sensitive to irrelevant features
- Requires selecting optimal k value
- Poor performance in high-dimensional spaces

2.3 Linear Discriminant Analysis

Linear Discriminant Analysis

Formula

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\pi_k) \tag{6}$$

R Implementation

```
# Fit (MASS package)
library(MASS)
model <- lda(y ~ ., data = train_data)
# Predict
predictions <- predict(model, newdata = test_data)
# Access classes and probabilities
classes <- predictions$class
probs <- predictions$posterior
```

Useful Functions

```
# Inspect model
model$means                        # Group means
model$scaling                      # Discriminant coefficients
model$prior                        # Prior probabilities
plot(model)                        # Plot discriminant functions
```

Advantages

- Works well with small sample sizes
- Provides dimensionality reduction
- Handles multiclass problems naturally
- Less sensitive to outliers than logistic regression
- Computationally efficient

Disadvantages

- Assumes normal distribution of features
- Assumes homoscedasticity (equal covariance matrices)
- Limited to linear decision boundaries
- Sensitive to multicollinearity
- Requires complete cases or imputation

2.4 Quadratic Discriminant Analysis

Quadratic Discriminant Analysis

Formula

$$\delta_k(x) = -\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) - \frac{1}{2} \log |\Sigma_k| + \log(\pi_k) \tag{7}$$

R Implementation

```
# Fit (MASS package)
library(MASS)
model <- qda(y ~ ., data = train_data)
# Predict
predictions <- predict(model, newdata = test_data)
# Access classes and probabilities
classes <- predictions$class
probs <- predictions$posterior
```

Useful Functions

```
# Inspect model
model$means                       # Group means
model$scaling                     # Not available for QDA
model$prior                       # Prior probabilities
```

Advantages

- Models non-linear decision boundaries
- Does not assume equal covariance matrices
- Often outperforms LDA when assumption of equal covariances is violated
- Handles multiclass problems naturally
- Works well with sufficient training data

Disadvantages

- Requires more parameters than LDA
- Needs larger sample sizes
- Assumes normal distribution of features
- More prone to overfitting than LDA
- Computationally more expensive than LDA

3 Tree-Based Models

3.1 Decision Trees

Decision Trees

Formula

$$R_m = \{X | X_j \leq t_m\} \text{ or } R_m = \{X | X_j > t_m\} \tag{8}$$

R Implementation

```r
# Fit using tree package
library(tree)
model <- tree(y ~ ., data = train_data)
# Predict classes (for classification)
class_pred <- predict(model, newdata = test_data, type = "class")
# Predict probabilities (for classification)
prob_pred <- predict(model, newdata = test_data)
# Predict values (for regression)
reg_pred <- predict(model, newdata = test_data)
# Pruning
cv_tree <- cv.tree(model, FUN = prune.misclass)  # For classification
opt_size <- cv_tree$size[which.min(cv_tree$dev)]
pruned_model <- prune.misclass(model, best = opt_size)
```

Useful Functions

```r
# Inspect model
summary(model)                    # Model summary
print(model)                      # Print tree details
plot(model)                       # Plot the tree
text(model, pretty = 0)           # Add text labels to plot
model$frame                       # Node information
model$splits                      # Split details
# Cross-validation results
plot(cv_tree)                     # Plot CV results
cv_tree$size                      # Tree sizes
cv_tree$dev                       # Deviance for each size
cv_tree$k                         # Cost-complexity parameter
```

Advantages

- Highly interpretable model
- Handles both numerical and categorical features
- No feature scaling required
- Captures non-linear relationships
- Performs automatic feature selection

Disadvantages

- Prone to overfitting
- Unstable - small data changes can significantly alter tree structure
- Biased toward features with more levels
- Cannot extrapolate beyond training data
- Struggles with unbalanced datasets

3.2 Random Forests

Random Forests

Formula

$$\hat{f}_{rf}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{9}$$

R Implementation

```r
# Fit (randomForest package)
library(randomForest)
model <- randomForest(y ~ ., data = train_data, ntree = 500)
# Predict classes
predictions <- predict(model, newdata = test_data)
# Predict probabilities (for classification)
prob_pred <- predict(model, newdata = test_data, type = "prob")
```

Useful Functions

```r
# Inspect model
print(model)                          # Basic model info
summary(model)                        # Model summary
# Variable importance
importance(model)                     # Numerical importance
varImpPlot(model)                     # Plot importance
# Out-of-bag error
plot(model)                           # Plot OOB error vs trees
model$err.rate                        # Error rate matrix
model$confusion                       # OOB confusion matrix
# Extract a specific tree
getTree(model, k = 1, labelVar = TRUE)  # Get kth tree
# Partial dependence plots
partialPlot(model, pred.data = train_data, x.var = "variable_name")
```

Advantages

- Reduces overfitting compared to individual trees
- Handles high-dimensional data well
- Provides feature importance metrics
- Robust to outliers and noise
- Handles missing values effectively

Disadvantages

- Less interpretable than single decision trees
- Computationally intensive
- Slower prediction time than single trees
- Cannot extrapolate beyond training data
- May overfit in some noisy classification tasks

3.3 Bagging

Bagging

Formula

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x) \tag{10}$$

R Implementation

```r
# Fit (randomForest with all features)
library(randomForest)
p <- ncol(train_data) - 1  # number of features
model <- randomForest(y ~ ., data = train_data, mtry = p)
# Predict
predictions <- predict(model, newdata = test_data)
```

Useful Functions

```r
# Same functions as for Random Forests
print(model)                        # Basic model info
summary(model)                      # Model summary
importance(model)                   # Variable importance
varImpPlot(model)                   # Plot importance
plot(model)                         # Plot OOB error vs trees
```

Advantages

- Reduces variance without increasing bias
- Works well with strong, complex base learners
- Parallelizable training process
- Provides out-of-bag error estimate
- Less prone to overfitting than single models

Disadvantages

- Computationally intensive
- Does not reduce bias
- Increased model complexity
- Less interpretable than individual models
- Not effective for high-bias base learners

3.4 Gradient Boosting

Gradient Boosting

Formula

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma) \tag{11}$$

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x) \text{ for } m = 1, 2, ..., M \tag{12}$$

R Implementation

```
# Fit (gbm package)
library(gbm)
# For classification
model <- gbm(y ~ ., data = train_data,
             distribution = "bernoulli", n.trees = 100)
# For regression
model <- gbm(y ~ ., data = train_data,
             distribution = "gaussian", n.trees = 100)
# Predict
predictions <- predict(model, newdata = test_data,
                       n.trees = 100, type = "response")
```

Useful Functions

```
# Inspect model
summary(model)                       # Variable importance and plots
# Find optimal number of trees
gbm.perf(model, method = "OOB")     # Using out-of-bag samples
gbm.perf(model, method = "test")    # Using test set
gbm.perf(model, method = "cv")      # Using cross-validation
# Partial dependence plots
plot(model, i = "variable_name")
plot(model, i.var = 1)               # First variable
# Variable importance
relative_influence(model, n.trees = 100)
```

Advantages

- Often achieves state-of-the-art performance
- Handles different loss functions
- Works well with mixed data types
- Provides feature importance metrics
- Robust to outliers with robust loss functions

Disadvantages

- Prone to overfitting without careful tuning
- Computationally intensive
- Sequential nature limits parallelization
- Sensitive to noisy data
- Requires more hyperparameter tuning than random forests

4 Support Vector Machines

4.1 Linear SVM

Support Vector Machines (Linear)

Formula

$$\max_{\beta_0,\beta,||\beta||=1} M \tag{13}$$

$$\text{subject to } y_i(\beta_0 + \beta^T x_i) \geq M, i = 1, ..., n \tag{14}$$

R Implementation

```r
# Fit (e1071 package)
library(e1071)
model <- svm(y ~ ., data = train_data, kernel = "linear")
# Predict classes
predictions <- predict(model, newdata = test_data)
# Fit with probability estimation for ROC
model <- svm(y ~ ., data = train_data, kernel = "linear",
             probability = TRUE)
# Get probabilities
prob_pred <- predict(model, newdata = test_data, probability = TRUE)
probs <- attr(prob_pred, "probabilities")
```

Useful Functions

```r
# Inspect model
summary(model)                      # Model summary
model$SV                            # Support vectors
model$index                         # Indices of support vectors
length(model$index)                 # Number of support vectors
# Tuning
tune_result <- tune(svm, train.x = x, train.y = y,
                    kernel = "linear",
                    ranges = list(cost = 10^(-1:2)))
tune_result$best.parameters         # Best parameters
plot(tune_result)                   # Plot tuning results
```

Advantages

- Effective in high-dimensional spaces
- Memory efficient as only support vectors matter
- Robust to outliers with proper regularization
- Strong theoretical guarantees
- Works well with clear margin of separation

Disadvantages

- Poor performance with overlapping classes
- Sensitive to choice of kernel and parameters
- No probabilistic output by default
- Computationally intensive for large datasets
- Limited to linear decision boundaries

4.2 Radial SVM

Support Vector Machines (Radial)

Formula

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x, x_i) \tag{15}$$

With radial basis function (RBF) kernel:

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2) \tag{16}$$

R Implementation

```r
# Fit (e1071 package)
library(e1071)
model <- svm(y ~ ., data = train_data, kernel = "radial", gamma = 0.5)
# Predict
predictions <- predict(model, newdata = test_data)
# For probabilities
model <- svm(y ~ ., data = train_data, kernel = "radial",
             probability = TRUE)
prob_pred <- predict(model, newdata = test_data, probability = TRUE)
probs <- attr(prob_pred, "probabilities")
```

Useful Functions

```r
# Inspect model - same functions as linear SVM plus:
model$gamma                      # Gamma parameter value
# Tuning both cost and gamma parameters
tune_result <- tune(svm, train.x = x, train.y = y,
                   kernel = "radial",
                   ranges = list(cost = 10^(-1:2),
                                 gamma = c(0.1, 0.5, 1, 2)))
```

Advantages

- Handles complex non-linear boundaries effectively
- Works well for data that is not linearly separable
- Effective when number of features exceeds samples
- Less prone to overfitting with proper regularization
- Versatile for different types of data

Disadvantages

- Highly sensitive to gamma parameter choice
- Poor performance with large datasets due to $O(n^2)$ scaling
- Difficult to interpret feature importance
- Requires careful feature scaling
- Memory intensive for large datasets

4.3 Polynomial SVM

Support Vector Machines (Polynomial)

Formula

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x, x_i) \tag{17}$$

With polynomial kernel:

$$K(x_i, x_j) = (\gamma \cdot x_i^T x_j + r)^d \tag{18}$$

R Implementation

```r
# Fit (e1071 package)
library(e1071)
model <- svm(y ~ ., data = train_data,
             kernel = "polynomial",
             degree = 3,        # Polynomial degree
             coef0 = 1)         # Parameter r in the formula
# Predict
predictions <- predict(model, newdata = test_data)
# For probabilities
model <- svm(y ~ ., data = train_data,
             kernel = "polynomial",
             degree = 3,
             coef0 = 1,
             probability = TRUE)
prob_pred <- predict(model, newdata = test_data, probability = TRUE)
probs <- attr(prob_pred, "probabilities")
```

Useful Functions

```
# Inspect model
summary(model)                    # Model summary
model$SV                          # Support vectors
model$degree                      # Polynomial degree
model$coef0                       # Value of r parameter
# Tuning parameters
tune_result <- tune(svm, train.x = x, train.y = y,
                   kernel = "polynomial",
                   ranges = list(degree = 2:4,
                                cost = 10^(-1:2),
                                coef0 = c(0, 1)))
```

Advantages

- More flexible than linear SVM
- Can model non-linear decision boundaries of varying complexity
- Often less computationally intensive than RBF kernel
- Good for data with clear polynomial trends
- Works well for normalized data

Disadvantages

- More parameters to tune (degree, coefficient, cost)
- Can lead to overfitting with high degree values
- Less common in practice than linear or RBF kernels
- Numeric instability with high-degree polynomials
- Difficult to interpret

5 Neural Networks

Neural Networks

Formula

$$z_j^{(l)} = \sigma\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) \quad (19)$$

$$a_j^{(l)} = \sigma(z_j^{(l)}) \quad (20)$$

R Implementation

```r
# Scale data first (important for neural networks)
library(nnet)
train_scaled <- scale(train_data[, -1])
test_scaled <- scale(test_data[, -1],
                     center = attr(train_scaled, "scaled:center"),
                     scale = attr(train_scaled, "scaled:scale"))

# Fit (nnet package)
model <- nnet(y ~ .,
             data = data.frame(y = train_data$y, train_scaled),
             size = 5,      # Number of hidden units
             decay = 0.01,  # Weight decay for regularization
             maxit = 1000)  # Maximum iterations

# Predict classes
predictions <- predict(model,
                       newdata = data.frame(test_scaled),
                       type = "class")
# Predict probabilities
prob_pred <- predict(model,
                     newdata = data.frame(test_scaled),
                     type = "raw")
```

Useful Functions

```r
# Inspect model
summary(model)                    # Network summary
model$wts                         # Neural network weights
model$n                           # Network architecture info
# Network visualization (requires NeuralNetTools package)
library(NeuralNetTools)
plotnet(model)                    # Visualize network structure
garson(model)                     # Variable importance
olden(model)                      # Connection weights approach
```

Advantages

- Models complex non-linear relationships
- Automatic feature extraction
- Highly adaptable to different data types
- Strong performance on unstructured data
- Parallelizable training with GPUs

Disadvantages

- Requires large amounts of training data
- Computationally intensive training
- Prone to overfitting with small datasets
- Difficult to interpret (”black box”)
- Many hyperparameters requiring tuning

6 Survival Analysis

Cox Proportional Hazards Model

Formula

$$h(t|X) = h_0(t)\exp(\beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p) \tag{21}$$

R Implementation

```r
# Fit (survival package)
library(survival)
model <- coxph(Surv(time, status) ~ x1 + x2, data = train_data)
# Predict risk scores
predictions <- predict(model, newdata = test_data, type = "risk")
# Predict survival curves
surv_curves <- survfit(model, newdata = test_data)
```

Useful Functions

```r
# Inspect model
summary(model)                    # Model summary
coef(model)                       # Coefficients
exp(coef(model))                  # Hazard ratios
confint(model)                    # Confidence intervals
exp(confint(model))               # CIs for hazard ratios
# Test proportional hazards assumption
ph_test <- cox.zph(model)
print(ph_test)
plot(ph_test)                     # Plot test results
# Baseline hazard/survival
basehaz(model)                    # Baseline cumulative hazard
# Visualizing survival curves
plot(surv_curves)                 # Plot survival curves
```

Advantages

- Handles censored survival data
- No assumption about baseline hazard distribution
- Provides interpretable hazard ratios
- Works well with mixed categorical and continuous predictors
- Robust to non-normal data

Disadvantages

- Assumes proportional hazards over time
- Cannot handle time-varying effects without modification
- Sensitive to outliers
- Limited to survival analysis applications
- May overfit with high-dimensional data

7 Dimensionality Reduction

Principal Component Analysis

Formula

$$Z_i = \phi_1^T X_i + \phi_2^T X_i + ... + \phi_m^T X_i \tag{22}$$

R Implementation

```
# Fit
pca <- prcomp(train_data[, -1], center = TRUE, scale. = TRUE)
# Project data to PC space
pc_scores <- predict(pca, newdata = test_data[, -1])
# Variance explained
var_explained <- pca$sdev^2 / sum(pca$sdev^2)
# Reconstruct data using first 2 components
reconstructed <- pc_scores[, 1:2] %*% t(pca$rotation[, 1:2])
```

Useful Functions

```
# Inspect model
summary(pca)                        # Summary of components
pca$rotation                        # Loadings (eigenvectors)
pca$sdev                            # Standard deviations of components
pca$sdev^2 / sum(pca$sdev^2)        # Proportion of variance explained
# Plotting
biplot(pca)                         # Biplot (observations and variables)
plot(pca)                           # Scree plot
screeplot(pca, type = "lines")      # Prettier scree plot
```

Advantages

- Reduces dimensionality while preserving variance
- Removes multicollinearity
- Improves computational efficiency for subsequent models
- Helpful for visualization of high-dimensional data
- Unsupervised technique requiring no labels

Disadvantages

- Components can be difficult to interpret
- Assumes linear relationships between variables
- Sensitive to feature scaling
- May lose important information with aggressive dimensionality reduction
- Computationally expensive for very large datasets

8 Caret Implementation

Caret Framework

Caret provides a unified interface for model training and evaluation. Here's how to use it with various models:

```
library(caret)

# Create trainControl object for resampling
# For classification
ctrl_class <- trainControl(
  method = "cv",                # k-fold cross-validation
  number = 5,                   # number of folds
  classProbs = TRUE,            # compute class probabilities
  summaryFunction = twoClassSummary,  # ROC, Sens, Spec metrics
  savePredictions = "final"     # save final predictions
)

# For regression
ctrl_reg <- trainControl(
  method = "cv",                # k-fold cross-validation
  number = 5                    # number of folds
)
```

Model Training and Tuning

```
# Linear Regression
lm_model <- train(
  y ~ .,
  data = train_data,
  method = "lm",
  trControl = ctrl_reg
)

# Ridge Regression
ridge_model <- train(
  y ~ .,
  data = train_data,
  method = "glmnet",
  tuneGrid = expand.grid(alpha = 0,
                         lambda = seq(0.0001, 1, length = 20)),
  trControl = ctrl_reg
)

# LASSO
lasso_model <- train(
  y ~ .,
  data = train_data,
  method = "glmnet",
  tuneGrid = expand.grid(alpha = 1,
                         lambda = seq(0.0001, 1, length = 20)),
  trControl = ctrl_reg
)

# Logistic Regression
logreg_model <- train(
  y ~ .,
  data = train_data,
  method = "glm",
  family = "binomial",
  trControl = ctrl_class,
  metric = "ROC"
)

# k-Nearest Neighbors
knn_model <- train(
  y ~ .,
  data = train_data,
  method = "knn",
  tuneGrid = data.frame(k = seq(1, 20, by = 2)),
  trControl = ctrl_class,
  metric = "ROC"
)

# Random Forest
rf_model <- train(
  y ~ .,
  data = train_data,
  method = "rf",
  tuneLength = 5,
  trControl = ctrl_class,
  metric = "ROC"
)

# Support Vector Machine - Radial
svm_radial_model <- train(
  y ~ .,
  data = train_data,
  method = "svmRadial",
  tuneLength = 5,
  trControl = ctrl_class,
  metric = "ROC"
)

# Neural Network
nnet_model <- train(
  y ~ .,
  data = train_data,
  method = "nnet",
  tuneLength = 5,
  trControl = ctrl_class,
  metric = "ROC",
  trace = FALSE,
  maxit = 100
)
```

Prediction

```
# Standard prediction
predictions <- predict(model, newdata = test_data)

# Probability prediction
```

```
prob_predictions <- predict(model, newdata = test_data, type = "prob")
```

Model Comparison

```
# Combine models for comparison
model_list <- list(
  LR = lm_model,
  Ridge = ridge_model,
  LASSO = lasso_model,
  RF = rf_model,
  SVM = svm_radial_model,
  NNet = nnet_model
)

# Compare models
resamples_obj <- resamples(model_list)
summary(resamples_obj)
dotplot(resamples_obj, metric = "ROC")  # For classification
dotplot(resamples_obj, metric = "RMSE") # For regression
```

Useful Functions

```
# Model information
getModelInfo("method_name")       # Details of a specific model
names(getModelInfo())             # List all available models

# Tuning results
model$results                     # All tuning results
model$bestTune                    # Best tuning parameters
plot(model)                       # Plot tuning results

# Variable importance
varImp(model)                     # Variable importance
plot(varImp(model))               # Plot variable importance

# Confusion matrix
conf_matrix <- confusionMatrix(predictions, test_data$y)
conf_matrix$table                 # Confusion matrix
conf_matrix$byClass               # Class-specific metrics
```

9 Model Selection and Evaluation

Model Selection

Best Subset Selection

```r
library(leaps)
# Best subset selection
reg_subset <- regsubsets(y ~ ., data = train_data, nvmax = 10)
# Inspect results
summary(reg_subset)$which        # Selected variables for each size
summary(reg_subset)$adjr2        # Adjusted R  for each model size
summary(reg_subset)$bic          # BIC for each model size
plot(reg_subset, scale = "adjr2") # Plot results
```

Forward/Backward Selection

```r
# Using regsubsets
reg_fwd <- regsubsets(y ~ ., data = train_data, nvmax = 10, method = "forward
    ")
reg_bwd <- regsubsets(y ~ ., data = train_data, nvmax = 10, method = "
    backward")

# Using step function from stats
full_model <- lm(y ~ ., data = train_data)
null_model <- lm(y ~ 1, data = train_data)

# Forward selection
fwd_model <- step(null_model, scope = formula(full_model),
                  direction = "forward", trace = FALSE)

# Backward selection
bwd_model <- step(full_model, direction = "backward", trace = FALSE)

# Both (stepwise)
both_model <- step(null_model, scope = formula(full_model),
                   direction = "both", trace = FALSE)
```

Cross-Validation

```r
# k-fold cross-validation (manual implementation)
set.seed(123)
k <- 10
folds <- sample(1:k, nrow(train_data), replace = TRUE)
cv_errors <- numeric(k)

for (i in 1:k) {
  # Split data
  train_fold <- train_data[folds != i, ]
  test_fold <- train_data[folds == i, ]

  # Fit model on training fold
  fold_model <- lm(y ~ ., data = train_fold)

  # Predict on test fold
  fold_preds <- predict(fold_model, newdata = test_fold)

  # Calculate error (MSE for regression)
  cv_errors[i] <- mean((fold_preds - test_fold$y)^2)
}

# Overall CV error
mean(cv_errors)
```

Model Evaluation

Classification Metrics

```
# Confusion Matrix
conf_mat <- table(predictions, true_values)
accuracy <- sum(diag(conf_mat)) / sum(conf_mat)

# Using caret's confusionMatrix
library(caret)
cm <- confusionMatrix(predictions, true_values)
cm$overall[1]                     # Accuracy
cm$byClass["Sensitivity"]         # Sensitivity (TPR)
cm$byClass["Specificity"]         # Specificity (TNR)
cm$byClass["Precision"]           # Precision (PPV)
cm$byClass["F1"]                  # F1 Score

# ROC and AUC
library(pROC)
roc_obj <- roc(true_values, prob_predictions)
auc_value <- auc(roc_obj)
plot(roc_obj)
```

Regression Metrics

```
# Mean Squared Error
mse <- mean((predictions - true_values)^2)
# Root Mean Squared Error
rmse <- sqrt(mse)
# Mean Absolute Error
mae <- mean(abs(predictions - true_values))
# R-squared
ssr <- sum((predictions - true_values)^2)
sst <- sum((true_values - mean(true_values))^2)
r_squared <- 1 - (ssr / sst)
```

10 Quick Reference Guide

Classification Models

Model	Package
Logistic Regression	stats (glm)
LDA	MASS
QDA	MASS
k-NN	class
Decision Trees	tree, rpart
Random Forests	randomForest
SVM	e1071
Neural Networks	nnet

Useful Parameters

Parameter	Description
k	Neighbors in k-NN
lambda	Regularization strength
ntree	Trees in Random Forest
mtry	Variables per split (RF)
cost	Cost parameter (SVM)
gamma	Kernel parameter (SVM)
size	Hidden nodes (nnet)
decay	Weight decay (nnet)

Common Caret Methods

Method	Model
"lm"	Linear regression
"glm"	Generalized linear model
"glmnet"	Elastic net (Ridge, LASSO)
"knn"	k-Nearest Neighbors
"lda"	Linear Discriminant Analysis
"rpart"	Decision Trees
"rf"	Random Forests
"svmLinear"	Linear SVM
"svmRadial"	Radial SVM
"nnet"	Neural Networks

Regression Models

Model	Package
Linear Regression	stats (lm)
Ridge Regression	glmnet
LASSO	glmnet
Decision Trees	tree, rpart
Random Forests	randomForest
SVM Regression	e1071
Neural Networks	nnet

11 Key Commands to Remember

Essential R Commands

```r
# Data splitting
set.seed(123)  # Always set seed for reproducibility
train_indices <- sample(1:nrow(data), 0.7 * nrow(data))
train_data <- data[train_indices, ]
test_data <- data[-train_indices, ]

# Data preprocessing
# Scale continuous variables
scaled_data <- scale(data[, continuous_cols])
# Create dummy variables for categorical
model_matrix <- model.matrix(~ . - 1, data[, categorical_cols])

# Missing data
complete_cases <- complete.cases(data)  # Logical vector of complete rows
na_omit_data <- na.omit(data)  # Remove rows with any NA
# Imputation
data$x[is.na(data$x)] <- mean(data$x, na.rm = TRUE)  # Mean imputation

# Basic model evaluation
confusion_matrix <- table(predictions, true_values)
```

```
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
mse <- mean((predictions - true_values)^2)

# Saving/loading models
saveRDS(model, "model.rds")  # Save model
model <- readRDS("model.rds")  # Load model
```

11.1 Recursive Feature Elimination

Recursive Feature Elimination (RFE)

```
# Set up rfeControl object
ctrl <- rfeControl(
  functions = rfFuncs,          # RF as base learner
  method = "cv",                # Cross-validation
  number = 10,                  # Number of folds
  verbose = FALSE
)

# Run RFE
rfe_result <- rfe(
  x = train_data[, -outcome_col],
  y = train_data[, outcome_col],
  sizes = c(1:10),              # Subsets of features to evaluate
  rfeControl = ctrl
)

# Examine results
print(rfe_result)
rfe_result$optVariables         # Optimal feature subset
plot(rfe_result)                # Plot performance by subset size
```

11.2 Feature Filtering Methods

Correlation and Statistical Filtering

```r
# Correlation filter
cor_matrix <- cor(feature_data)
high_cor <- findCorrelation(cor_matrix, cutoff = 0.75)
filtered_data <- feature_data[, -high_cor]

# Univariate feature selection with multiple testing correction
p_values <- numeric(ncol(feature_data))
for(i in 1:ncol(feature_data)) {
  # Simple univariate model for each feature
  model <- lm(y ~ feature_data[,i])
  p_values[i] <- summary(model)$coefficients[2,4]
}

# Adjust p-values for multiple testing
adj_p <- p.adjust(p_values, method = "fdr")  # False Discovery Rate
significant_features <- which(adj_p < 0.05)
```

11.3 Confusion Matrix Metrics

Confusion Matrix Analysis

```r
# Create confusion matrix
conf_mat <- table(predictions, true_values)

# Basic metrics
total_observations <- sum(conf_mat)
accuracy <- sum(diag(conf_mat)) / total_observations
error_rate <- 1 - accuracy

# Class-specific metrics
# For binary classification with positive = "Yes"
TP <- conf_mat["Yes", "Yes"]
TN <- conf_mat["No", "No"]
FP <- conf_mat["Yes", "No"]
FN <- conf_mat["No", "Yes"]

# Class metrics
sensitivity <- TP / (TP + FN)  # Recall, True Positive Rate
specificity <- TN / (TN + FP)  # True Negative Rate
precision <- TP / (TP + FP)    # Positive Predictive Value
f1_score <- 2 * precision * sensitivity / (precision + sensitivity)

# Using caret
cm <- confusionMatrix(factor(predictions), factor(true_values),
                      positive = "Yes")
print(cm)
cm$overall                     # Overall statistics
cm$byClass                     # Class-specific metrics
```

11.4 Missing Data Handling

Missing Data Techniques

```r
# Check for missing values
sum(is.na(data))                # Total missing values
colSums(is.na(data))            # Missing values per column
complete_cases <- complete.cases(data)  # Logical vector of complete rows

# Simple imputation methods
# Mean imputation
data$x[is.na(data$x)] <- mean(data$x, na.rm = TRUE)

# Median imputation
data$x[is.na(data$x)] <- median(data$x, na.rm = TRUE)

# Model-based imputation
# Using regression
fit <- lm(x ~ y + z, data = data[!is.na(data$x),])
missing_indices <- which(is.na(data$x))
data$x[missing_indices] <- predict(fit, newdata = data[missing_indices,])

# Using mice package for multiple imputation
library(mice)
imputed_data <- mice(data, m = 5, method = "pmm")
completed_data <- complete(imputed_data)
```

11.5 ROC Analysis

ROC Curve Analysis

```r
library(pROC)

# For models that output probabilities directly
roc_obj <- roc(true_values, predicted_probs)

# For kNN (need to extract probabilities)
knn_pred <- knn(train_x, test_x, train_y, k = 5, prob = TRUE)
knn_probs <- attr(knn_pred, "prob")
roc_obj_knn <- roc(true_values, knn_probs)

# Plot ROC curves
plot(roc_obj, main = "ROC Curve")
plot(roc_obj_knn, add = TRUE, col = "blue")

# Get AUC
auc_value <- auc(roc_obj)

# Find optimal threshold
coords(roc_obj, "best")   # Best based on sensitivity + specificity

# Compare ROC curves
roc.test(roc_obj, roc_obj_knn)
```

11.6 Data Normalization/Scaling

Data Scaling Techniques

```
# Z-score standardization (mean=0, sd=1)
scaled_data <- scale(data)
# To apply same scaling to test data
test_scaled <- scale(test_data,
                     center = attr(scaled_data, "scaled:center"),
                     scale = attr(scaled_data, "scaled:scale"))

# Min-max normalization (range [0,1])
min_max_norm <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}
normalized_data <- as.data.frame(lapply(data, min_max_norm))

# Apply same normalization to test data
min_max_norm_test <- function(x, train_col) {
  return((x - min(train_col)) / (max(train_col) - min(train_col)))
}
# Apply to each column of test data using training data ranges
for(col in names(test_data)) {
  test_data[,col] <- min_max_norm_test(test_data[,col], data[,col])
}
```

11.7 Cross-Validation Types

Cross-Validation Methods

```
# K-fold cross-validation
ctrl_kfold <- trainControl(
  method = "cv",          # k-fold cross-validation
  number = 10,            # Number of folds
  savePredictions = TRUE
)

# Leave-One-Out Cross-Validation (LOOCV)
ctrl_loocv <- trainControl(
  method = "LOOCV"        # Each observation becomes a test set
)

# Repeated k-fold CV
ctrl_repeated <- trainControl(
  method = "repeatedcv",  # Repeated k-fold CV
  number = 10,            # Number of folds
  repeats = 5             # Number of complete repeats
)

# Bootstrap
ctrl_boot <- trainControl(
  method = "boot",        # Bootstrap resampling
  number = 50,            # Number of resamples
  savePredictions = TRUE
)
```

```r
# Stratified k-fold CV (maintains class proportions)
ctrl_strat <- trainControl(
  method = "cv",
  number = 10,
  savePredictions = TRUE,
  classProbs = TRUE,
  summaryFunction = twoClassSummary,
  sampling = "up"          # Upsampling for imbalanced data
)
```

11.8 Variable Importance Interpretation

Variable Importance Methods

```r
# Random Forest variable importance
rf_model <- randomForest(y ~ ., data = train_data)
importance(rf_model)                    # Mean decrease in accuracy/Gini
varImpPlot(rf_model)                    # Plot importance measures

# Understanding importance measures:
# 1. Mean Decrease Accuracy - Permutation importance
#    Higher values indicate more important features
#    Measures accuracy decrease when the variable is permuted

# 2. Mean Decrease Gini - Impurity-based importance
#    Higher values indicate features used for splitting more data
#    Measures average decrease in node impurity

# Caret's variable importance (model agnostic)
varImp(model)                           # For any caret model
plot(varImp(model))                     # Plot importance

# Linear model coefficients as importance
coef_importance <- abs(coef(lm_model))[-1]  # Remove intercept
coef_importance_scaled <- coef_importance/sum(coef_importance)*100
```

11.9 Advanced Ensemble Methods

Ensemble Method Comparison

```r
# Bagging: Bootstrap aggregating with all features
# Reduces variance, doesn't reduce bias
# All features considered at each split
p <- ncol(train_data) - 1               # Number of predictors
bagging_model <- randomForest(y ~ ., data = train_data,
                            mtry = p)  # Use all predictors

# Random Forest: Bagging + feature randomization
# Each tree considers subset of features
# Decorrelates trees, further reduces variance
rf_model <- randomForest(y ~ ., data = train_data,
                        mtry = sqrt(p))  # Default for classification

# Boosting: Sequential ensemble learning
# Focuses on errors of previous models
# Reduces bias and variance
boost_model <- gbm(y ~ ., data = train_data,
                  distribution = "bernoulli",  # For classification
                  n.trees = 100,
                  interaction.depth =.3,      # Tree complexity
                  shrinkage = 0.1,            # Learning rate
                  bag.fraction = 0.5)         # Stochastic boosting
```

11.10 Neural Network Configuration

Neural Network Parameters

```
library(nnet)

# Neural network with specific architecture
# For classification
model <- nnet(
  y ~ .,                                    # Formula
  data = train_data,
  size = 5,                                 # Hidden neurons
  decay = 0.01,                             # Weight decay (regularization)
  maxit = 1000,                             # Max iterations
  rang = 0.1,                               # Initial random weights range
  abstol = 1.0e-4,                          # Absolute tolerance
  reltol = 1.0e-8                           # Relative tolerance
)

# For regression
model <- nnet(
  y ~ .,                                    # Formula
  data = train_data,
  size = 5,                                 # Hidden neurons
  decay = 0.01,                             # Weight decay
  linout = TRUE                             # Linear output (for regression)
)

# Comparing architectures
sizes <- c(3, 5, 10, 15)
results <- list()

for(i in seq_along(sizes)) {
  model <- nnet(y ~ ., data = train_data, size = sizes[i],
               decay = 0.01, maxit = 1000, trace = FALSE)

  # Get predictions
  pred <- predict(model, newdata = test_data)

  # Calculate performance
  if(is.factor(test_data$y)) {  # Classification
    acc <- mean(pred == test_data$y)
    results[[i]] <- list(size = sizes[i], accuracy = acc)
  } else {  # Regression
    mse <- mean((pred - test_data$y)^2)
    results[[i]] <- list(size = sizes[i], mse = mse)
  }
}
```

11.11 Multiple Testing Correction

P-value Adjustment Methods

```
# Calculate p-values (e.g., from univariate tests)
p_values <- numeric(ncol(predictors))
for(i in 1:length(p_values)) {
  # Example: t-test or regression for each predictor
  test_result <- t.test(predictors[, i] ~ outcome)
  p_values[i] <- test_result$p.value
}

# Bonferroni correction (most conservative)
# Controls family-wise error rate (FWER)
p_bonferroni <- p.adjust(p_values, method = "bonferroni")
sig_bonferroni <- which(p_bonferroni < 0.05)

# Benjamini-Hochberg (BH) procedure
# Controls false discovery rate (FDR)
p_bh <- p.adjust(p_values, method = "BH")
sig_bh <- which(p_bh < 0.05)

# Benjamini-Yekutieli (BY) procedure
# Controls FDR under dependence
p_by <- p.adjust(p_values, method = "BY")
sig_by <- which(p_by < 0.05)

# Holm method
# Step-down version of Bonferroni
p_holm <- p.adjust(p_values, method = "holm")
sig_holm <- which(p_holm < 0.05)

# Hochberg method
# Step-up version of Bonferroni
p_hochberg <- p.adjust(p_values, method = "hochberg")
sig_hochberg <- which(p_hochberg < 0.05)
```

11.12 Decision Tree Visualizations

Tree Visualization and Interpretation

```
# Using tree package
library(tree)
tree_model <- tree(y ~ ., data = train_data)

# Basic tree plot
```

```
plot(tree_model)
text(tree_model, pretty = 0)

# Detailed node information
print(tree_model)            # Shows splits and node information
summary(tree_model)          # Summary statistics about the tree

# Tree pruning
cv_tree <- cv.tree(tree_model)
plot(cv_tree$size, cv_tree$dev, type = "b",
     xlab = "Tree Size", ylab = "Deviance")

# Find optimal tree size
opt_size <- cv_tree$size[which.min(cv_tree$dev)]
pruned_tree <- prune.tree(tree_model, best = opt_size)

# Compare before and after pruning
plot(pruned_tree)
text(pruned_tree, pretty = 0)

# Using rpart for prettier visualizations
library(rpart)
library(rpart.plot)
rpart_model <- rpart(y ~ ., data = train_data)
rpart.plot(rpart_model, extra = 1)  # extra=1 shows percentages

# Extracting decision rules
rules <- rpart.rules(rpart_model, extra = 4, cover = TRUE)
print(rules)
```

11.13 Performance Comparison Workflow

Multi-Model Comparison Workflow

```
# Set up resampling control
ctrl <- trainControl(
  method = "cv",
  number = 10,
  classProbs = TRUE,
  summaryFunction = twoClassSummary,
  savePredictions = "final"
)

# Define multiple models to compare
model_list <- list(
  # Linear methods
  LR = train(y ~ ., data = train_data, method = "glm", trControl = ctrl),
  Ridge = train(y ~ ., data = train_data, method = "glmnet",
             tuneGrid = expand.grid(alpha = 0, lambda = seq(0.0001, 1,
   length = 10)),
             trControl = ctrl),
  LASSO = train(y ~ ., data = train_data, method = "glmnet",
             tuneGrid = expand.grid(alpha = 1, lambda = seq(0.0001, 1,
   length = 10)),
             trControl = ctrl),
```

```r
  # Tree-based methods
  Tree = train(y ~ ., data = train_data, method = "rpart", trControl = ctrl),
  RF = train(y ~ ., data = train_data, method = "rf", trControl = ctrl),
  GBM = train(y ~ ., data = train_data, method = "gbm",
              verbose = FALSE, trControl = ctrl),

  # Other methods
  SVM = train(y ~ ., data = train_data, method = "svmRadial", trControl =
    ctrl),
  KNN = train(y ~ ., data = train_data, method = "knn", trControl = ctrl)
)

# Compare models
results <- resamples(model_list)
summary(results)

# Visualize comparison
dotplot(results, metric = "ROC")  # For classification
dotplot(results, metric = "RMSE") # For regression

# Statistical comparison of models
model_diffs <- diff(results)
summary(model_diffs)

# Final model performance on test set
predictions <- lapply(model_list, predict, newdata = test_data)
performance <- data.frame(
  Model = names(model_list),
  Accuracy = sapply(predictions, function(p) mean(p == test_data$y))
)

# Best model's details
best_model <- model_list[[which.max(performance$Accuracy)]]
varImp(best_model)
```

11.14 Model Selection Decision Framework

When to Use Which Model

Linear Models:

- **Use when:** Linear relationship between predictors and response; need interpretability; small dataset; low variance
- **Examples:** Linear/Logistic Regression, Ridge, LASSO

Tree-Based Models:

- **Use when:** Non-linear relationships; handling categorical variables; capturing interactions; need to rank variable importance
- **Examples:** Decision Trees, Random Forests, Gradient Boosting

SVMs:

- **Use when:** High-dimensional data; complex non-linear boundaries; moderate dataset size; good with irrelevant features

- **Examples:** Linear, Polynomial, Radial SVMs

Neural Networks:

- **Use when:** Complex non-linear patterns; large datasets; willing to sacrifice interpretability; features have complex interactions
- **Examples:** Single-layer NN, Multi-layer NN

k-NN:

- **Use when:** Simple implementation needed; non-parametric approach; small to moderate dataset size; feature importance not needed
- **Examples:** k-Nearest Neighbors

Dimensionality Reduction:

- **Use when:** High-dimensional data; multicollinearity issues; visualization needed; preprocessing step
- **Examples:** PCA, Feature Selection

Ensemble Methods:

- **Use when:** Need highest possible performance; willing to sacrifice some interpretability; have computational resources
- **Examples:** Bagging, Random Forests, Boosting

Model Selection Strategy:

1. Start simple (linear models)
2. Check performance via cross-validation
3. Try more complex models if needed
4. Compare performance and interpretability trade-offs
5. Ensure generalizability to new data

11.15 Quick Model Fitting Reference

Quick Model Fitting Cheatsheet

Linear/Logistic Regression:

```
# Linear Regression
lm_model <- lm(y ~ ., data = train_data)
lm_pred <- predict(lm_model, newdata = test_data)

# Logistic Regression
logit_model <- glm(y ~ ., data = train_data, family = "binomial")
logit_prob <- predict(logit_model, newdata = test_data, type = "response")
logit_class <- ifelse(logit_prob > 0.5, 1, 0)
```

Regularized Models:

```
# Ridge Regression
```

```
x_train <- model.matrix(y ~ . + 0, data = train_data)
x_test <- model.matrix(~ . + 0, test_data)
ridge_model <- glmnet(x_train, train_data$y, alpha = 0, lambda = 0.1)
ridge_pred <- predict(ridge_model, newx = x_test)

# LASSO
lasso_model <- glmnet(x_train, train_data$y, alpha = 1, lambda = 0.1)
lasso_pred <- predict(lasso_model, newx = x_test)

# Elastic Net
enet_model <- glmnet(x_train, train_data$y, alpha = 0.5, lambda = 0.1)
enet_pred <- predict(enet_model, newx = x_test)
```

Tree-Based Models:

```
# Decision Tree (tree package)
tree_model <- tree(y ~ ., data = train_data)
tree_pred <- predict(tree_model, newdata = test_data, type = "class") #
    classification
tree_pred <- predict(tree_model, newdata = test_data) # regression

# Decision Tree (rpart package)
rpart_model <- rpart(y ~ ., data = train_data)
rpart_pred <- predict(rpart_model, newdata = test_data, type = "class") #
    classification
rpart_pred <- predict(rpart_model, newdata = test_data) # regression

# Random Forest
rf_model <- randomForest(y ~ ., data = train_data, ntree = 500, mtry = sqrt(
    ncol(train_data)-1))
rf_pred <- predict(rf_model, newdata = test_data)
rf_prob <- predict(rf_model, newdata = test_data, type = "prob") #
    classification only

# Gradient Boosting
gbm_model <- gbm(y ~ ., data = train_data, distribution = "bernoulli", # for
    classification
                 n.trees = 100, interaction.depth = C3)
gbm_pred <- predict(gbm_model, newdata = test_data, n.trees = 100, type = "
    response")
```

SVMs:

```
# Linear SVM
svm_lin_model <- svm(y ~ ., data = train_data, kernel = "linear")
svm_lin_pred <- predict(svm_lin_model, newdata = test_data)

# Polynomial SVM
svm_poly_model <- svm(y ~ ., data = train_data, kernel = "polynomial", degree
     = 3)
svm_poly_pred <- predict(svm_poly_model, newdata = test_data)

# Radial SVM
svm_rad_model <- svm(y ~ ., data = train_data, kernel = "radial", gamma =
    0.1)
svm_rad_pred <- predict(svm_rad_model, newdata = test_data)

# SVM with probability output
```

```r
svm_prob_model <- svm(y ~ ., data = train_data, kernel = "radial",
    probability = TRUE)
svm_prob_pred <- predict(svm_prob_model, newdata = test_data, probability =
    TRUE)
probabilities <- attr(svm_prob_pred, "probabilities")
```

Neural Networks:

```r
# Classification Neural Network
nn_class_model <- nnet(y ~ ., data = train_data, size = 5, decay = 0.01,
    maxit = 1000)
nn_class_pred <- predict(nn_class_model, newdata = test_data, type = "class")
nn_class_prob <- predict(nn_class_model, newdata = test_data, type = "raw")

# Regression Neural Network
nn_reg_model <- nnet(y ~ ., data = train_data, size = 5, decay = 0.01, linout
     = TRUE)
nn_reg_pred <- predict(nn_reg_model, newdata = test_data)
```

Discriminant Analysis:

```r
# Linear Discriminant Analysis
lda_model <- lda(y ~ ., data = train_data)
lda_pred <- predict(lda_model, newdata = test_data)
lda_class <- lda_pred$class
lda_prob <- lda_pred$posterior

# Quadratic Discriminant Analysis
qda_model <- qda(y ~ ., data = train_data)
qda_pred <- predict(qda_model, newdata = test_data)
qda_class <- qda_pred$class
qda_prob <- qda_pred$posterior
```

k-Nearest Neighbors:

```r
# k-NN (class package)
knn_pred <- knn(train = train_data[, -1], test = test_data[, -1],
               cl = train_data$y, k = 5)

# k-NN with probabilities
knn_prob_pred <- knn(train = train_data[, -1], test = test_data[, -1],
                    cl = train_data$y, k = 5, prob = TRUE)
knn_probs <- attr(knn_prob_pred, "prob")
```

Dimension Reduction:

```r
# Principal Component Analysis
pca_model <- prcomp(train_data[, -1], center = TRUE, scale. = TRUE)
train_pca <- predict(pca_model, newdata = train_data[, -1])
test_pca <- predict(pca_model, newdata = test_data[, -1])

# Use first two components for visualization
plot(train_pca[, 1:2], col = train_data$y)
```

Survival Analysis:

```r
# Cox Proportional Hazards Model
cox_model <- coxph(Surv(time, status) ~ ., data = train_data)
cox_pred <- predict(cox_model, newdata = test_data, type = "risk")
cox_surv <- survfit(cox_model, newdata = test_data)
```

Caret Quick Models:

```r
# Basic trainControl object
ctrl <- trainControl(method = "cv", number = 5)

# Linear Model
lm_caret <- train(y ~ ., data = train_data, method = "lm", trControl = ctrl)
lm_caret_pred <- predict(lm_caret, newdata = test_data)

# Random Forest
rf_caret <- train(y ~ ., data = train_data, method = "rf", trControl = ctrl)
rf_caret_pred <- predict(rf_caret, newdata = test_data)

# SVM Radial
svm_caret <- train(y ~ ., data = train_data, method = "svmRadial", trControl
    = ctrl)
svm_caret_pred <- predict(svm_caret, newdata = test_data)

# Neural Network
nn_caret <- train(y ~ ., data = train_data, method = "nnet",
                  trace = FALSE, trControl = ctrl)
nn_caret_pred <- predict(nn_caret, newdata = test_data)

# k-NN
knn_caret <- train(y ~ ., data = train_data, method = "knn", trControl = ctrl
    )
knn_caret_pred <- predict(knn_caret, newdata = test_data)

# Gradient Boosting
gbm_caret <- train(y ~ ., data = train_data, method = "gbm",
                   verbose = FALSE, trControl = ctrl)
gbm_caret_pred <- predict(gbm_caret, newdata = test_data)
```

Recursive Feature Elimination:

```r
# RFE with Random Forest
rfe_ctrl <- rfeControl(functions = rfFuncs, method = "cv", number = 5)
rfe_result <- rfe(x = train_data[, -1], y = train_data[, 1],
                  sizes = 1:10, rfeControl = rfe_ctrl)
selected_vars <- rfe_result$optVariables
```

Model Evaluation:

```r
# Classification metrics
conf_mat <- table(predictions, test_data$y)
accuracy <- sum(diag(conf_mat))/sum(conf_mat)
sensitivity <- conf_mat[2,2]/sum(conf_mat[,2])  # True positive rate
specificity <- conf_mat[1,1]/sum(conf_mat[,1])  # True negative rate

# ROC curve
roc_obj <- roc(test_data$y, predicted_probs)
auc_value <- auc(roc_obj)
plot(roc_obj)

# Regression metrics
mse <- mean((predictions - test_data$y)^2)
rmse <- sqrt(mse)
mae <- mean(abs(predictions - test_data$y))
r_squared <- cor(predictions, test_data$y)^2
```

11.16 Simple ROC Curve Generation

Quick ROC Curve Analysis

Basic ROC Curve for a Single Model:

```r
library(pROC)

# For any model with probability output:
model_probs <- predict(model, newdata = test_data, type = "prob")[,2]  # Get
    probabilities
roc_obj <- roc(test_data$outcome, model_probs)  # Create ROC object
plot(roc_obj)  # Plot ROC curve
auc(roc_obj)   # Get AUC value
```

By Model Type (Common Examples):

```r
# Logistic Regression
logit_probs <- predict(logit_model, newdata = test_data, type = "response")
roc(response = test_data$outcome, predictor = logit_probs)$auc
plot(roc(response = test_data$outcome, predictor = logit_probs))

# Random Forest
rf_probs <- predict(rf_model, newdata = test_data, type = "prob")[,2]
roc(response = test_data$outcome, predictor = rf_probs)$auc
plot(roc(response = test_data$outcome, predictor = rf_probs))

# SVM (need probability = TRUE during fitting)
svm_pred <- predict(svm_model, newdata = test_data, probability = TRUE)
svm_probs <- attr(svm_pred, "probabilities")[,2]
roc(response = test_data$outcome, predictor = svm_probs)$auc
plot(roc(response = test_data$outcome, predictor = svm_probs))

# LDA
lda_pred <- predict(lda_model, newdata = test_data)
lda_probs <- lda_pred$posterior[,2]
roc(response = test_data$outcome, predictor = lda_probs)$auc
plot(roc(response = test_data$outcome, predictor = lda_probs))

# Neural Network (nnet)
nn_probs <- predict(nn_model, newdata = test_data, type = "raw")
roc(response = test_data$outcome, predictor = nn_probs)$auc
plot(roc(response = test_data$outcome, predictor = nn_probs))

# Regularized Models (glmnet)
ridge_probs <- predict(ridge_model, newx = x_test, type = "response", s =
    lambda)
roc(response = test_data$outcome, predictor = as.vector(ridge_probs))$auc
plot(roc(response = test_data$outcome, predictor = as.vector(ridge_probs)))
```

11.17 Exam Day Quick Reference

Exam Day Checklist

Before Starting:

- Set seed for reproducibility: `set.seed(XXXX)`
- Check for missing values: `sum(is.na(data))`
- Check data structure: `str(data)`
- Verify data dimensions: `dim(data)`

Common Errors to Avoid:

- Not setting random seed before sampling/fitting
- Forgetting to handle missing values
- Not converting categorical variables to factors
- Applying models that require numeric data to categorical features
- Not scaling features for distance-based methods
- Forgetting to specify `type="response"` for probabilities
- Using random splits for time series data

Fast Copy-Paste Code Snippets:

```r
# Data splitting
set.seed(XXXX)
train_idx <- sample(1:nrow(data), 0.7*nrow(data))
train_data <- data[train_idx, ]
test_data <- data[-train_idx, ]

# Confusion matrix
conf_mat <- table(predictions, test_data$y)
accuracy <- sum(diag(conf_mat))/sum(conf_mat)

# Quick RF
model <- randomForest(y ~ ., data = train_data)
pred <- predict(model, newdata = test_data)

# Quick SVM
model <- svm(y ~ ., data = train_data, kernel = "radial")
pred <- predict(model, newdata = test_data)

# Quick neural net
model <- nnet(y ~ ., data = train_data, size = 5, decay = 0.01)
pred <- predict(model, newdata = test_data, type = "class")
```