Machine Learning Cheat Sheet

1. Data Pre-Processing

Scaling Data

Scaling for ML algorithms when features have different units or ranges.

Methods of Scaling in R

```r
# Standard scaling (mean=0, sd=1)
scaled_data = scale(data)

# Apply scaling to specific columns
df[,1:4] = apply(df[,1:4], 2, scale)

# In PCA
pca.unscaled = prcomp(x)
pca.scaled = prcomp(x, scale=TRUE)
```

Effects of scaling:

- **PCA**: Variables with larger variances dominate unscaled PCA; scaling ensures equal contribution
- **Classification**: Scaling affects coefficient magnitudes in logistic regression
- **Regularization**: Required for Lasso/Ridge (penalty applies equally to all features)

Data Imputation

Missing data handling for model performance:

Imputation Methods

```r
# 1. Deletion
clean_data = na.omit(data)

# 2. Mean imputation (simple)
data$var[is.na(data$var)] = mean(data$var, na.rm=TRUE)

# 3. Group-specific imputation
# For gender-specific means:
m_mean = mean(data$var[data$sex==1], na.rm=TRUE)
f_mean = mean(data$var[data$sex==2], na.rm=TRUE)

# Apply to missing values by group
data$var[is.na(data$var) & data$sex==1] = m_mean
data$var[is.na(data$var) & data$sex==2] = f_mean
```

Missing data mechanisms:

- **MCAR**: Missing Completely At Random
- **MAR**: Missing At Random (depends on observed data)
- **MNAR**: Missing Not At Random (depends on missing values)

Check if group-specific imputation is needed (with t-tests comparing groups).

Resampling Methods

Resampling to estimate model performance and uncertainty.

Cross-Validation

```r
# K-fold Cross-Validation
set.seed(4060)
k = 10
n = nrow(data)
folds = sample(rep(1:k, length.out = n))
results = numeric(k)

for(i in 1:k){
  test_ind = which(folds == i)
  train_data = data[-test_ind, ]
  test_data = data[test_ind, ]
  model = lm(y ~ x, data=train_data)
  # Store metric (e.g., RMSE, coefficient)
  results[i] = ...
}
cv_estimate = mean(results)
```

Shuffling Datasets

```r
# Shuffle dataset rows randomly
set.seed(123)
shuffled_data = data[sample(1:nrow(data)), ]

# Shuffle specific vectors
x_shuffled = x[sample(1:length(x))]
y_shuffled = y[sample(1:length(y))]
```

Bootstrapping

```r
# Bootstrapping
set.seed(4060)
B = 100
results = numeric(B)
N = nrow(data)

for(i in 1:B){
  # Sample with replacement
  boot_indices = sample(1:N, N, replace=TRUE)
  boot_data = data[boot_indices, ]
  model = lm(y ~ x, data=boot_data)
  # Store result
  results[i] = ...
}
boot_estimate = mean(results)
```

Key differences:

- CV uses all data points exactly once (disjoint test sets)
- Bootstrap samples with replacement (some observations appear multiple times, others not at all)
- Bootstrap better for uncertainty estimation, CV better for performance estimation
- Bootstrap can estimate out-of-bag (OOB) error using non-selected observations

Considerations:

- Randomizing data before CV affects results in ordered datasets
- Bootstrapping tends to be more optimistic about error rates
- Both methods can be used for hyperparameter tuning
- **Shuffled datasets**: When comparing original vs. shuffled datasets:
  - Point estimates may be similar (e.g., mean coefficient values)
  - Variance can differ due to inherent data structure
  - Statistical tests (t-test, F-test) can be used to compare distributions of estimates

2. Regularization

Ridge and LASSO

Regularization Methods

```r
# Ridge regression (L2 penalty, alpha=0)
ridge = glmnet(x, y, alpha=0, lambda=lambda_values)

# LASSO regression (L1 penalty, alpha=1)
lasso = glmnet(x, y, alpha=1, lambda=lambda_values)

# Elastic Net (both L1 and L2, 0<alpha<1)
elasticnet = glmnet(x, y, alpha=0.5, lambda=lambda_values)
```

Differences:

- **Ridge**: Shrinks coefficients toward zero, none set exactly to zero
- **LASSO**: Performs variable selection by setting some coefficients exactly to zero
- **Lambda**: Controls penalty strength (higher λ = more shrinkage)

Parameter Tuning

Cross-Validation for Lambda Selection

```
# Create lambda sequence
lambda_grid = 10^seq(10, -2, length=100)

# Cross-validation for optimal lambda
cv_ridge = cv.glmnet(X_train, y_train, alpha=0,
                     lambda=lambda_grid)
best_lambda_ridge = cv_ridge$lambda.min

# Alternative: 1SE rule for sparser models
lambda_1se = cv_ridge$lambda.1se
```

Implementation

Model Fitting and Prediction

```
# Data preparation
x = model.matrix(Salary~., data=dat)[,-1]
y = dat$Salary

# Fitting final model with optimal lambda
ridge_model = glmnet(X_train, y_train, alpha=0,
                     lambda=best_lambda_ridge)

# Getting coefficients
coef_ridge = predict(ridge_model,
                     type="coefficients",
                     s=best_lambda_ridge)

# Making predictions
pred_ridge = predict(ridge_model,
                     s=best_lambda_ridge,
                     newx=X_test)
```

Notes:

- The `s` parameter in `predict()` specifies the lambda value(s) for which to make predictions or extract coefficients
- `s=best_lambda_min` - the lambda that minimizes CV error
- `s=lambda.1se` - the largest lambda within 1 standard error of minimum (sparser model)
- `s` can be a vector, returning predictions for multiple lambda values

Model Evaluation

Train/Test Split and Evaluation

```
# Train/test split (70/30)
n = nrow(data)
train_idx = sample(1:n, size=0.7*n)
train = data[train_idx, ]
test = data[-train_idx, ]

# Model matrices for glmnet
X_train = model.matrix(y~., train)[,-1]
X_test = model.matrix(y~., test)[,-1]

# Evaluation metric: MSE
mse = function(actual, pred) mean((actual-pred)^2)
ridge_mse = mse(y_test, ridge_pred)
lasso_mse = mse(y_test, lasso_pred)
ols_mse = mse(y_test, ols_pred)
```

Effect Comparison

- Ridge keeps all variables but reduces their influence
- LASSO performs feature selection, eliminating some variables
- OLS can overfit with high-dimensional data
- For correlated predictors:
  - Ridge tends to shrink correlated predictors together
  - LASSO tends to pick one from a correlated group

3. Classification

K-Nearest Neighbors (KNN)

Non-parametric method that classifies new data points based on the majority vote of K nearest neighbors.

KNN Implementation

```r
# KNN classification
library(class)

# Split data
i.train = sample(1:n, 100)
x.train = x[i.train, ]
x.test = x[-i.train, ]
y.train = y[i.train]
y.test = y[-i.train]

# Run with K=5
k = 5
knn_pred = knn(x.train, x.test, y.train, k)

# Evaluate
conf_matrix = table(knn_pred, y.test)
accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)

# Find optimal K
Kmax = 30
acc = numeric(Kmax)
for(k in 1:Kmax){
  knn_pred = knn(x.train, x.test, y.train, k)
  conf_matrix = table(knn_pred, y.test)
  acc[k] = sum(diag(conf_matrix)) / sum(conf_matrix)
}
plot(1-acc, type='b', xlab='k')
```

Notes:

- For probabilities: `knn(..., prob=TRUE)`
- Access probability with `attributes(knn_pred)$prob`
- Scale features before KNN to avoid domination by variables with larger ranges
- `prob` attribute returns proportion of votes for winning class, not P(Y=1—X)

Logistic Regression

Parametric method modeling log-odds of class membership as a linear function of predictors.

Logistic Regression Implementation

```r
# Logistic regression
glm_model = glm(response ~ predictors,
                data=train_data,
                family=binomial(logit))

# Predictions (probabilities)
pred_prob = predict(glm_model, newdata=test_data,
                    type="response")

# Class predictions with threshold 0.5
pred_class = ifelse(pred_prob > 0.5, 1, 0)

# Confusion matrix
library(caret)
cm = confusionMatrix(
  data=as.factor(pred_class),
  reference=as.factor(test_data$response),
  positive="1"
)
```

Threshold Selection:

Testing Different Thresholds

```r
# Test different thresholds
thresholds = seq(0.1, 0.9, by=0.1)
sensitivity = numeric(length(thresholds))

for(i in 1:length(thresholds)){
  pred_class = ifelse(pred_prob > thresholds[i], 1, 0)
  cm = confusionMatrix(
    data=as.factor(pred_class),
    reference=as.factor(test_data$response),
    positive="1"
  )
  sensitivity[i] = cm$byClass["Sensitivity"]
}
plot(thresholds, sensitivity, type="b")
```

Discriminant Analysis

LDA and QDA Implementation

```r
library(MASS)

# Linear Discriminant Analysis
lda_model = lda(Species ~ ., data=train_data)
lda_pred = predict(lda_model, newdata=test_data)
lda_class = lda_pred$class
lda_post = lda_pred$posterior  # class probabilities

# Quadratic Discriminant Analysis
qda_model = qda(Species ~ ., data=train_data)
qda_pred = predict(qda_model, newdata=test_data)
```

LDA Assumptions Testing:

Testing Normality and Equal Variance

```r
# Equal variance: Bartlett's test
# H0: All group variances are equal
for(j in 1:ncol(predictors)){
  bartlett.test(data[,j] ~ data$group)
}

# Normality: Shapiro-Wilk test
# H0: Data follows normal distribution
for(j in 1:ncol(predictors)){
  shapiro.test(data[data$group=="A", j])
  shapiro.test(data[data$group=="B", j])
}

# Visual checks: Normal QQ plots
for(j in 1:ncol(predictors)){
  x1 = data[data$group=="A", j]
  qqnorm(x1, main=names(data)[j])
  qqline(x1)
}
```

Key Differences:

- **LDA**: Assumes equal covariance matrices across classes
- **QDA**: Allows different covariance matrices for each class
- LDA creates linear decision boundaries, QDA creates quadratic ones
- QDA more flexible but requires more parameters to estimate

Model Evaluation

Classification Metrics

```
# Confusion matrix-based metrics
TP = conf_matrix[2,2]  # True positives
TN = conf_matrix[1,1]  # True negatives
FP = conf_matrix[2,1]  # False positives
FN = conf_matrix[1,2]  # False negatives

accuracy = (TP + TN) / (TP + TN + FP + FN)
sensitivity = TP / (TP + FN)  # Recall
specificity = TN / (TN + FP)
precision = TP / (TP + FP)
F1 = 2 * precision * sensitivity / (precision + sensitivity)
```

ROC Analysis

```
library(pROC)

# ROC curve and AUC
roc_obj = roc(response, predicted_probs)
auc_value = auc(roc_obj)
plot(roc_obj)

# Manual ROC curve
thresholds = seq(0.05, 0.95, by=0.05)
sensitivity = specificity = numeric(length(thresholds))

for(i in 1:length(thresholds)){
  pred_class = ifelse(pred_prob > thresholds[i], 1, 0)
  cm = table(pred_class, true_class)

  # True positive rate (Sensitivity)
  sensitivity[i] = cm[2,2] / sum(cm[,2])

  # True negative rate (Specificity)
  specificity[i] = cm[1,1] / sum(cm[,1])
}

# Plot ROC curve
plot(1-specificity, sensitivity, type="l")
abline(0, 1, lty=2)  # Diagonal reference line
```

ROC Curves and AUC in Depth

ROC (Receiver Operating Characteristic) curves plot sensitivity vs. 1-specificity across all possible classification thresholds.

Understanding ROC and AUC

Key Concepts:

- **ROC curve** plots True Positive Rate (Sensitivity) against False Positive Rate (1-Specificity)
- **AUC** (Area Under Curve): Probability a randomly selected positive instance ranks higher than a randomly selected negative instance
- AUC = 0.5: No discriminative power (random classifier)
- AUC = 1.0: Perfect classifier
- AUC = 0.7-0.8: Acceptable discrimination
- AUC = 0.8-0.9: Excellent discrimination
- AUC ¿ 0.9: Outstanding discrimination

Calculating and Comparing AUC

```
library(pROC)

# Calculate AUC for multiple models
glm_probs = predict(glm_model, newdata=test_data, type="response")
lda_probs = predict(lda_model, newdata=test_data)$posterior[,2]
qda_probs = predict(qda_model, newdata=test_data)$posterior[,2]

# Compute ROC objects
roc_glm = roc(test_data$response, glm_probs)
roc_lda = roc(test_data$response, lda_probs)
roc_qda = roc(test_data$response, qda_probs)

# Get AUC values
auc_glm = auc(roc_glm)
auc_lda = auc(roc_lda)
auc_qda = auc(roc_qda)

# Compare models
print(auc_glm)
print(auc_lda)
print(auc_qda)

# Plot multiple ROC curves
plot(roc_glm, col="red")
plot(roc_lda, col="blue", add=TRUE)
plot(roc_qda, col="green", add=TRUE)
legend("bottomright", legend=c("GLM", "LDA", "QDA"),
       col=c("red", "blue", "green"), lwd=2)
```

Optimal Threshold Selection

```
# Find optimal threshold using different methods

# 1. Youden's J statistic (maximizes sensitivity+specificity-1)
roc_obj = roc(response, predicted_probs)
coords = coords(roc_obj, "best", ret="threshold", best.method="youden")
best_threshold = coords[1]

# 2. Closest point to (0,1) on ROC curve
coords = coords(roc_obj, "best", ret="threshold",
                best.method="closest.topleft")
best_threshold = coords[1]

# 3. Cost-based approach (when FP and FN have different costs)
# Example: FN costs 3x more than FP
cost_ratio = 3  # cost(FN)/cost(FP)
weights = c(1, cost_ratio)
coords = coords(roc_obj, "best", ret="threshold",
                best.method="youden", best.weights=weights)
best_threshold = coords[1]

# Apply optimal threshold
optimal_preds = ifelse(predicted_probs > best_threshold, 1, 0)
```

ROC for Multi-class Problems

```r
# For multi-class problems: One-vs-All (OVA) approach
library(pROC)

# Assuming 3 classes in posterior probabilities
class_labels = c("A", "B", "C")
true_labels = test_data$class  # factor with levels A, B, C
posterior_probs = predict(model, newdata=test_data)$posterior

# Calculate ROC for each class
auc_values = numeric(length(class_labels))

for(i in 1:length(class_labels)){
  # Create binary problem: class i vs rest
  binary_true = ifelse(true_labels == class_labels[i], 1, 0)

  # Get probability for this class
  class_prob = posterior_probs[,i]

  # Calculate ROC and AUC
  roc_obj = roc(binary_true, class_prob)
  auc_values[i] = auc(roc_obj)

  # Plot ROC curve
  if(i == 1) plot(roc_obj, col=i)
  else plot(roc_obj, col=i, add=TRUE)
}

# Average AUC across all classes (macro-averaging)
mean_auc = mean(auc_values)
```

Cross-Validated AUC

```r
# K-fold cross-validated AUC
set.seed(123)
K = 10
n = nrow(data)
folds = cut(1:n, K, labels=FALSE)
cv_auc = numeric(K)

for(k in 1:K){
  # Split into train and test
  train_idx = which(folds != k)
  test_idx = which(folds == k)

  # Train model
  model = glm(y ~ ., data=data[train_idx,], family=binomial)

  # Make predictions on test fold
  probs = predict(model, data[test_idx,], type="response")

  # Calculate AUC for this fold
  if(length(unique(data$y[test_idx])) > 1) {
    # Only calculate if both classes present in test set
    roc_obj = roc(data$y[test_idx], probs)
    cv_auc[k] = auc(roc_obj)
  }
}

# Mean and SD of cross-validated AUC
mean_cv_auc = mean(cv_auc, na.rm=TRUE)
sd_cv_auc = sd(cv_auc, na.rm=TRUE)
```

Common ROC/AUC Pitfalls

Common Issues:

- Using AUC on highly imbalanced data without appropriate sampling
- Calculating AUC when KNN returns winning class probability (`attributes(knn_pred)$prob`) instead of class membership probability
- Using default threshold (0.5) when class distributions are imbalanced
- Not considering costs of misclassification in threshold selection
- ROC curves can be overly optimistic on small datasets

Solutions:

- Cross-validate AUC estimates to improve reliability
- Consider precision-recall curves for imbalanced data
- When using KNN for ROC curves, class probabilities must be calculated differently than standard output
- Use domain knowledge to set appropriate thresholds for sensitivity/specificity trade-offs

Cross-Validation for Model Comparison

K-fold CV for Model Comparison

```r
# K-fold CV to compare classification models
set.seed(123)
K = 10
folds = cut(1:nrow(data), K, labels=FALSE)
acc.knn = acc.glm = acc.lda = acc.qda = numeric(K)
auc.knn = auc.glm = auc.lda = auc.qda = numeric(K)

for(k in 1:K){
  # Split data
  train_idx = which(folds != k)
  test_idx = which(folds == k)
  train_data = data[train_idx, ]
  test_data = data[test_idx, ]

  # Fit models
  knn_pred = knn(train_data[,-1], test_data[,-1],
                 train_data[,1], k=5)
  glm_model = glm(y~., data=train_data, family=binomial)
  lda_model = lda(y~., data=train_data)
  qda_model = qda(y~., data=train_data)

  # Make predictions
  glm_prob = predict(glm_model, test_data, type="response")
  lda_prob = predict(lda_model, test_data)$posterior[,2]
  qda_prob = predict(qda_model, test_data)$posterior[,2]

  # Calculate metrics
  acc.knn[k] = mean(knn_pred == test_data[,1])
  acc.glm[k] = mean((glm_prob > 0.5) == test_data[,1])
  acc.lda[k] = mean(predict(lda_model, test_data)$class
                    == test_data[,1])
  acc.qda[k] = mean(predict(qda_model, test_data)$class
                    == test_data[,1])

  # AUC values
  auc.glm[k] = auc(roc(test_data[,1], glm_prob))
  auc.lda[k] = auc(roc(test_data[,1], lda_prob))
  auc.qda[k] = auc(roc(test_data[,1], qda_prob))
}

# Compare performances
par(mfrow=c(1,2))
boxplot(acc.knn, acc.glm, acc.lda, acc.qda,
        names=c("KNN", "LogReg", "LDA", "QDA"))
boxplot(auc.glm, auc.lda, auc.qda,
        names=c("LogReg", "LDA", "QDA"))
```

4. Tree-Based Methods

Decision Trees

Decision trees partition the feature space into a set of rectangles and then fit a simple model (like a constant) in each one. Trees are highly interpretable and can capture complex non-linear relationships.

Key Components:

- **Node:** Each decision point in the tree
- **Root:** Top node where the tree begins
- **Splitting:** Process of dividing a node based on a feature and threshold value
- **Terminal Node/Leaf:** Nodes that don't split further, contain final predictions
- **Path:** Sequence of nodes from root to leaf

Splitting Criteria:

- **Classification:** Gini impurity or entropy (information gain)
- **Regression:** Reduction in residual sum of squares (RSS)

Growing a Classification Tree

```r
library(tree)
library(ISLR)

# Prepare data (binary classification example)
High = ifelse(Carseats$Sales <= 8, "No", "Yes")
CS = data.frame(Carseats, High=as.factor(High))
CS$Sales = NULL  # Remove original response

# Grow the tree
tree_model = tree(High ~ ., data=CS)
summary(tree_model)

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

Reading a Tree: To make a prediction, follow the path from root to leaf. At each internal node, check the condition (e.g., "Shelf Location ¡ Good"): if TRUE, go left; if FALSE, go right. The leaf node contains the predicted value.

Advantages and Disadvantages:

- **Pros:** Interpretable, handles mixed data types, automatically captures interactions
- **Cons:** High variance (small changes in data can produce very different trees), tend to overfit

Tree Pruning

Pruning reduces overfitting by removing branches that add little predictive power. It's a crucial step for single trees to improve generalization.

Cost-Complexity Pruning: The algorithm grows a large tree and then prunes it back based on a cost-complexity parameter :

- For each value of , find the subtree that minimizes: RSS + × (number of terminal nodes)
- As increases, we get smaller trees
- Cross-validation is used to select the optimal

Pruning Trees with Cross-Validation

```r
# Cross-validation for optimal tree size
cv_tree = cv.tree(tree_model, FUN=prune.misclass)

# Plot CV error vs. tree size
plot(cv_tree$size, cv_tree$dev, type="b",
     xlab="Tree Size", ylab="CV Error")
abline(v=cv_tree$size[which.min(cv_tree$dev)])

# Get optimal size and prune
opt_size = cv_tree$size[which.min(cv_tree$dev)]
pruned_tree = prune.misclass(tree_model, best=opt_size)

# Compare original and pruned trees
par(mfrow=c(1,2))
plot(tree_model); text(tree_model, pretty=0)
plot(pruned_tree); text(pruned_tree, pretty=0)
```

Pruning Functions:

- `prune.misclass()`: Prunes based on misclassification error (for classification trees)
- `prune.tree()`: Prunes based on deviance (for regression trees)
- `cv.tree()`: Cross-validation to find optimal tree size (parameter `FUN` specifies the pruning criterion)

Pruning Effects: Pruning typically increases training error but decreases test error by reducing overfitting. The optimal tree size balances bias and variance.

Tree Prediction and Evaluation

Trees can output both class predictions and class probabilities for classification problems, allowing for threshold adjustments and ROC analysis.

Tree Prediction and Evaluation

```r
# Train/test split
set.seed(123)
train_idx = sample(1:nrow(CS), 200)
train_data = CS[train_idx,]
test_data = CS[-train_idx,]

# Build tree on training data
tree_model = tree(High ~ ., data=train_data)

# Class predictions
class_pred = predict(tree_model, test_data, type="class")
conf_matrix = table(class_pred, test_data$High)
accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
print(conf_matrix)
cat("Accuracy:", accuracy, "\n")

# Probability predictions for ROC analysis
prob_pred = predict(tree_model, test_data, type="vector")
library(pROC)
roc_obj = roc(response=test_data$High, predictor=prob_pred[,1])
auc_value = auc(roc_obj)
plot(roc_obj)
```

Tree Prediction Types:

- `type="class"`: Returns predicted classes (factors)
- `type="vector"`: Returns class probabilities (for classification trees)
- `type="matrix"`: Returns a matrix of class probabilities
- Default: Returns predicted values (mean response in each leaf)

Evaluating Classification Trees:

- **Accuracy:** Proportion of correct predictions
- **Sensitivity:** True Positive Rate = TP/(TP+FN)
- **Specificity:** True Negative Rate = TN/(TN+FP)
- **AUC:** Area under ROC curve, measures discriminative ability across thresholds

Random Forests and Bagging

Random Forests reduce variance by averaging multiple decision trees, each trained on bootstrapped samples of data with random feature subsets considered at each split.

Bagging vs. Random Forests:

- **Bagging:** Bootstrap Aggregation, uses all features at each split
- **Random Forests:** Only considers a random subset of features at each split
- RF creates more diverse trees, reducing correlation between trees
- Bagging is equivalent to RF with mtry = p (total number of predictors)

Key Parameters in Random Forests:

- **ntree:** Number of trees in the forest (higher = more stable, but diminishing returns)
- **mtry:** Number of variables randomly sampled at each split
  - Default for classification: $\sqrt{p}$
  - Default for regression: $p/3$
- **nodesize:** Minimum size of terminal nodes (controls tree complexity)

Random Forest and Bagging Implementation

```r
library(randomForest)

# Training data preparation
set.seed(123)
train_idx = sample(1:nrow(CS), 200)
train_data = CS[train_idx,]
test_data = CS[-train_idx,]

# Number of predictors
p = ncol(train_data) - 1

# Random Forest (mtry defaults to sqrt(p) for classification)
rf_model = randomForest(High ~ .,
                        data=train_data,
                        ntree=500,  # number of trees
                        importance=TRUE)  # compute variable importance

# Bagging (special case of RF with mtry=p)
bag_model = randomForest(High ~ .,
                         data=train_data,
                         mtry=p,  # use all predictors
                         importance=TRUE)

# Print models to see OOB error
print(rf_model)
print(bag_model)

# Predictions
rf_pred = predict(rf_model, test_data)
bag_pred = predict(bag_model, test_data)

# Evaluation
rf_conf = table(rf_pred, test_data$High)
bag_conf = table(bag_pred, test_data$High)

# Accuracy
rf_acc = sum(diag(rf_conf)) / sum(rf_conf)
bag_acc = sum(diag(bag_conf)) / sum(bag_conf)
cat("RF Accuracy:", rf_acc, "\n")
cat("Bagging Accuracy:", bag_acc, "\n")
```

Out-of-Bag (OOB) Error: Since each tree is built on a bootstrap sample (about 63% of the original data), the remaining 37% can serve as a validation set. This OOB error is a built-in cross-validation mechanism in random forests.

OOB calculations:

- For each observation, only trees where that observation was OOB are used for prediction
- These predictions are aggregated to get the OOB prediction
- OOB error rate is the error rate of these predictions

Variable Importance

Variable importance measures in random forests help identify which features contribute most to prediction accuracy. Two main measures are provided:

- **Mean Decrease in Accuracy:** Average decrease in accuracy when a variable is permuted
- **Mean Decrease in Gini:** Average decrease in node impurity by splits on a variable

Variable Importance

```r
# Variable importance measures
rf_importance = importance(rf_model)
print(rf_importance)

# Plot variable importance
varImpPlot(rf_model, main="Random Forest Variable Importance",
           type=1)  # type=1 for Mean Decrease Accuracy

# Compare variable importance between RF and Bagging
par(mfrow=c(1,2))
varImpPlot(rf_model, main="RF Importance")
varImpPlot(bag_model, main="Bagging Importance")

# Partial dependence plots (effect of predictors)
partialPlot(rf_model, train_data, x.var="Price")
```

Interpreting Variable Importance:

- Higher values indicate more important variables
- Mean Decrease Accuracy directly measures predictive power
- Mean Decrease Gini measures node purity improvement
- Variables with zero importance can often be removed without affecting performance

Partial Dependence Plots: Show the marginal effect of a variable on the prediction after accounting for the average effects of all other variables.

Gradient Boosting

Boosting builds an ensemble sequentially, where each tree tries to correct the errors of the previous trees. While random forests reduce variance through averaging independent trees, boosting reduces both bias and variance by iteratively focusing on difficult examples.

Gradient Boosting Algorithm:

1. Initialize prediction with a constant value (mean for regression, log-odds for classification)
2. For m = 1 to M (number of trees):
   (a) Compute residuals (difference between actual values and current predictions)
   (b) Fit a weak learner (shallow tree) to the residuals
   (c) Update predictions by adding a shrunk version of the new tree's predictions
3. Return the final ensemble model

Key Parameters:

- **n.trees:** Number of boosting iterations (trees)
- **interaction.depth:** Maximum depth of each tree (1 = stumps)
- **shrinkage:** Learning rate (how much each tree contributes)
- **bag.fraction:** Fraction of data used for each iteration (introduces randomness)

Gradient Boosting

```r
library(gbm)

# For binary classification
# Convert response to numeric 0/1
train_data_gbm = train_data
train_data_gbm$High = as.numeric(train_data$High == "Yes")

# Gradient Boosting Machine
gb_model = gbm(High ~ .,
               data=train_data_gbm,
               distribution="bernoulli", # for classification
               n.trees=5000,             # ensemble size
               interaction.depth=2,      # tree depth (1=stumps)
               shrinkage=0.01,           # learning rate
               cv.folds=5)               # for CV

# Find optimal number of trees
best_iter = gbm.perf(gb_model)
print(paste("Optimal number of trees:", best_iter))

# Variable importance
summary(gb_model)

# Partial dependence plots
plot(gb_model, i="Price")

# Predictions
test_data_gbm = test_data
gb_prob = predict(gb_model,
                  newdata=test_data_gbm,
                  n.trees=best_iter,
                  type="response")  # probabilities

# Convert to class predictions
gb_class = ifelse(gb_prob > 0.5, "Yes", "No")

# Confusion matrix
conf_matrix = table(gb_class, test_data$High)
accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
print(conf_matrix)
cat("Accuracy:", accuracy, "\n")
```

Boosting Distributions:

- **gaussian:** For regression problems
- **bernoulli:** For binary classification
- **adaboost:** AdaBoost algorithm for classification
- **poisson:** For count data

Boosting vs. Random Forests:

- RF builds independent trees in parallel; Boosting builds sequential trees
- RF typically requires less parameter tuning than Boosting
- Boosting often achieves higher accuracy but is more prone to overfitting
- Lower learning rates in Boosting require more trees but often give better performance

Trees as Piecewise Models

Trees can be viewed as creating piecewise constant (or piecewise linear) functions. Each leaf represents a region where a constant prediction is made.

Trees as Piecewise Models

```r
# Fit a tree to continuous data
data = data.frame(x=x, y=y)
tree_model = tree(y ~ x, data=data)

# Extract split points
split_text = tree_model$frame$splits[,1]
split_values = as.numeric(na.omit(
  unlist(strsplit(split_text, "<"))
))
cut_points = c(min(x), sort(split_values), max(x))

# Function to demonstrate piecewise prediction
plot(x, y, pch=20)
for(i in 1:(length(cut_points)-1)) {
  subset_idx = which(x >= cut_points[i] &
                     x < cut_points[i+1])
  if(length(subset_idx) > 0) {
    # Fit a model in each region
    region_model = lm(y ~ x, data=data[subset_idx,])

    # Get predictions at region boundaries
    pred_x = c(cut_points[i], cut_points[i+1])
    pred_y = predict(region_model,
                     newdata=data.frame(x=pred_x))

    # Draw line segment
    segments(x0=pred_x[1], x1=pred_x[2],
             y0=pred_y[1], y1=pred_y[2],
             col="blue", lwd=2)
  }
}
```

Key Concepts in Piecewise Models:

- Each leaf in a tree corresponds to a region in the feature space
- Standard trees fit constant models in each region (piecewise constant)
- We can extend this to fit linear (or other) models in each region
- This approach can lead to smoother predictions and better performance

Model Comparison

In practice, it's essential to compare different tree-based methods to select the best approach for a specific problem.

Benchmarking Tree Methods

```r
# Compare tree-based methods with other classifiers
set.seed(123)
K = 10  # Number of folds
folds = cut(1:nrow(data), K, labels=FALSE)

# Accuracy storage
acc_tree = acc_rf = acc_gbm = acc_glm = numeric(K)
auc_tree = auc_rf = auc_gbm = auc_glm = numeric(K)

for(k in 1:K) {
  # Split data
  train_idx = which(folds != k)
  test_idx = which(folds == k)
  train_data = data[train_idx, ]
  test_data = data[test_idx, ]

  # Fit models
  tree_model = tree(response ~ ., data=train_data)
  rf_model = randomForest(response ~ ., data=train_data)

  # Convert response for GBM if needed
  train_data_gbm = train_data
  if(is.factor(train_data$response)) {
    train_data_gbm$response = as.numeric(train_data$response == levels(train_data$
    response)[2])
  }

  gbm_model = gbm(response ~ ., data=train_data_gbm,
                  distribution="bernoulli", n.trees=1000,
                  interaction.depth=2, shrinkage=0.01)

  glm_model = glm(response ~ ., data=train_data,
                  family=binomial)

  # Predictions
  tree_prob = predict(tree_model, test_data, type="vector")[,2]
  rf_prob = predict(rf_model, test_data, type="prob")[,2]

  test_data_gbm = test_data
  gbm_prob = predict(gbm_model, test_data_gbm, n.trees=1000,
                     type="response")
  glm_prob = predict(glm_model, test_data, type="response")

  # Class predictions (threshold 0.5)
  tree_pred = factor(ifelse(tree_prob > 0.5, 1, 0))
  rf_pred = factor(ifelse(rf_prob > 0.5, 1, 0))
  gbm_pred = factor(ifelse(gbm_prob > 0.5, 1, 0))
  glm_pred = factor(ifelse(glm_prob > 0.5, 1, 0))

  # Accuracy
  acc_tree[k] = mean(tree_pred == test_data$response)
  acc_rf[k] = mean(rf_pred == test_data$response)
  acc_gbm[k] = mean(gbm_pred == test_data$response)
  acc_glm[k] = mean(glm_pred == test_data$response)

  # AUC
  auc_tree[k] = auc(roc(test_data$response, tree_prob))
  auc_rf[k] = auc(roc(test_data$response, rf_prob))
  auc_gbm[k] = auc(roc(test_data$response, gbm_prob))
  auc_glm[k] = auc(roc(test_data$response, glm_prob))
}

# Compare model performance
par(mfrow=c(1,2))
boxplot(acc_tree, acc_rf, acc_gbm, acc_glm,
        names=c("Tree", "RF", "GBM", "GLM"),
        main="Accuracy Comparison")

boxplot(auc_tree, auc_rf, auc_gbm, auc_glm,
        names=c("Tree", "RF", "GBM", "GLM"),
        main="AUC Comparison")
```

Common Performance Patterns:

- Single trees usually have the lowest performance but highest interpretability
- Random Forests typically outperform single trees due to variance reduction
- Gradient Boosting often achieves the highest accuracy with proper tuning
- Tree-based methods generally perform well without extensive feature engineering
- For very high-dimensional data, regularized methods may outperform tree-based methods

Choosing Between Tree-Based Methods:

- **Single Tree:** When interpretability is paramount
- **Random Forest:** For robust performance with minimal tuning
- **Gradient Boosting:** When maximizing predictive accuracy is the goal
- **Bagging:** When you want to reduce variance but keep all features

5. Support Vector Machines (SVMs)

SVM Basics

Support Vector Machines are powerful supervised learning algorithms that find the optimal hyperplane to separate classes by maximizing the margin between the closest points (support vectors) from each class.

Key Concepts:

- **Support Vectors:** Data points closest to the decision boundary that define its position
- **Margin:** Distance between decision boundary and closest support vectors
- **Kernel Trick:** Method to implicitly map data to higher dimensions where classes become separable
- **C Parameter:** Controls trade-off between margin width and misclassification penalty

Basic SVM Implementation

```r
# Load packages
library(e1071)  # Main package for SVM implementation

# Prepare data
set.seed(123)
train_idx <- sample(nrow(iris), 0.7*nrow(iris))
train_data <- iris[train_idx, ]
test_data <- iris[-train_idx, ]

# Fit SVM model with radial kernel
svm_model <- svm(Species ~ .,
                 data = train_data,
                 kernel = "radial",
                 cost = 1,
                 scale = TRUE)

# Model summary
summary(svm_model)

# Make predictions
predictions <- predict(svm_model, test_data)

# Evaluate model
confusion_matrix <- table(
  Predicted = predictions,
  Actual = test_data$Species
)
print(confusion_matrix)
accuracy <- sum(diag(confusion_matrix))/sum(confusion_matrix)
print(paste("Accuracy:", round(accuracy, 4)))
```

Handling Categorical Variables

SVMs work with numeric data, so categorical variables need to be converted to numeric form.

Categorical Variables in SVMs

```r
# Prepare dataset with categorical variables
data(mtcars)
head(mtcars)

# Create categorical variables
mtcars$vs_factor <- as.factor(mtcars$vs)
mtcars$am_factor <- as.factor(mtcars$am)

# Convert to numeric using one-hot encoding
formula <- ~ vs_factor + am_factor
model_matrix <- model.matrix(formula, mtcars)[,-1]  # Drop intercept
head(model_matrix)

# Combine with numeric predictors
X <- cbind(mtcars[, c("mpg", "disp", "hp")], model_matrix)
y <- mtcars$cyl

# Now use in SVM
svm_model <- svm(y ~ ., data = data.frame(X, y = as.factor(y)))
summary(svm_model)
```

Kernel Types

SVMs can use different kernel functions to implicitly map data to higher dimensions where they may be more easily separable.

Common Kernel Types:

- **Linear:** No mapping, suitable for linearly separable data
- **Polynomial:** Maps to polynomial space with degree d
- **Radial (RBF):** Maps to infinite-dimensional space, best for non-linear boundaries
- **Sigmoid:** Similar to neural networks

Comparing Different Kernels

```r
# Generate 2D nonlinear data
library(e1071)
set.seed(123)
x <- matrix(rnorm(200*2), ncol = 2)
x[1:100,] <- x[1:100,] + 2
x[101:150,] <- x[101:150,] - 2
y <- c(rep(1,150), rep(2,50))
data <- data.frame(x = x, y = as.factor(y))

# Split into train/test
train_idx <- sample(nrow(data), 0.7*nrow(data))
train_data <- data[train_idx, ]
test_data <- data[-train_idx, ]

# Compare different kernels
kernels <- c("linear", "polynomial", "radial", "sigmoid")
results <- data.frame(
  Kernel = kernels,
  Accuracy = numeric(length(kernels))
)

for (i in 1:length(kernels)) {
  svm_model <- svm(y ~ .,
                   data = train_data,
                   kernel = kernels[i],
                   cost = 1)

  predictions <- predict(svm_model, test_data)
  confusion <- table(Predicted = predictions, Actual = test_data$y)
  accuracy <- sum(diag(confusion))/sum(confusion)
  results$Accuracy[i] <- accuracy
}

print(results)
```

Kernel Selection Guidelines:

- **Linear:** Try first for high-dimensional data (many features)
- **Radial:** Good default for low-dimensional data with complex boundaries
- **Polynomial:** When relationship may follow polynomial pattern
- Consider dimensionality, sample size, and computational resources

Parameter Tuning

Tuning SVM parameters is crucial for optimal performance:

- **Cost (C):** Higher values penalize misclassification more strongly
- **Gamma:** For radial and polynomial kernels; controls influence range of each point
- **Degree:** For polynomial kernel; higher values create more complex boundaries

Parameter Tuning for SVMs

```r
# Grid search for best parameters using tune()
library(e1071)
data(iris)

# Define parameter ranges to search
tuned <- tune.svm(Species ~ .,
                  data = iris,
                  kernel = "radial",
                  cost = 10^(-1:2),    # 0.1, 1, 10, 100
                  gamma = 10^(-2:1))   # 0.01, 0.1, 1, 10

# Best model parameters
print(tuned$best.parameters)

# Performance at best parameters
print(tuned$best.performance)

# Full performance table
head(tuned$performances)

# Visualize parameter tuning results
plot(tuned)

# Use the best model
best_model <- tuned$best.model
summary(best_model)
```

Multi-class SVM

SVMs are binary classifiers, but can be extended to multi-class problems using one of two strategies:

- **One-vs-One (OVO):** Train binary classifier for each pair of classes
- **One-vs-Rest (OVR):** Train binary classifier for each class against all others

Multi-class SVM

```r
# Multi-class SVM with One-vs-One strategy (default in e1071)
library(e1071)
data(iris)

# Split data
set.seed(123)
train_idx <- sample(nrow(iris), 0.7*nrow(iris))
train_data <- iris[train_idx, ]
test_data <- iris[-train_idx, ]

# Fit multi-class SVM
svm_model <- svm(Species ~ .,
                 data = train_data,
                 kernel = "radial",
                 cost = 1,
                 probability = TRUE)  # For class probabilities

# Prediction type
# - type="response" gives class labels (default)
# - type="probabilities" gives class probabilities
predictions <- predict(svm_model, test_data, type="response")
prob_predictions <- predict(svm_model, test_data, probability=TRUE)

# Extract probabilities
prob_attr <- attr(prob_predictions, "probabilities")
head(prob_attr)

# Evaluate model
confusion <- table(Predicted = predictions, Actual = test_data$Species)
print(confusion)
accuracy <- sum(diag(confusion))/sum(confusion)
print(paste("Accuracy:", round(accuracy, 4)))
```

SVM Regression

SVMs can be used for regression (SVR) by finding the hyperplane that maximizes the number of points within a specified epsilon distance.

SVM Regression

```r
# SVM for regression (SVR)
library(e1071)
data(mtcars)

# Scale predictors
mtcars_scaled <- mtcars
mtcars_scaled[, -1] <- scale(mtcars_scaled[, -1])

# Split data
set.seed(123)
train_idx <- sample(nrow(mtcars_scaled), 0.7*nrow(mtcars_scaled))
train_data <- mtcars_scaled[train_idx, ]
test_data <- mtcars_scaled[-train_idx, ]

# Fit SVM regression model
svr_model <- svm(mpg ~ .,
                 data = train_data,
                 kernel = "radial",
                 cost = 10,
                 epsilon = 0.1,  # Width of epsilon-insensitive tube
                 type = "eps-regression")

# Make predictions
predictions <- predict(svr_model, test_data)

# Evaluate model
mse <- mean((test_data$mpg - predictions)^2)
rmse <- sqrt(mse)
print(paste("RMSE:", round(rmse, 4)))

# Visualize actual vs predicted
plot(test_data$mpg, predictions,
     main="Actual vs Predicted MPG",
     xlab="Actual", ylab="Predicted")
abline(0, 1, col="red")  # Perfect prediction line
```

Evaluating and Comparing SVM Models

When evaluating SVM models, compare different kernels and parameter settings with standard evaluation metrics.

SVM Evaluation and Comparison

```r
# Comparing SVM models with different kernels and parameters
library(e1071)
library(pROC)
library(caret)

# Prepare data
data(iris)
iris_binary <- iris[iris$Species != "virginica", ]  # Make binary for ROC
iris_binary$Species <- factor(iris_binary$Species)

# Split data
set.seed(123)
train_idx <- sample(nrow(iris_binary), 0.7*nrow(iris_binary))
train_data <- iris_binary[train_idx, ]
test_data <- iris_binary[-train_idx, ]

# Define models to compare
models <- list(
  linear_svm = svm(Species ~ ., data = train_data, kernel = "linear", cost = 1, probability = TRUE),
  radial_svm = svm(Species ~ ., data = train_data, kernel = "radial", cost = 1, probability = TRUE),
  poly_svm = svm(Species ~ ., data = train_data, kernel = "polynomial", degree = 2, cost = 1,
    probability = TRUE)
)

# Evaluate each model
results <- data.frame(
  Model = names(models),
  Accuracy = numeric(length(models)),
  AUC = numeric(length(models))
)

roc_curves <- list()

for (i in 1:length(models)) {
  # Predictions
  model_name <- names(models)[i]
  model <- models[[i]]

  # Class predictions
  pred_class <- predict(model, test_data)
  confusion <- confusionMatrix(pred_class, test_data$Species)

  # Probability predictions
  pred_prob <- predict(model, test_data, probability = TRUE)
  prob <- attr(pred_prob, "probabilities")[, 2]  # Probability of second class

  # ROC curve
  roc_obj <- roc(test_data$Species, prob)
  roc_curves[[model_name]] <- roc_obj

  # Store results
  results$Accuracy[i] <- confusion$overall["Accuracy"]
  results$AUC[i] <- auc(roc_obj)
}

# Display results
print(results)

# Plot ROC curves for comparison
plot(roc_curves[[1]], main="ROC Curve Comparison")
for (i in 2:length(roc_curves)) {
  plot(roc_curves[[i]], add=TRUE, col=i)
}
legend("bottomright", names(models), col=1:length(models), lwd=2)
```

SVM Advantages and Disadvantages

Advantages:

- Effective in high-dimensional spaces
- Works well with clear margins of separation
- Memory efficient (only support vectors matter)
- Kernel trick allows for handling non-linear problems
- Robust against overfitting in high-dimensional spaces

Disadvantages:

- Computationally intensive for large datasets
- Requires careful parameter tuning
- No direct probability estimates (requires additional calculation)
- Not straightforward to interpret

- Sensitive to scaling and preprocessing

When to Use SVMs:

- When you have a clear margin of separation between classes
- For high-dimensional data with many features
- When interpretability is not a primary concern
- For processing structured data (tabular format)
- When traditional models show poor performance

6. Neural Networks

Note: Code examples in this section are adapted from materials by Eric Wolsztynski.

Neural Network Basics

Neural networks are computational models inspired by the human brain that consist of layers of interconnected nodes (neurons). Each node receives input, applies a transformation, and passes the output to the next layer.

Key Components:

- **Input Layer:** Receives the features/variables
- **Hidden Layers:** Intermediate layers that transform the data
- **Output Layer:** Produces the prediction or classification
- **Weights:** Connection strengths between neurons
- **Activation Functions:** Non-linear transformations applied to inputs (e.g., sigmoid, ReLU, tanh)

Basic Neural Network Implementation

```r
# Implementation using neuralnet package
library(neuralnet)

# Prepare data
set.seed(123)
n <- nrow(iris)
data <- iris[sample(1:n), ]  # Shuffle dataset

# Train neural network with 2 hidden layers (6 and 5 neurons)
nn_model <- neuralnet(Species ~ .,
                      data=data,
                      hidden=c(6, 5))  # Two hidden layers

# Visualize the network
plot(nn_model,
     information=FALSE,
     col.entry='red',
     col.out='green',
     show.weights=FALSE)

# Prediction
predictions <- predict(nn_model, newdata=test_data)
```

Data Preprocessing for Neural Networks

Neural networks often perform better with scaled input data to prevent issues with gradient calculations and to ensure all features contribute equally.

Common Preprocessing Steps:

- **Scaling numeric features** to [0,1] or standardizing to mean=0, sd=1
- **One-hot encoding** categorical variables
- **Normalizing the response variable** for regression problems

Data Preprocessing for Neural Networks

```r
# Function for min-max scaling (normalize to [0,1])
minmax_scale <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}

# Prepare dataset
data(Boston)
set.seed(123)
n <- nrow(Boston)
train_idx <- sample(1:n, round(0.7*n))

# Create scaled version for training
boston_scaled <- Boston
# Scale predictors
boston_scaled[, -14] <- apply(boston_scaled[, -14], 2, minmax_scale)
# Scale response (alternative approach - divide by max value)
boston_scaled$medv <- boston_scaled$medv / 50  # Max value is around 50

# Handle categorical variables (if present)
# Convert factors to numeric (0, 1, ...)
recode_factor <- function(x) {
  if(is.factor(x)) {
    return(as.numeric(x) - 1)  # Subtract 1 to start from 0
  } else {
    return(x)
  }
}

# Apply recoding to each column if needed
# boston_scaled <- data.frame(lapply(boston_scaled, recode_factor))

# Split into train/test
train_data <- boston_scaled[train_idx, ]
test_data <- boston_scaled[-train_idx, ]
```

Neural Network Packages in R

R offers several packages for implementing neural networks, each with different approaches and capabilities:

- **neuralnet:** Flexible implementation with multiple hidden layers and various activation functions
- **nnet:** Single hidden layer networks, efficient for smaller networks
- **keras/tensorflow:** Advanced deep learning capabilities
- **h2o:** Distributed machine learning platform with neural network capabilities

Different Neural Network Packages

```r
# Using nnet for single-layer network
library(nnet)
set.seed(123)

# Train model
nnet_model <- nnet(medv ~ .,
                   data=train_data,
                   size=5,        # 5 nodes in hidden layer
                   decay=0.01,    # Weight decay parameter
                   linout=TRUE)  # Linear output for regression

# Predictions with nnet
nnet_preds <- predict(nnet_model, test_data) * 50  # Scale back

# Using neuralnet for multi-layer network
library(neuralnet)
set.seed(123)

# Prepare formula
predictors <- names(train_data)[-which(names(train_data) == "medv")]
formula <- as.formula(paste("medv ~", paste(predictors, collapse=" + ")))

# Train model
neuralnet_model <- neuralnet(formula,
                             data=train_data,
                             hidden=c(5, 3),  # Two hidden layers
                             linear.output=TRUE)  # For regression

# Predictions with neuralnet
nn_preds <- predict(neuralnet_model, test_data) * 50  # Scale back

# Compare RMSE
nnet_rmse <- sqrt(mean((nnet_preds - Boston$medv[-train_idx])^2))
nn_rmse <- sqrt(mean((nn_preds - Boston$medv[-train_idx])^2))
print(c(nnet_rmse, nn_rmse))
```

Tuning Neural Network Parameters

Neural networks have various parameters that significantly impact performance:

Key Hyperparameters:

- **Network Architecture:** Number of hidden layers and neurons per layer
- **Activation Function:** sigmoid, tanh, ReLU, linear
- **Learning Rate:** Controls step size during optimization
- **Weight Decay (Regularization):** Prevents overfitting
- **Number of Iterations:** Maximum epochs for training

Neural Network Parameter Tuning

```r
# Exploring the effect of hidden layer size
library(nnet)
set.seed(4061)

# Data preparation (iris dataset)
data(iris)
n <- nrow(iris)
iris_shuffled <- iris[sample(1:n), ]
# Scale predictors to [0,1]
iris_scaled <- iris_shuffled
iris_scaled[, 1:4] <- scale(iris_scaled[, 1:4],
                            center=apply(iris_scaled[, 1:4], 2, min),
                            scale=apply(iris_scaled[, 1:4], 2, max) -
                                  apply(iris_scaled[, 1:4], 2, min))

# Split data
train_idx <- sample(1:n, round(0.7*n))
train_data <- iris_scaled[train_idx, ]
test_data <- iris_scaled[-train_idx, ]

# Test different network sizes
sizes <- c(1, 3, 5, 10, 15)
train_accuracy <- numeric(length(sizes))
test_accuracy <- numeric(length(sizes))

for(i in 1:length(sizes)) {
  # Train model with current size
  nn_model <- nnet(Species ~ .,
                   data=train_data,
                   size=sizes[i],
                   decay=0.01,
                   maxit=500)

  # Training accuracy
  train_pred <- predict(nn_model, train_data, type="class")
  train_accuracy[i] <- mean(train_pred == train_data$Species)

  # Test accuracy
  test_pred <- predict(nn_model, test_data, type="class")
  test_accuracy[i] <- mean(test_pred == test_data$Species)
}

# Test different decay values (regularization)
decays <- c(0, 0.001, 0.01, 0.1, 0.5)
train_decay_accuracy <- numeric(length(decays))
test_decay_accuracy <- numeric(length(decays))

for(i in 1:length(decays)) {
  # Train model with optimal size and current decay
  nn_model <- nnet(Species ~ .,
                   data=train_data,
                   size=5,  # Using a fixed size
                   decay=decays[i],
                   maxit=500)

  # Training accuracy
  train_pred <- predict(nn_model, train_data, type="class")
  train_decay_accuracy[i] <- mean(train_pred == train_data$Species)

  # Test accuracy
  test_pred <- predict(nn_model, test_data, type="class")
  test_decay_accuracy[i] <- mean(test_pred == test_data$Species)
}

# Plot results
par(mfrow=c(1,2))
plot(sizes, train_accuracy, type="b", col="blue", ylim=c(min(train_accuracy, test_
    accuracy), 1),
     xlab="Hidden Layer Size", ylab="Accuracy", main="Effect of Network Size")
lines(sizes, test_accuracy, type="b", col="red")
legend("bottomright", legend=c("Train", "Test"), col=c("blue", "red"), lty=1)

plot(decays, train_decay_accuracy, type="b", col="blue",
     ylim=c(min(train_decay_accuracy, test_decay_accuracy), 1),
     xlab="Weight Decay", ylab="Accuracy", main="Effect of Regularization")
lines(decays, test_decay_accuracy, type="b", col="red")
legend("bottomright", legend=c("Train", "Test"), col=c("blue", "red"), lty=1)
```

Neural Networks for Classification

Neural networks can handle both binary and multi-class classification problems efficiently:

Neural Networks for Classification

```r
# Classification with neural networks
library(nnet)
library(neuralnet)
set.seed(123)

# Prepare data (iris dataset)
data(iris)
iris_scaled <- iris
iris_scaled[, 1:4] <- scale(iris_scaled[, 1:4])

# Split data
n <- nrow(iris_scaled)
train_idx <- sample(1:n, round(0.7*n))
train_data <- iris_scaled[train_idx, ]
test_data <- iris_scaled[-train_idx, ]

# Single-layer network with nnet
nnet_model <- nnet(Species ~ .,
                   data=train_data,
                   size=5,
                   decay=0.01)

# Get class predictions
nnet_class_pred <- predict(nnet_model, test_data, type="class")

# Get probability predictions
nnet_prob_pred <- predict(nnet_model, test_data, type="raw")

# Evaluate model
nnet_confusion <- table(Predicted=nnet_class_pred, Actual=test_data$Species)
nnet_accuracy <- sum(diag(nnet_confusion)) / sum(nnet_confusion)

print(nnet_confusion)
print(paste("Accuracy:", round(nnet_accuracy, 4)))

# Use neuralnet for comparison
# Convert Species to dummy variables for neuralnet
species_dummies <- model.matrix(~ Species - 1, train_data)
train_data_nn <- cbind(train_data[, 1:4], species_dummies)

# Create formula for neuralnet
input_vars <- paste(names(train_data)[1:4], collapse=" + ")
output_vars <- paste(colnames(species_dummies), collapse=" + ")
nn_formula <- as.formula(paste(output_vars, "~", input_vars))

# Train neuralnet model
nn_model <- neuralnet(nn_formula,
                      data=train_data_nn,
                      hidden=c(5),
                      linear.output=FALSE)

# Make predictions
test_pred_input <- as.matrix(test_data[, 1:4])
nn_prediction <- compute(nn_model, test_pred_input)

# Convert to class predictions
predicted_classes <- max.col(nn_prediction$net.result)
actual_classes <- as.numeric(test_data$Species)

# Calculate accuracy
nn_accuracy <- mean(predicted_classes == actual_classes)
print(paste("neuralnet Accuracy:", round(nn_accuracy, 4)))
```

Neural Networks for Regression

Neural networks are also powerful for regression tasks, but often require normalization of the response variable:

Neural Networks for Regression

```r
# Regression with neural networks
library(nnet)
library(MASS)  # For Boston housing dataset
set.seed(123)

# Load and prepare data
data(Boston)
boston_scaled <- Boston
# Scale all predictors
boston_scaled[, -14] <- scale(boston_scaled[, -14])
# Scale the target variable
max_medv <- max(boston_scaled$medv)
boston_scaled$medv <- boston_scaled$medv / max_medv

# Split data
n <- nrow(boston_scaled)
train_idx <- sample(1:n, round(0.7*n))
train_data <- boston_scaled[train_idx, ]
test_data <- boston_scaled[-train_idx, ]

# Train neural network for regression
# linout=TRUE is critical for regression tasks
nn_reg_model <- nnet(medv ~ .,
                     data=train_data,
                     size=10,
                     decay=0.01,
                     linout=TRUE,
                     maxit=1000)

# Make predictions and rescale back
nn_pred <- predict(nn_reg_model, test_data) * max_medv
actual_values <- Boston$medv[-train_idx]

# Evaluate performance
rmse <- sqrt(mean((nn_pred - actual_values)^2))
r_squared <- cor(nn_pred, actual_values)^2

print(paste("RMSE:", round(rmse, 4)))
print(paste("R-squared:", round(r_squared, 4)))

# Compare with linear regression
lm_model <- lm(medv ~ ., data=train_data)
lm_pred <- predict(lm_model, test_data) * max_medv
lm_rmse <- sqrt(mean((lm_pred - actual_values)^2))
lm_r_squared <- cor(lm_pred, actual_values)^2

print(paste("Linear Regression RMSE:", round(lm_rmse, 4)))
print(paste("Linear Regression R-squared:", round(lm_r_squared, 4)))

# Visualization
par(mfrow=c(1,2))
plot(actual_values, nn_pred, main="Neural Network",
     xlab="Actual", ylab="Predicted", pch=19)
abline(0, 1, col="red", lwd=2)

plot(actual_values, lm_pred, main="Linear Regression",
     xlab="Actual", ylab="Predicted", pch=19)
abline(0, 1, col="red", lwd=2)
```

Variable Importance in Neural Networks

Unlike tree-based methods, neural networks don't provide variable importance measures directly. The Olden method calculates importance based on the product of connection weights between layers:

Variable Importance with Olden Index

```r
# Variable importance in neural networks using Olden index
library(nnet)
library(NeuralNetTools)
library(randomForest)
set.seed(123)

# Prepare data (Boston Housing)
data(Boston)
boston_scaled <- Boston
# Scale predictors to [0,1]
minmax_scale <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
}
boston_scaled[, -14] <- apply(boston_scaled[, -14], 2, minmax_scale)
boston_scaled$medv <- Boston$medv / 50  # Scale response

# Train neural network
nn_model <- nnet(medv ~ .,
                 data=boston_scaled,
                 size=10,
                 decay=0.1,
                 linout=TRUE)

# Calculate Olden's variable importance
olden_importance <- olden(nn_model, bar_plot=FALSE)

# Train random forest for comparison
rf_model <- randomForest(medv ~ ., data=Boston)
rf_importance <- importance(rf_model)

# Normalize importance values for comparison
olden_abs <- abs(olden_importance$importance)
olden_normalized <- olden_abs / sum(olden_abs)
rf_normalized <- rf_importance / sum(rf_importance)

# Compare importance values
importance_comparison <- data.frame(
  Variable = rownames(olden_importance),
  NN_Importance = olden_normalized,
  RF_Importance = rf_normalized
)

# Plot comparison
par(mfrow=c(1,2))
barplot(olden_normalized,
        names.arg=rownames(olden_importance),
        las=2, main="Neural Network Importance",
        cex.names=0.7)

barplot(rf_normalized,
        names.arg=rownames(rf_importance),
        las=2, main="Random Forest Importance",
        cex.names=0.7)

# Alternative: plot combined importance
par(mfrow=c(1,1))
importance_matrix <- rbind(
  NN = olden_normalized,
  RF = rf_normalized
)
barplot(importance_matrix, beside=TRUE,
        names.arg=rownames(olden_importance),
        las=2, legend.text=TRUE,
        col=c("lightblue", "salmon"),
        main="Variable Importance Comparison",
        cex.names=0.7)
```

Advantages and Disadvantages

Advantages:

- Can model complex non-linear relationships

- Flexibility to handle various types of data and problems
- Automatic feature interaction detection
- Often performs well with large datasets
- Single model can handle multiple outputs (multi-task learning)

Disadvantages:

- Requires careful hyperparameter tuning
- Computationally intensive to train
- "Black box" nature limits interpretability
- Sensitive to scaling and preprocessing
- May overfit with small datasets
- Performance can vary based on random initialization

When to Use Neural Networks:

- Complex pattern recognition tasks
- Large datasets with many features
- When prediction accuracy is more important than interpretability
- For processing unstructured data (images, text, audio)
- When traditional models show poor performance

7. Feature Selection

Note: Code examples in this section are adapted from materials by Eric Wolsztynski.

Feature Selection Basics

Feature selection is the process of selecting a subset of relevant features for model building. It reduces overfitting, improves accuracy, and creates more interpretable models.

Motivation for Feature Selection:

- **Curse of Dimensionality:** Models with too many features often perform poorly
- **Interpretability:** Fewer features lead to more understandable models
- **Computational Efficiency:** Training and prediction are faster with fewer features
- **Avoiding Multicollinearity:** Removing redundant features improves stability

Impact of Redundant Features

```r
# Demonstrate effect of adding noise variables to Random Forest
library(ISLR)
library(randomForest)

set.seed(4061)
dat <- na.omit(Hitters)
x <- dat
x$Salary <- NULL
y <- log(dat$Salary)
n <- nrow(x)
p <- ncol(x)

# Baseline model
rfo1 <- randomForest(x=x, y=y)
oob_error_base <- mean(rfo1$mse)

# Add random noise variables and track performance
K <- 10
res <- numeric(K)
for(k in 1:K) {
  # Add k*p noise variables
  x2 <- cbind(x, matrix(rnorm(n*k*p), nrow=n))
  rfo2 <- randomForest(x=x2, y=y)
  res[k] <- mean(rfo2$mse)
}

# Plot change in error with increasing noise variables
resf <- c(oob_error_base, res)
plot(c(0:K), resf, type='b', pch=20,
     main="OOB MSE with Increasing Noise Variables",
     xlab="Number of Noise Variable Groups", ylab="OOB MSE")

# Plot percentage change
resp <- (resf-resf[1])/resf[1]*100
plot(c(0:K), resp, type='b', pch=20,
     main="% Change in OOB MSE",
     xlab="Number of Noise Variable Groups", ylab="% Change")
```

Subset Selection Methods

Subset selection involves evaluating different combinations of features to find the optimal subset.

Key Methods:

- **Best Subset Selection:** Evaluates all possible combinations (exhaustive)
- **Forward Selection:** Starts with no variables and adds the most significant ones
- **Backward Selection:** Starts with all variables and removes the least significant ones
- **Stepwise Selection:** Combination of forward and backward approaches

Best Subset Selection

```r
library(ISLR)
library(leaps)  # For regsubsets function

# Load and prepare data
data <- na.omit(Hitters)

# Best subset selection (exhaustive search)
set.seed(4061)
reg.full <- regsubsets(Salary ~ ., data=data,
                       method="exhaustive",
                       nvmax=19)  # Up to 19 variables

# Examine results
summary_res <- summary(reg.full)

# Plot model fit metrics
par(mfrow=c(1,2))
# R-squared
plot(summary_res$rsq, type="b", pch=20,
     xlab="Number of Variables", ylab="R-squared")

# Adjusted R-squared
plot(summary_res$adjr2, type="b", pch=20,
     xlab="Number of Variables", ylab="Adjusted R-squared")

# Find optimal model by adjusted R-squared
best_model_idx <- which.max(summary_res$adjr2)
best_model_vars <- summary_res$which[best_model_idx, ]

# Print selected variables
print(names(which(best_model_vars)[-1]))  # Exclude intercept

# Visualize variable selection across models
plot(reg.full, scale="adjr2")
```

Forward and Backward Selection

```r
# Forward selection
set.seed(4061)
reg.fwd <- regsubsets(Salary ~ ., data=data,
                      method="forward",
                      nvmax=19)

# Backward selection
set.seed(4061)
reg.bwd <- regsubsets(Salary ~ ., data=data,
                      method="backward",
                      nvmax=19)

# Compare performance
par(mfrow=c(1,2))
# Compare RSS
plot(summary(reg.bwd)$rss, type='b', pch=20,
     xlab="Number of Variables", ylab="RSS",
     main="RSS Comparison")
points(summary(reg.fwd)$rss, type='b', pch=15, col=2)
legend("topright", c("Backward", "Forward"),
       pch=c(20,15), col=c(1,2))

# Compare Adjusted R-squared
plot(summary(reg.bwd)$adjr2, type='b', pch=20,
     xlab="Number of Variables", ylab="Adjusted R-squared",
     main="Adj  R   Comparison")
points(summary(reg.fwd)$adjr2, type='b', pch=15, col=2)
legend("bottomright", c("Backward", "Forward"),
       pch=c(20,15), col=c(1,2))

# Optimal models
best_bwd_idx <- which.max(summary(reg.bwd)$adjr2)
best_fwd_idx <- which.max(summary(reg.fwd)$adjr2)

# Compare coefficients in 4-variable models
coef(reg.fwd, id=4)
coef(reg.bwd, id=4)
```

Stepwise Selection with stats::step

```r
# Stepwise selection using stats::step
# Start with full model
lm.full <- lm(Salary ~ ., data=data)

# Bidirectional stepwise selection (both)
set.seed(4061)
step.both <- step(lm.full, direction="both")

# Forward stepwise selection
set.seed(4061)
step.fwd <- step(lm.full, direction="forward")

# Backward stepwise selection
set.seed(4061)
step.bwd <- step(lm.full, direction="backward")

# Examine final model
summary(step.bwd)

# Get coefficients from optimal model
coef(step.bwd)

# Compare variable importance by coefficient magnitude
coefs <- abs(coef(step.bwd)[-1])  # Remove intercept
coefs_percent <- 100 * coefs / sum(coefs)
sort(coefs_percent, decreasing=TRUE)

# Make predictions
# Split data into train and test
set.seed(4061)
n <- nrow(data)
train_idx <- sample(1:n, 0.7*n)
train_data <- data[train_idx, ]
test_data <- data[-train_idx, ]

# Train model
lm.full <- lm(Salary ~ ., data=train_data)
step.model <- step(lm.full, direction="both")

# Predict and evaluate
predictions <- predict(step.model, newdata=test_data)
rmse <- sqrt(mean((predictions - test_data$Salary)^2))
print(paste("Test RMSE:", round(rmse, 2)))
```

Regularization-Based Selection

Regularization methods like LASSO (L1) automatically perform feature selection by shrinking coefficients of less important features to zero.

LASSO for Feature Selection

```r
library(glmnet)

# Load and prepare data
data <- na.omit(Hitters)
x <- model.matrix(Salary ~ ., data)[, -1]  # Remove intercept
y <- data$Salary

# Split data
set.seed(4061)
n <- nrow(data)
train_idx <- sample(1:n, 0.7*n)
x_train <- x[train_idx, ]
y_train <- y[train_idx]
x_test <- x[-train_idx, ]
y_test <- y[-train_idx]

# Fit LASSO with cross-validation for optimal lambda
set.seed(4061)
lasso.cv <- cv.glmnet(x_train, y_train, alpha=1)
plot(lasso.cv)

# Get optimal lambda values
lambda_min <- lasso.cv$lambda.min  # Minimum MSE
lambda_1se <- lasso.cv$lambda.1se  # 1 Std. Error rule (more regularization)
print(c(lambda_min, lambda_1se))

# Fit LASSO with optimal lambda
lasso_min <- glmnet(x_train, y_train, alpha=1, lambda=lambda_min)
lasso_1se <- glmnet(x_train, y_train, alpha=1, lambda=lambda_1se)

# Selected variables
coef_min <- coef(lasso_min)
coef_1se <- coef(lasso_1se)

# Count non-zero coefficients
sum(coef_min[-1] != 0)  # Exclude intercept
sum(coef_1se[-1] != 0)  # Exclude intercept

# Print selected variables
selected_min <- which(coef_min[-1] != 0)
selected_1se <- which(coef_1se[-1] != 0)
print(colnames(x)[selected_min])
print(colnames(x)[selected_1se])

# Evaluate models
pred_min <- predict(lasso_min, x_test)
pred_1se <- predict(lasso_1se, x_test)
rmse_min <- sqrt(mean((pred_min - y_test)^2))
rmse_1se <- sqrt(mean((pred_1se - y_test)^2))
print(paste("RMSE with lambda_min:", round(rmse_min, 2)))
print(paste("RMSE with lambda_1se:", round(rmse_1se, 2)))
```

Recursive Feature Elimination

Recursive Feature Elimination (RFE) recursively removes the least important features based on model performance.

Recursive Feature Elimination

```r
library(caret)

# Load and prepare data
data <- na.omit(Hitters)
x <- data[, -which(names(data) == "Salary")]
y <- data$Salary

# Split data
set.seed(4061)
train_idx <- createDataPartition(y, p=0.7, list=FALSE)
x_train <- x[train_idx, ]
y_train <- y[train_idx]
x_test <- x[-train_idx, ]
y_test <- y[-train_idx]

# Configure RFE
set.seed(4061)
# Define which feature subsets to evaluate
subsets <- c(1:5, 10, 15, ncol(x))

# RFE with linear regression
ctrl_lm <- rfeControl(functions=lmFuncs,
                      method="cv",
                      number=10,
                      verbose=FALSE)

lm_rfe <- rfe(x_train, y_train,
              sizes=subsets,
              rfeControl=ctrl_lm)

# RFE with random forest
ctrl_rf <- rfeControl(functions=rfFuncs,
                      method="cv",
                      number=5,
                      verbose=FALSE)

rf_rfe <- rfe(x_train, y_train,
              sizes=subsets,
              rfeControl=ctrl_rf)

# Print results
print(lm_rfe)
print(rf_rfe)

# Plot results
par(mfrow=c(1,2))
plot(lm_rfe, main="Linear Model RFE")
plot(rf_rfe, main="Random Forest RFE")

# Get selected variables
lm_vars <- predictors(lm_rfe)
rf_vars <- predictors(rf_rfe)
print(lm_vars)
print(rf_vars)

# Compare variable sets
common_vars <- intersect(lm_vars, rf_vars)
print(paste("Common variables:",
            paste(common_vars, collapse=", ")))

# Make predictions using selected features
lm_pred <- predict(lm_rfe, x_test)
rf_pred <- predict(rf_rfe, x_test)
lm_rmse <- sqrt(mean((lm_pred - y_test)^2))
rf_rmse <- sqrt(mean((rf_pred - y_test)^2))
print(paste("Linear Model RFE RMSE:", round(lm_rmse, 2)))
print(paste("Random Forest RFE RMSE:", round(rf_rmse, 2)))
```

Avoiding Selection Bias

Selection bias occurs when the feature selection process is influenced by the test data, leading to overly optimistic performance estimates.

Proper Cross-Validation for Feature Selection

```r
# Demonstration of selection bias in feature selection
library(caret)
library(randomForest)

# Example of WRONG approach (selection bias)
biased_cv <- function(data, target) {
  # Filter features using the entire dataset
  my_filter <- function(x, y, cutoff=0.05) {
    p <- ncol(x)
    pvals <- numeric(p)
    for(i in 1:p) {
      pvals[i] <- wilcox.test(as.numeric(x[,i]) ~ y)$p.value
    }
    return(which(pvals < cutoff))
  }

  # Select features using all data
  selected_features <- my_filter(data[, -target], data[, target], 0.2)
  filtered_data <- data[, c(target, selected_features)]

  # Cross-validate on the filtered data
  cv_results <- train(
    x = filtered_data[, -1],
    y = filtered_data[, 1],
    method = "rf",
    trControl = trainControl(method="cv", number=5)
  )

  return(cv_results)
}

# CORRECT approach (avoid selection bias)
unbiased_cv <- function(data, target, B=10) {
  n <- nrow(data)
  acc <- numeric(B)

  for(b in 1:B) {
    # Create bootstrap sample
    boot_idx <- sample(1:n, n, replace=TRUE)
    train_data <- data[boot_idx, ]
    test_idx <- setdiff(1:n, unique(boot_idx))
    test_data <- data[test_idx, ]

    # Filter features using ONLY training data
    my_filter <- function(x, y, cutoff=0.05) {
      p <- ncol(x)
      pvals <- numeric(p)
      for(i in 1:p) {
        pvals[i] <- wilcox.test(as.numeric(x[,i]) ~ y)$p.value
      }
      return(which(pvals < cutoff))
    }

    # Select features using only training data
    selected_features <- my_filter(
      train_data[, -target],
      train_data[, target],
      0.2
    )

    # Only proceed if features were selected
    if(length(selected_features) > 0) {
      # Train model on filtered training data
      rf_model <- randomForest(
        x = train_data[, selected_features],
        y = train_data[, target]
      )

      # Test on held-out data
      predictions <- predict(
        rf_model,
        test_data[, selected_features]
      )

      # Calculate accuracy
      if(is.factor(test_data[, target])) {
        acc[b] <- mean(predictions == test_data[, target])
      } else {
        acc[b] <- sqrt(mean((predictions - test_data[, target])^2))
      }
```

Comparison of Methods

Different feature selection methods have varying strengths, weaknesses, and computational requirements.
Comparison:

- **Best Subset Selection:** Optimal but computationally expensive; only feasible for small p
- **Stepwise Methods:** Computationally efficient but may not find the optimal subset
- **LASSO:** Automatic feature selection with continuous shrinkage; handles high-dimensional data
- **RFE:** Flexible with various ML algorithms; can be computationally intensive
- **Filter Methods:** Fast but ignore feature interactions and the modeling algorithm

Comparing Feature Selection Methods

```r
# Function to compare feature selection methods
compare_feature_selection <- function(data, target_col, k_folds=5) {
  library(leaps)
  library(glmnet)
  library(caret)

  # Extract data
  x <- data[, -which(names(data) == target_col)]
  y <- data[[target_col]]
  n <- nrow(data)
  p <- ncol(x)

  # Prepare matrix format for glmnet
  x_mat <- model.matrix(~ ., data=x)[, -1]

  # Create folds
  set.seed(4061)
  folds <- createFolds(y, k=k_folds, returnTrain=TRUE)

  # Initialize results storage
  rmse_full <- rmse_step <- rmse_lasso <- rmse_rfe <- numeric(k_folds)
  num_vars_step <- num_vars_lasso <- num_vars_rfe <- numeric(k_folds)

  for(i in 1:k_folds) {
    # Split data
    train_idx <- folds[[i]]
    test_idx <- setdiff(1:n, train_idx)

    # Training and test sets
    x_train <- x[train_idx, ]
    y_train <- y[train_idx]
    x_test <- x[test_idx, ]
    y_test <- y[test_idx]

    # Matrix format for train/test
    x_mat_train <- x_mat[train_idx, ]
    x_mat_test <- x_mat[test_idx, ]

    # 1. Full model (baseline)
    lm_full <- lm(y_train ~ ., data=x_train)
    pred_full <- predict(lm_full, x_test)
    rmse_full[i] <- sqrt(mean((pred_full - y_test)^2))

    # 2. Stepwise selection
    lm_step <- step(lm_full, direction="both", trace=0)
    pred_step <- predict(lm_step, x_test)
    rmse_step[i] <- sqrt(mean((pred_step - y_test)^2))
    num_vars_step[i] <- length(coef(lm_step)) - 1  # Subtract intercept

    # 3. LASSO
    cv_lasso <- cv.glmnet(x_mat_train, y_train, alpha=1)
    lambda_1se <- cv_lasso$lambda.1se
    lasso_model <- glmnet(x_mat_train, y_train, alpha=1, lambda=lambda_1se)
    pred_lasso <- predict(lasso_model, x_mat_test, s=lambda_1se)
    rmse_lasso[i] <- sqrt(mean((pred_lasso - y_test)^2))
    num_vars_lasso[i] <- sum(coef(lasso_model)[-1] != 0)

    # 4. RFE with linear model
    ctrl <- rfeControl(functions=lmFuncs, method="cv", number=5)
    subsets <- c(1:min(5, p-1), p/2, p)  # Adjust based on p
    rfe_model <- rfe(x_train, y_train, sizes=subsets, rfeControl=ctrl)
    pred_rfe <- predict(rfe_model, x_test)
    rmse_rfe[i] <- sqrt(mean((pred_rfe - y_test)^2))
    num_vars_rfe[i] <- length(predictors(rfe_model))
  }

  # Aggregate results
  results <- data.frame(
    Method = c("Full Model", "Stepwise", "LASSO", "RFE"),
    Mean_RMSE = c(mean(rmse_full), mean(rmse_step),
                  mean(rmse_lasso), mean(rmse_rfe)),
    SD_RMSE = c(sd(rmse_full), sd(rmse_step),
                sd(rmse_lasso), sd(rmse_rfe)),
    Mean_Features = c(p, mean(num_vars_step),
                      mean(num_vars_lasso), mean(num_vars_rfe))
  )

  return(results)
}
```

Guidelines for Feature Selection

Best Practices:

- Perform feature selection only on the training data to avoid selection bias
- Use cross-validation to evaluate the stability of selected features
- Consider domain knowledge when interpreting selected features
- For small p (¡ 40), consider exhaustive or stepwise methods
- For large p (¿ 40), prefer regularization or embedded methods
- Evaluate multiple feature selection methods for critical applications
- Check for multicollinearity among selected features

Common Pitfalls:

- Selecting features based on the entire dataset (data leakage)
- Not accounting for feature interactions
- Overfitting to the selected features
- Selection bias in cross-validation estimates
- Ignoring domain knowledge in favor of purely statistical selection

8. Practical Applications

Note: Code examples in this section are adapted from materials by Eric Wolsztynski.

Exploratory Data Analysis and Imputation

Proper data analysis begins with understanding the data structure and relationships between variables. Missing data often requires imputation before modeling.

Exploring Data Relationships

```r
# Prepare the dataset
dat.nona = na.omit(airquality)
dat = airquality
dat.nona$Month = as.factor(dat.nona$Month)
dat.nona$Day = as.factor(dat.nona$Day)
dat$Month = as.factor(dat$Month)
dat$Day = as.factor(dat$Day)

# Association analysis
# 1. Correlation matrix
round(cor(na.omit(dat)),3)

# 2. Visual exploratory analysis
pairs(airquality)
boxplot(Wind~Month, data=airquality)
boxplot(Temp~Month, data=airquality)
boxplot(Ozone~Month, data=airquality)

# 3. Single correlation test
cor(airquality$Ozone, airquality$Temp, "pairwise")

# 4. QQ plot to compare distributions
qqplot(scale(airquality$Wind), scale(airquality$Temp))
abline(a=0, b=1)

# Extended EDA with DataExplorer package
library(DataExplorer)

# Overall dataset structure
introduce(dat)
plot_intro(dat)
plot_str(dat)
plot_missing(dat)

# Variable distributions
plot_bar(dat)
plot_bar(dat, with="Ozone")
plot_boxplot(dat, by="Solar.R")
plot_histogram(dat)
plot_qq(dat)
plot_qq(dat, by="Ozone")

# Transform variables for normality if needed
dat2 = dat
dat2$Solar.R = sqrt(dat2$Solar.R+1)
plot_qq(dat2)

# Correlation analysis
plot_correlation(dat[,1:4])
cor(na.omit(dat[,1:4]))
plot_correlation(na.omit(dat[,1:4]))
plot_correlation(na.omit(dat))
```

Handling Missing Data

Missing data can be handled using various imputation strategies based on the relationships identified during EDA.

Imputation Strategies

```r
# 1. Simple imputation with median
is = which(is.na(dat$Solar.R))
dat$Solar.R[is] = median(dat$Solar.R[-is])

# Evaluate imputation with a GLM
glm1.fit = glm(Wind~., data=dat)
cat("RSS after median imputation:", sum((glm1.fit$residuals)^2), "\n")

# 2. Model-based imputation using quadratic regression
is = which(is.na(dat$Ozone))
qmo = lm(Ozone~Temp+I(Temp^2), data=na.omit(airquality))
Ozq = predict(qmo, newdata=airquality[is,])
dat$Ozone[is] = Ozq

# Evaluate imputation with a GLM
glm3.fit = glm(Wind~., data=dat)
cat("RSS after model-based imputation:", sum((glm3.fit$residuals)^2), "\n")

# Visualize the imputation
plot(dat$Ozone, dat$Temp)
points(airquality$Ozone[-is], airquality$Temp[-is], pch=20)
```

Decision Tree Models

Decision trees provide interpretable models for both regression and classification tasks.

Tree-Based Models and Pruning

```r
library(tree)

# 1. Fit an unpruned regression tree
tree.fit = tree(Wind~., data=dat)
summary(tree.fit)

# Identify variables actually used in the tree
summary(tree.fit)$used

# Visualize the tree
plot(tree.fit)
text(tree.fit)

# Calculate RSS for the unpruned tree
rss_unpruned = sum(summary(tree.fit)$residuals^2)
cat("RSS for unpruned tree:", rss_unpruned, "\n")

# 2. Prune the tree to a specific size
pruned.fit = prune.tree(tree.fit, best=10)
summary(pruned.fit)

# Identify variables used in the pruned tree
summary(pruned.fit)$used

# Calculate RSS for the pruned tree
rss_pruned = sum(summary(pruned.fit)$residuals^2)
cat("RSS for pruned tree:", rss_pruned, "\n")

# Compare RSS values
cat("RSS difference (pruned - unpruned):", rss_pruned - rss_unpruned, "\n")
cat("Percentage increase in RSS:",
    100 * (rss_pruned - rss_unpruned) / rss_unpruned, "%\n")

# The RSS typically increases slightly when pruning as we trade a small
# amount of training fit for better generalization
```

Random Forests and Gradient Boosting

Ensemble methods like Random Forests and Gradient Boosting often provide superior predictive performance.

Random Forest Classification

```r
library(ISLR)  # For the Khan dataset
library(randomForest)
library(gbm)

# Set up data
xtrain = Khan$xtrain
xtest = Khan$xtest
ytrain = as.factor(Khan$ytrain)
ytest = as.factor(Khan$ytest)

# Check class distributions
round(prop.table(table(ytrain)), 3)
round(prop.table(table(ytest)), 3)
round(table(ytrain)/length(ytrain), 3) * 100
round(table(ytest)/length(ytest), 3) * 100

# Fit random forest
set.seed(4061)
rfo = randomForest(xtrain, ytrain)
print(rfo)

# Generate predictions
rfpr = predict(rfo, xtest, type="prob")  # Probability predictions
rfp = predict(rfo, xtest)                # Class predictions

# Evaluate accuracy
conf_matrix = table(rfp, ytest)
accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
cat("Random Forest Test Accuracy:", accuracy, "\n")
print(conf_matrix)

# Variable importance
is = which(rfo$importance > 0.4)
important_vars = rownames(rfo$importance)[is]
cat("Important variables (importance > 0.4):\n")
print(important_vars)

# Optional: Visualize variable importance
# barplot(rfo$importance[is,], las=2)
```

Gradient Boosting Model

```r
# Fit gradient boosting model for multinomial classification
gb.out = gbm(ytrain~., data=as.data.frame(xtrain),
             distribution='multinomial')

# Generate predictions
gb.fitted = predict(gb.out, n.trees=gb.out$n.trees)
gb.pred = predict(gb.out, as.data.frame(xtest),
                  n.trees=gb.out$n.trees)

# Convert predictions to class labels
gbp = apply(gb.pred, 1, which.max)

# Evaluate accuracy
gb_conf_matrix = table(ytest, gbp)
gb_accuracy = sum(diag(gb_conf_matrix)) / sum(gb_conf_matrix)
cat("Gradient Boosting Test Accuracy:", gb_accuracy, "\n")
print(gb_conf_matrix)

# Compare model performances
cat("Model comparison:\n")
cat("Random Forest accuracy:", accuracy, "\n")
cat("Gradient Boosting accuracy:", gb_accuracy, "\n")
```

Key Insights from Practical Applications

Data Preparation and Imputation:

- Always examine relationships between variables before imputation

- Use simple methods (median, mean) for weakly correlated variables
- Use model-based approaches when strong relationships exist
- Validate imputation quality through downstream model performance

Model Selection and Evaluation:

- Decision trees offer interpretability but may overfit without pruning
- Pruning reduces model complexity with a small trade-off in training fit
- Random forests often outperform single trees and require less tuning
- For high-dimensional data (like gene expression), feature importance helps identify relevant predictors
- Gradient boosting can provide competitive or superior performance to random forests

Best Practices:

- Always split data before model development to ensure unbiased evaluation
- Use appropriate metrics for model comparison (accuracy for classification, RMSE for regression)
- Interpret variable importance with caution, especially with correlated predictors
- Consider computational requirements when choosing between different ensemble methods
- Document data preparation steps to ensure reproducibility