

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

```
1 # Standard scaling (mean=0, sd=1)
2 scaled_data = scale(data)
3
4 # Apply scaling to specific columns
5 df[,1:4] = apply(df[,1:4], 2, scale)
6
7 # In PCA
8 pca.unscaled = prcomp(x)
9 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

```
1 # 1. Deletion
2 clean_data = na.omit(data)
3
4 # 2. Mean imputation (simple)
5 data$var[is.na(data$var)] = mean(data$var, na.rm=TRUE)
6
7 # 3. Group-specific imputation
8 # For gender-specific means:
9 m_mean = mean(data$var[data$sex==1], na.rm=TRUE)
10 f_mean = mean(data$var[data$sex==2], na.rm=TRUE)
11
12 # Apply to missing values by group
13 data$var[is.na(data$var) & data$sex==1] = m_mean
14 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

```
1 # K-fold Cross-Validation
2 set.seed(4060)
3 k = 10
4 n = nrow(data)
5 folds = sample(rep(1:k, length.out = n))
6 results = numeric(k)
7
8 for(i in 1:k){
9   test_ind = which(folds == i)
10  train_data = data[-test_ind, ]
11  test_data = data[test_ind, ]
12  model = lm(y ~ x, data=train_data)
13  # Store metric (e.g., RMSE, coefficient)
14  results[i] = ...
15 }
16 cv_estimate = mean(results)
```

Shuffling Datasets

```
1 # Shuffle dataset rows randomly
2 set.seed(123)
3 shuffled_data = data[sample(1:nrow(data)), ]
4
5 # Shuffle specific vectors
6 x_shuffled = x[sample(1:length(x))]
7 y_shuffled = y[sample(1:length(y))]
```

Bootstrapping

```
1 # Bootstrapping
2 set.seed(4060)
3 B = 100
4 results = numeric(B)
5 N = nrow(data)
6
7 for(i in 1:B){
8   # Sample with replacement
9   boot_indices = sample(1:N, N, replace=TRUE)
10  boot_data = data[boot_indices, ]
11  model = lm(y ~ x, data=boot_data)
12  # Store result
13  results[i] = ...
14 }
15 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

```
1 # Ridge regression (L2 penalty, alpha=0)
2 ridge = glmnet(x, y, alpha=0, lambda=lambda_values)
3
4 # LASSO regression (L1 penalty, alpha=1)
5 lasso = glmnet(x, y, alpha=1, lambda=lambda_values)
6
7 # Elastic Net (both L1 and L2, 0<alpha<1)
8 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

```
1 # Create lambda sequence
2 lambda_grid = 10^seq(10, -2, length=100)
3
4 # Cross-validation for optimal lambda
5 cv_ridge = cv.glmnet(X_train, y_train, alpha=0,
6                     lambda=lambda_grid)
7 best_lambda_ridge = cv_ridge$lambda.min
8
9 # Alternative: 1SE rule for sparser models
10 lambda_1se = cv_ridge$lambda.1se
```

Implementation

Model Fitting and Prediction

```
1 # Data preparation
2 x = model.matrix(Salary~., data=dat)[,-1]
3 y = dat$Salary
4
5 # Fitting final model with optimal lambda
6 ridge_model = glmnet(X_train, y_train, alpha=0,
7                     lambda=best_lambda_ridge)
8
9 # Getting coefficients
10 coef_ridge = predict(ridge_model,
11                     type="coefficients",
12                     s=best_lambda_ridge)
13
14 # Making predictions
15 pred_ridge = predict(ridge_model,
16                     s=best_lambda_ridge,
17                     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

```
1 # Train/test split (70/30)
2 n = nrow(data)
3 train_idx = sample(1:n, size=0.7*n)
4 train = data[train_idx, ]
5 test = data[-train_idx, ]
6
7 # Model matrices for glmnet
8 X_train = model.matrix(y~., train)[,-1]
9 X_test = model.matrix(y~., test)[,-1]
10
11 # Evaluation metric: MSE
12 mse = function(actual, pred) mean((actual-pred)^2)
13 ridge_mse = mse(y_test, ridge_pred)
14 lasso_mse = mse(y_test, lasso_pred)
15 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

```
1 # KNN classification
2 library(class)
3
4 # Split data
5 i.train = sample(1:n, 100)
6 x.train = x[i.train, ]
7 x.test = x[-i.train, ]
8 y.train = y[i.train]
9 y.test = y[-i.train]
10
11 # Run with K=5
12 k = 5
13 knn_pred = knn(x.train, x.test, y.train, k)
14
15 # Evaluate
16 conf_matrix = table(knn_pred, y.test)
17 accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
18
19 # Find optimal K
20 Kmax = 30
21 acc = numeric(Kmax)
22 for(k in 1:Kmax){
23   knn_pred = knn(x.train, x.test, y.train, k)
24   conf_matrix = table(knn_pred, y.test)
25   acc[k] = sum(diag(conf_matrix)) / sum(conf_matrix)
26 }
27 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

```
1 # Logistic regression
2 glm_model = glm(response ~ predictors,
3                 data=train_data,
4                 family=binomial(logit))
5
6 # Predictions (probabilities)
7 pred_prob = predict(glm_model, newdata=test_data,
8                     type="response")
9
10 # Class predictions with threshold 0.5
11 pred_class = ifelse(pred_prob > 0.5, 1, 0)
12
13 # Confusion matrix
14 library(caret)
15 cm = confusionMatrix(
16   data=as.factor(pred_class),
17   reference=as.factor(test_data$response),
18   positive="1"
19 )
```

Threshold Selection:

Testing Different Thresholds

```
1 # Test different thresholds
2 thresholds = seq(0.1, 0.9, by=0.1)
3 sensitivity = numeric(length(thresholds))
4
5 for(i in 1:length(thresholds)){
6   pred_class = ifelse(pred_prob > thresholds[i], 1, 0)
7   cm = confusionMatrix(
8     data=as.factor(pred_class),
9     reference=as.factor(test_data$response),
10    positive="1"
11  )
12  sensitivity[i] = cm$byClass["Sensitivity"]
13 }
14 plot(thresholds, sensitivity, type="b")
```

Discriminant Analysis

LDA and QDA Implementation

```
1 library(MASS)
2
3 # Linear Discriminant Analysis
4 lda_model = lda(Species ~ ., data=train_data)
5 lda_pred = predict(lda_model, newdata=test_data)
6 lda_class = lda_pred$class
7 lda_post = lda_pred$posterior # class probabilities
8
9 # Quadratic Discriminant Analysis
10 qda_model = qda(Species ~ ., data=train_data)
11 qda_pred = predict(qda_model, newdata=test_data)
```

LDA Assumptions Testing:

Testing Normality and Equal Variance

```
1 # Equal variance: Bartlett's test
2 # H0: All group variances are equal
3 for(j in 1:ncol(predictors)){
4   bartlett.test(data[,j] ~ data$group)
5 }
6
7 # Normality: Shapiro-Wilk test
8 # H0: Data follows normal distribution
9 for(j in 1:ncol(predictors)){
10   shapiro.test(data[data$group=="A", j])
11   shapiro.test(data[data$group=="B", j])
12 }
13
14 # Visual checks: Normal QQ plots
15 for(j in 1:ncol(predictors)){
16   x1 = data[data$group=="A", j]
17   qqnorm(x1, main=names(data)[j])
18   qqline(x1)
19 }
```

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

```
1 # Confusion matrix-based metrics
2 TP = conf_matrix[2,2] # True positives
3 TN = conf_matrix[1,1] # True negatives
4 FP = conf_matrix[2,1] # False positives
5 FN = conf_matrix[1,2] # False negatives
6
7 accuracy = (TP + TN) / (TP + TN + FP + FN)
8 sensitivity = TP / (TP + FN) # Recall
9 specificity = TN / (TN + FP)
10 precision = TP / (TP + FP)
11 F1 = 2 * precision * sensitivity / (precision + sensitivity)
```

ROC Analysis

```
1 library(pROC)
2
3 # ROC curve and AUC
4 roc_obj = roc(response, predicted_probs)
5 auc_value = auc(roc_obj)
6 plot(roc_obj)
7
8 # Manual ROC curve
9 thresholds = seq(0.05, 0.95, by=0.05)
10 sensitivity = specificity = numeric(length(thresholds))
11
12 for(i in 1:length(thresholds)){
13   pred_class = ifelse(pred_prob > thresholds[i], 1, 0)
14   cm = table(pred_class, true_class)
15
16   # True positive rate (Sensitivity)
17   sensitivity[i] = cm[2,2] / sum(cm[,2])
18
19   # True negative rate (Specificity)
20   specificity[i] = cm[1,1] / sum(cm[,1])
21 }
22
23 # Plot ROC curve
24 plot(1-specificity, sensitivity, type="l")
25 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

```
1 library(pROC)
2
3 # Calculate AUC for multiple models
4 glm_probs = predict(glm_model, newdata=test_data, type="response")
5 lda_probs = predict(lda_model, newdata=test_data)$posterior[,2]
6 qda_probs = predict(qda_model, newdata=test_data)$posterior[,2]
7
8 # Compute ROC objects
9 roc_glm = roc(test_data$response, glm_probs)
10 roc_lda = roc(test_data$response, lda_probs)
11 roc_qda = roc(test_data$response, qda_probs)
12
13 # Get AUC values
14 auc_glm = auc(roc_glm)
15 auc_lda = auc(roc_lda)
16 auc_qda = auc(roc_qda)
17
18 # Compare models
19 print(auc_glm)
20 print(auc_lda)
21 print(auc_qda)
22
23 # Plot multiple ROC curves
24 plot(roc_glm, col="red")
25 plot(roc_lda, col="blue", add=TRUE)
26 plot(roc_qda, col="green", add=TRUE)
27 legend("bottomright", legend=c("GLM", "LDA", "QDA"),
28       col=c("red", "blue", "green"), lwd=2)
```

Optimal Threshold Selection

```
1 # Find optimal threshold using different methods
2
3 # 1. Youden's J statistic (maximizes sensitivity+specificity-1)
4 roc_obj = roc(response, predicted_probs)
5 coords = coords(roc_obj, "best", ret="threshold", best.method="youden")
6 best_threshold = coords[1]
7
8 # 2. Closest point to (0,1) on ROC curve
9 coords = coords(roc_obj, "best", ret="threshold",
10               best.method="closest.topleft")
11 best_threshold = coords[1]
12
13 # 3. Cost-based approach (when FP and FN have different costs)
14 # Example: FN costs 3x more than FP
15 cost_ratio = 3 # cost(FN)/cost(FP)
16 weights = c(1, cost_ratio)
17 coords = coords(roc_obj, "best", ret="threshold",
18               best.method="youden", best.weights=weights)
19 best_threshold = coords[1]
20
21 # Apply optimal threshold
22 optimal_preds = ifelse(predicted_probs > best_threshold, 1, 0)
```


ROC for Multi-class Problems

```
1 # For multi-class problems: One-vs-All (OVA) approach
2 library(pROC)
3
4 # Assuming 3 classes in posterior probabilities
5 class_labels = c("A", "B", "C")
6 true_labels = test_data$class # factor with levels A, B, C
7 posterior_probs = predict(model, newdata=test_data)$posterior
8
9 # Calculate ROC for each class
10 auc_values = numeric(length(class_labels))
11
12 for(i in 1:length(class_labels)){
13   # Create binary problem: class i vs rest
14   binary_true = ifelse(true_labels == class_labels[i], 1, 0)
15
16   # Get probability for this class
17   class_prob = posterior_probs[,i]
18
19   # Calculate ROC and AUC
20   roc_obj = roc(binary_true, class_prob)
21   auc_values[i] = auc(roc_obj)
22
23   # Plot ROC curve
24   if(i == 1) plot(roc_obj, col=i)
25   else plot(roc_obj, col=i, add=TRUE)
26 }
27
28 # Average AUC across all classes (macro-averaging)
29 mean_auc = mean(auc_values)
```

Cross-Validated AUC

```
1 # K-fold cross-validated AUC
2 set.seed(123)
3 K = 10
4 n = nrow(data)
5 folds = cut(1:n, K, labels=FALSE)
6 cv_auc = numeric(K)
7
8 for(k in 1:K){
9   # Split into train and test
10   train_idx = which(folds != k)
11   test_idx = which(folds == k)
12
13   # Train model
14   model = glm(y ~ ., data=data[train_idx,], family=binomial)
15
16   # Make predictions on test fold
17   probs = predict(model, data=data[test_idx,], type="response")
18
19   # Calculate AUC for this fold
20   if(length(unique(data$y[test_idx])) > 1) {
21     # Only calculate if both classes present in test set
22     roc_obj = roc(data$y[test_idx], probs)
23     cv_auc[k] = auc(roc_obj)
24   }
25 }
26
27 # Mean and SD of cross-validated AUC
28 mean_cv_auc = mean(cv_auc, na.rm=TRUE)
29 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

```
1 # K-fold CV to compare classification models
2 set.seed(123)
3 K = 10
4 folds = cut(1:nrow(data), K, labels=FALSE)
5 acc.knn = acc.glm = acc.lda = acc.qda = numeric(K)
6 auc.knn = auc.glm = auc.lda = auc.qda = numeric(K)
7
8 for(k in 1:K){
9   # Split data
10  train_idx = which(folds != k)
11  test_idx = which(folds == k)
12  train_data = data[train_idx, ]
13  test_data = data[test_idx, ]
14
15  # Fit models
16  knn_pred = knn(train_data[, -1], test_data[, -1],
17                train_data[, 1], k=5)
18  glm_model = glm(y~., data=train_data, family=binomial)
19  lda_model = lda(y~., data=train_data)
20  qda_model = qda(y~., data=train_data)
21
22  # Make predictions
23  glm_prob = predict(glm_model, test_data, type="response")
24  lda_prob = predict(lda_model, test_data)$posterior[,2]
25  qda_prob = predict(qda_model, test_data)$posterior[,2]
26
27  # Calculate metrics
28  acc.knn[k] = mean(knn_pred == test_data[,1])
29  acc.glm[k] = mean((glm_prob > 0.5) == test_data[,1])
30  acc.lda[k] = mean(predict(lda_model, test_data)$class
31                    == test_data[,1])
32  acc.qda[k] = mean(predict(qda_model, test_data)$class
33                    == test_data[,1])
34
35  # AUC values
36  auc.glm[k] = auc(roc(test_data[,1], glm_prob))
37  auc.lda[k] = auc(roc(test_data[,1], lda_prob))
38  auc.qda[k] = auc(roc(test_data[,1], qda_prob))
39 }
40
41 # Compare performances
42 par(mfrow=c(1,2))
43 boxplot(acc.knn, acc.glm, acc.lda, acc.qda,
44         names=c("KNN", "LogReg", "LDA", "QDA"))
45 boxplot(auc.glm, auc.lda, auc.qda,
46         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

```
1 library(tree)
2 library(ISLR)
3
4 # Prepare data (binary classification example)
5 High = ifelse(Carseats$Sales <= 8, "No", "Yes")
6 CS = data.frame(Carseats, High=as.factor(High))
7 CS$Sales = NULL # Remove original response
8
9 # Grow the tree
10 tree_model = tree(High ~ ., data=CS)
11 summary(tree_model)
12
13 # Visualize
14 plot(tree_model)
15 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 λ , find the subtree that minimizes: $RSS + \lambda \times (\text{number of terminal nodes})$
- As λ increases, we get smaller trees
- Cross-validation is used to select the optimal

Pruning Trees with Cross-Validation

```
1 # Cross-validation for optimal tree size
2 cv_tree = cv.tree(tree_model, FUN=prune.misclass)
3
4 # Plot CV error vs. tree size
5 plot(cv_tree$size, cv_tree$dev, type="b",
6      xlab="Tree Size", ylab="CV Error")
7 abline(v=cv_tree$size[which.min(cv_tree$dev)])
8
9 # Get optimal size and prune
10 opt_size = cv_tree$size[which.min(cv_tree$dev)]
11 pruned_tree = prune.misclass(tree_model, best=opt_size)
12
13 # Compare original and pruned trees
14 par(mfrow=c(1,2))
15 plot(tree_model); text(tree_model, pretty=0)
16 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

```
1 # Train/test split
2 set.seed(123)
3 train_idx = sample(1:nrow(CS), 200)
4 train_data = CS[train_idx,]
5 test_data = CS[-train_idx,]
6
7 # Build tree on training data
8 tree_model = tree(High ~ ., data=train_data)
9
10 # Class predictions
11 class_pred = predict(tree_model, test_data, type="class")
12 conf_matrix = table(class_pred, test_data$High)
13 accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
14 print(conf_matrix)
15 cat("Accuracy:", accuracy, "\n")
16
17 # Probability predictions for ROC analysis
18 prob_pred = predict(tree_model, test_data, type="vector")
19 library(pROC)
20 roc_obj = roc(response=test_data$High, predictor=prob_pred[,1])
21 auc_value = auc(roc_obj)
22 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 $m_{try} = p$ (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

```
1 library(randomForest)
2
3 # Training data preparation
4 set.seed(123)
5 train_idx = sample(1:nrow(CS), 200)
6 train_data = CS[train_idx,]
7 test_data = CS[-train_idx,]
8
9 # Number of predictors
10 p = ncol(train_data) - 1
11
12 # Random Forest (mtry defaults to sqrt(p) for classification)
13 rf_model = randomForest(High ~ .,
14                           data=train_data,
15                           ntree=500, # number of trees
16                           importance=TRUE) # compute variable importance
17
18 # Bagging (special case of RF with mtry=p)
19 bag_model = randomForest(High ~ .,
20                           data=train_data,
21                           mtry=p, # use all predictors
22                           importance=TRUE)
23
24 # Print models to see OOB error
25 print(rf_model)
26 print(bag_model)
27
28 # Predictions
29 rf_pred = predict(rf_model, test_data)
30 bag_pred = predict(bag_model, test_data)
31
32 # Evaluation
33 rf_conf = table(rf_pred, test_data$High)
34 bag_conf = table(bag_pred, test_data$High)
35
36 # Accuracy
37 rf_acc = sum(diag(rf_conf)) / sum(rf_conf)
38 bag_acc = sum(diag(bag_conf)) / sum(bag_conf)
39 cat("RF Accuracy:", rf_acc, "\n")
40 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

```
1 # Variable importance measures
2 rf_importance = importance(rf_model)
3 print(rf_importance)
4
5 # Plot variable importance
6 varImpPlot(rf_model, main="Random Forest Variable Importance",
7            type=1) # type=1 for Mean Decrease Accuracy
8
9 # Compare variable importance between RF and Bagging
10 par(mfrow=c(1,2))
11 varImpPlot(rf_model, main="RF Importance")
12 varImpPlot(bag_model, main="Bagging Importance")
13
14 # Partial dependence plots (effect of predictors)
15 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

```
1 library(gbm)
2
3 # For binary classification
4 # Convert response to numeric 0/1
5 train_data_gbm = train_data
6 train_data_gbm$High = as.numeric(train_data$High == "Yes")
7
8 # Gradient Boosting Machine
9 gb_model = gbm(High ~ .,
10                data=train_data_gbm,
11                distribution="bernoulli", # for classification
12                n.trees=5000,           # ensemble size
13                interaction.depth=2,     # tree depth (1=stumps)
14                shrinkage=0.01,         # learning rate
15                cv.folds=5)             # for CV
16
17 # Find optimal number of trees
18 best_iter = gbm.perf(gb_model)
19 print(paste("Optimal number of trees:", best_iter))
20
21 # Variable importance
22 summary(gb_model)
23
24 # Partial dependence plots
25 plot(gb_model, i="Price")
26
27 # Predictions
28 test_data_gbm = test_data
29 gb_prob = predict(gb_model,
30                  newdata=test_data_gbm,
31                  n.trees=best_iter,
32                  type="response") # probabilities
33
34 # Convert to class predictions
35 gb_class = ifelse(gb_prob > 0.5, "Yes", "No")
36
37 # Confusion matrix
38 conf_matrix = table(gb_class, test_data$High)
39 accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
40 print(conf_matrix)
41 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

```
1 # Fit a tree to continuous data
2 data = data.frame(x=x, y=y)
3 tree_model = tree(y ~ x, data=data)
4
5 # Extract split points
6 split_text = tree_model$frame$splits[,1]
7 split_values = as.numeric(na.omit(
8   unlist(strsplit(split_text, "<"))
9 ))
10 cut_points = c(min(x), sort(split_values), max(x))
11
12 # Function to demonstrate piecewise prediction
13 plot(x, y, pch=20)
14 for(i in 1:(length(cut_points)-1)) {
15   subset_idx = which(x >= cut_points[i] &
16                     x < cut_points[i+1])
17   if(length(subset_idx) > 0) {
18     # Fit a model in each region
19     region_model = lm(y ~ x, data=data[subset_idx,])
20
21     # Get predictions at region boundaries
22     pred_x = c(cut_points[i], cut_points[i+1])
23     pred_y = predict(region_model,
24                     newdata=data.frame(x=pred_x))
25
26     # Draw line segment
27     segments(x0=pred_x[1], x1=pred_x[2],
28             y0=pred_y[1], y1=pred_y[2],
29             col="blue", lwd=2)
30   }
31 }
```

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

```
1 # Compare tree-based methods with other classifiers
2 set.seed(123)
3 K = 10 # Number of folds
4 folds = cut(1:nrow(data), K, labels=FALSE)
5
6 # Accuracy storage
7 acc_tree = acc_rf = acc_gbm = acc_glm = numeric(K)
8 auc_tree = auc_rf = auc_gbm = auc_glm = numeric(K)
9
10 for(k in 1:K) {
11   # Split data
12   train_idx = which(folds != k)
13   test_idx = which(folds == k)
14   train_data = data[train_idx, ]
15   test_data = data[test_idx, ]
16
17   # Fit models
18   tree_model = tree(response ~ ., data=train_data)
19   rf_model = randomForest(response ~ ., data=train_data)
20
21   # Convert response for GBM if needed
22   train_data_gbm = train_data
23   if(is.factor(train_data$response)) {
24     train_data_gbm$response = as.numeric(train_data$response == levels(train_data$
25       response)[2])
26   }
27
28   gbm_model = gbm(response ~ ., data=train_data_gbm,
29     distribution="bernoulli", n.trees=1000,
30     interaction.depth=2, shrinkage=0.01)
31
32   glm_model = glm(response ~ ., data=train_data,
33     family=binomial)
34
35   # Predictions
36   tree_prob = predict(tree_model, test_data, type="vector")[,2]
37   rf_prob = predict(rf_model, test_data, type="prob")[,2]
38
39   test_data_gbm = test_data
40   gbm_prob = predict(gbm_model, test_data_gbm, n.trees=1000,
41     type="response")
42   glm_prob = predict(glm_model, test_data, type="response")
43
44   # Class predictions (threshold 0.5)
45   tree_pred = factor(ifelse(tree_prob > 0.5, 1, 0))
46   rf_pred = factor(ifelse(rf_prob > 0.5, 1, 0))
47   gbm_pred = factor(ifelse(gbm_prob > 0.5, 1, 0))
48   glm_pred = factor(ifelse(glm_prob > 0.5, 1, 0))
49
50   # Accuracy
51   acc_tree[k] = mean(tree_pred == test_data$response)
52   acc_rf[k] = mean(rf_pred == test_data$response)
53   acc_gbm[k] = mean(gbm_pred == test_data$response)
54   acc_glm[k] = mean(glm_pred == test_data$response)
55
56   # AUC
57   auc_tree[k] = auc(roc(test_data$response, tree_prob))
58   auc_rf[k] = auc(roc(test_data$response, rf_prob))
59   auc_gbm[k] = auc(roc(test_data$response, gbm_prob))
60   auc_glm[k] = auc(roc(test_data$response, glm_prob))
61 }
62
63 # Compare model performance
64 par(mfrow=c(1,2))
65 boxplot(acc_tree, acc_rf, acc_gbm, acc_glm,
66   names=c("Tree", "RF", "GBM", "GLM"),
67   main="Accuracy Comparison")
68
69 boxplot(auc_tree, auc_rf, auc_gbm, auc_glm,
70   names=c("Tree", "RF", "GBM", "GLM"),
71   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

```
1 # Load packages
2 library(e1071) # Main package for SVM implementation
3
4 # Prepare data
5 set.seed(123)
6 train_idx <- sample(nrow(iris), 0.7*nrow(iris))
7 train_data <- iris[train_idx, ]
8 test_data <- iris[-train_idx, ]
9
10 # Fit SVM model with radial kernel
11 svm_model <- svm(Species ~ .,
12                 data = train_data,
13                 kernel = "radial",
14                 cost = 1,
15                 scale = TRUE)
16
17 # Model summary
18 summary(svm_model)
19
20 # Make predictions
21 predictions <- predict(svm_model, test_data)
22
23 # Evaluate model
24 confusion_matrix <- table(
25   Predicted = predictions,
26   Actual = test_data$Species
27 )
28 print(confusion_matrix)
29 accuracy <- sum(diag(confusion_matrix))/sum(confusion_matrix)
30 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

```
1 # Prepare dataset with categorical variables
2 data(mtcars)
3 head(mtcars)
4
5 # Create categorical variables
6 mtcars$vs_factor <- as.factor(mtcars$vs)
7 mtcars$am_factor <- as.factor(mtcars$am)
8
9 # Convert to numeric using one-hot encoding
10 formula <- ~ vs_factor + am_factor
11 model_matrix <- model.matrix(formula, mtcars)[,-1] # Drop intercept
12 head(model_matrix)
13
14 # Combine with numeric predictors
15 X <- cbind(mtcars[, c("mpg", "disp", "hp")], model_matrix)
16 y <- mtcars$cyl
17
18 # Now use in SVM
19 svm_model <- svm(y ~ ., data = data.frame(X, y = as.factor(y)))
20 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

```

1 # Generate 2D nonlinear data
2 library(e1071)
3 set.seed(123)
4 x <- matrix(rnorm(200*2), ncol = 2)
5 x[1:100,] <- x[1:100,] + 2
6 x[101:150,] <- x[101:150,] - 2
7 y <- c(rep(1,150), rep(2,50))
8 data <- data.frame(x = x, y = as.factor(y))
9
10 # Split into train/test
11 train_idx <- sample(nrow(data), 0.7*nrow(data))
12 train_data <- data[train_idx, ]
13 test_data <- data[-train_idx, ]
14
15 # Compare different kernels
16 kernels <- c("linear", "polynomial", "radial", "sigmoid")
17 results <- data.frame(
18   Kernel = kernels,
19   Accuracy = numeric(length(kernels))
20 )
21
22 for (i in 1:length(kernels)) {
23   svm_model <- svm(y ~ .,
24     data = train_data,
25     kernel = kernels[i],
26     cost = 1)
27
28   predictions <- predict(svm_model, test_data)
29   confusion <- table(Predicted = predictions, Actual = test_data$y)
30   accuracy <- sum(diag(confusion))/sum(confusion)
31   results$Accuracy[i] <- accuracy
32 }
33
34 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

```
1 # Grid search for best parameters using tune()
2 library(e1071)
3 data(iris)
4
5 # Define parameter ranges to search
6 tuned <- tune.svm(Species ~ .,
7                   data = iris,
8                   kernel = "radial",
9                   cost = 10^(-1:2), # 0.1, 1, 10, 100
10                  gamma = 10^(-2:1)) # 0.01, 0.1, 1, 10
11
12 # Best model parameters
13 print(tuned$best.parameters)
14
15 # Performance at best parameters
16 print(tuned$best.performance)
17
18 # Full performance table
19 head(tuned$performances)
20
21 # Visualize parameter tuning results
22 plot(tuned)
23
24 # Use the best model
25 best_model <- tuned$best.model
26 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

```
1 # Multi-class SVM with One-vs-One strategy (default in e1071)
2 library(e1071)
3 data(iris)
4
5 # Split data
6 set.seed(123)
7 train_idx <- sample(nrow(iris), 0.7*nrow(iris))
8 train_data <- iris[train_idx, ]
9 test_data <- iris[-train_idx, ]
10
11 # Fit multi-class SVM
12 svm_model <- svm(Species ~ .,
13                 data = train_data,
14                 kernel = "radial",
15                 cost = 1,
16                 probability = TRUE) # For class probabilities
17
18 # Prediction type
19 # - type="response" gives class labels (default)
20 # - type="probabilities" gives class probabilities
21 predictions <- predict(svm_model, test_data, type="response")
22 prob_predictions <- predict(svm_model, test_data, probability=TRUE)
23
24 # Extract probabilities
25 prob_attr <- attr(prob_predictions, "probabilities")
26 head(prob_attr)
27
28 # Evaluate model
29 confusion <- table(Predicted = predictions, Actual = test_data$Species)
30 print(confusion)
31 accuracy <- sum(diag(confusion))/sum(confusion)
32 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

```
1 # SVM for regression (SVR)
2 library(e1071)
3 data(mtcars)
4
5 # Scale predictors
6 mtcars_scaled <- mtcars
7 mtcars_scaled[, -1] <- scale(mtcars_scaled[, -1])
8
9 # Split data
10 set.seed(123)
11 train_idx <- sample(nrow(mtcars_scaled), 0.7*nrow(mtcars_scaled))
12 train_data <- mtcars_scaled[train_idx, ]
13 test_data <- mtcars_scaled[-train_idx, ]
14
15 # Fit SVM regression model
16 svr_model <- svm(mpg ~ .,
17                 data = train_data,
18                 kernel = "radial",
19                 cost = 10,
20                 epsilon = 0.1, # Width of epsilon-insensitive tube
21                 type = "eps-regression")
22
23 # Make predictions
24 predictions <- predict(svr_model, test_data)
25
26 # Evaluate model
27 mse <- mean((test_data$mpg - predictions)^2)
28 rmse <- sqrt(mse)
29 print(paste("RMSE:", round(rmse, 4)))
30
31 # Visualize actual vs predicted
32 plot(test_data$mpg, predictions,
33      main="Actual vs Predicted MPG",
34      xlab="Actual", ylab="Predicted")
35 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

```
1 # Comparing SVM models with different kernels and parameters
2 library(e1071)
3 library(pROC)
4 library(caret)
5
6 # Prepare data
7 data(iris)
8 iris_binary <- iris[iris$Species != "virginica", ] # Make binary for ROC
9 iris_binary$Species <- factor(iris_binary$Species)
10
11 # Split data
12 set.seed(123)
13 train_idx <- sample(nrow(iris_binary), 0.7*nrow(iris_binary))
14 train_data <- iris_binary[train_idx, ]
15 test_data <- iris_binary[-train_idx, ]
16
17 # Define models to compare
18 models <- list(
19   linear_svm = svm(Species ~ ., data = train_data, kernel = "linear", cost = 1, probability = TRUE),
20   radial_svm = svm(Species ~ ., data = train_data, kernel = "radial", cost = 1, probability = TRUE),
21   poly_svm = svm(Species ~ ., data = train_data, kernel = "polynomial", degree = 2, cost = 1,
22     probability = TRUE)
23 )
24
25 # Evaluate each model
26 results <- data.frame(
27   Model = names(models),
28   Accuracy = numeric(length(models)),
29   AUC = numeric(length(models))
30 )
31
32 roc_curves <- list()
33
34 for (i in 1:length(models)) {
35   # Predictions
36   model_name <- names(models)[i]
37   model <- models[[i]]
38
39   # Class predictions
40   pred_class <- predict(model, test_data)
41   confusion <- confusionMatrix(pred_class, test_data$Species)
42
43   # Probability predictions
44   pred_prob <- predict(model, test_data, probability = TRUE)
45   prob <- attr(pred_prob, "probabilities")[, 2] # Probability of second class
46
47   # ROC curve
48   roc_obj <- roc(test_data$Species, prob)
49   roc_curves[[model_name]] <- roc_obj
50
51   # Store results
52   results$Accuracy[i] <- confusion$overall["Accuracy"]
53   results$AUC[i] <- auc(roc_obj)
54 }
55
56 # Display results
57 print(results)
58
59 # Plot ROC curves for comparison
60 plot(roc_curves[[1]], main="ROC Curve Comparison")
61 for (i in 2:length(roc_curves)) {
62   plot(roc_curves[[i]], add=TRUE, col=i)
63 }
64 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

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

```
1 # Implementation using neuralnet package
2 library(neuralnet)
3
4 # Prepare data
5 set.seed(123)
6 n <- nrow(iris)
7 data <- iris[sample(1:n), ] # Shuffle dataset
8
9 # Train neural network with 2 hidden layers (6 and 5 neurons)
10 nn_model <- neuralnet(Species ~ .,
11                       data=data,
12                       hidden=c(6, 5)) # Two hidden layers
13
14 # Visualize the network
15 plot(nn_model,
16      information=FALSE,
17      col.entry='red',
18      col.out='green',
19      show.weights=FALSE)
20
21 # Prediction
22 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

```
1 # Function for min-max scaling (normalize to [0,1])
2 minmax_scale <- function(x) {
3   return((x - min(x)) / (max(x) - min(x)))
4 }
5
6 # Prepare dataset
7 data(Boston)
8 set.seed(123)
9 n <- nrow(Boston)
10 train_idx <- sample(1:n, round(0.7*n))
11
12 # Create scaled version for training
13 boston_scaled <- Boston
14 # Scale predictors
15 boston_scaled[, -14] <- apply(boston_scaled[, -14], 2, minmax_scale)
16 # Scale response (alternative approach - divide by max value)
17 boston_scaled$medv <- boston_scaled$medv / 50 # Max value is around 50
18
19 # Handle categorical variables (if present)
20 # Convert factors to numeric (0, 1, ...)
21 recode_factor <- function(x) {
22   if(is.factor(x)) {
23     return(as.numeric(x) - 1) # Subtract 1 to start from 0
24   } else {
25     return(x)
26   }
27 }
28
29 # Apply recoding to each column if needed
30 # boston_scaled <- data.frame(lapply(boston_scaled, recode_factor))
31
32 # Split into train/test
33 train_data <- boston_scaled[train_idx, ]
34 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

Different Neural Network Packages

```
1 # Using nnet for single-layer network
2 library(nnet)
3 set.seed(123)
4
5 # Train model
6 nnet_model <- nnet(medv ~ .,
7                   data=train_data,
8                   size=5,      # 5 nodes in hidden layer
9                   decay=0.01,  # Weight decay parameter
10                  linout=TRUE)  # Linear output for regression
11
12 # Predictions with nnet
13 nnet_preds <- predict(nnet_model, test_data) * 50 # Scale back
14
15 # Using neuralnet for multi-layer network
16 library(neuralnet)
17 set.seed(123)
18
19 # Prepare formula
20 predictors <- names(train_data)[-which(names(train_data) == "medv")]
21 formula <- as.formula(paste("medv ~", paste(predictors, collapse=" + ")))
22
23 # Train model
24 neuralnet_model <- neuralnet(formula,
25                             data=train_data,
26                             hidden=c(5, 3), # Two hidden layers
27                             linear.output=TRUE) # For regression
28
29 # Predictions with neuralnet
30 nn_preds <- predict(neuralnet_model, test_data) * 50 # Scale back
31
32 # Compare RMSE
33 nnet_rmse <- sqrt(mean((nnet_preds - Boston$medv[-train_idx])^2))
34 nn_rmse <- sqrt(mean((nn_preds - Boston$medv[-train_idx])^2))
35 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

```
1 # Exploring the effect of hidden layer size
2 library(nnet)
3 set.seed(4061)
4
5 # Data preparation (iris dataset)
6 data(iris)
7 n <- nrow(iris)
8 iris_shuffled <- iris[sample(1:n), ]
9 # Scale predictors to [0,1]
10 iris_scaled <- iris_shuffled
11 iris_scaled[, 1:4] <- scale(iris_scaled[, 1:4],
12                             center=apply(iris_scaled[, 1:4], 2, min),
13                             scale=apply(iris_scaled[, 1:4], 2, max) -
14                             apply(iris_scaled[, 1:4], 2, min))
15
16 # Split data
17 train_idx <- sample(1:n, round(0.7*n))
18 train_data <- iris_scaled[train_idx, ]
19 test_data <- iris_scaled[-train_idx, ]
20
21 # Test different network sizes
22 sizes <- c(1, 3, 5, 10, 15)
23 train_accuracy <- numeric(length(sizes))
24 test_accuracy <- numeric(length(sizes))
25
26 for(i in 1:length(sizes)) {
27   # Train model with current size
28   nn_model <- nnet(Species ~ .,
29                     data=train_data,
30                     size=sizes[i],
31                     decay=0.01,
32                     maxit=500)
33
34   # Training accuracy
35   train_pred <- predict(nn_model, train_data, type="class")
36   train_accuracy[i] <- mean(train_pred == train_data$Species)
37
38   # Test accuracy
39   test_pred <- predict(nn_model, test_data, type="class")
40   test_accuracy[i] <- mean(test_pred == test_data$Species)
41 }
42
43 # Test different decay values (regularization)
44 decays <- c(0, 0.001, 0.01, 0.1, 0.5)
45 train_decay_accuracy <- numeric(length(decays))
46 test_decay_accuracy <- numeric(length(decays))
47
48 for(i in 1:length(decays)) {
49   # Train model with optimal size and current decay
50   nn_model <- nnet(Species ~ .,
51                     data=train_data,
52                     size=5, # Using a fixed size
53                     decay=decays[i],
54                     maxit=500)
55
56   # Training accuracy
57   train_pred <- predict(nn_model, train_data, type="class")
58   train_decay_accuracy[i] <- mean(train_pred == train_data$Species)
59
60   # Test accuracy
61   test_pred <- predict(nn_model, test_data, type="class")
62   test_decay_accuracy[i] <- mean(test_pred == test_data$Species)
63 }
64
65 # Plot results
66 par(mfrow=c(1,2))
67 plot(sizes, train_accuracy, type="b", col="blue", ylim=c(min(train_accuracy, test_
68   accuracy), 1),
69       xlab="Hidden Layer Size", ylab="Accuracy", main="Effect of Network Size")
69 lines(sizes, test_accuracy, type="b", col="red")
70 legend("bottomright", legend=c("Train", "Test"), col=c("blue", "red"), lty=1)
71
72 plot(decays, train_decay_accuracy, type="b", col="blue",
73       ylim=c(min(train_decay_accuracy, test_decay_accuracy), 1),
74       xlab="Weight Decay", ylab="Accuracy", main="Effect of Regularization")
75 lines(decays, test_decay_accuracy, type="b", col="red")
76 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

```
1 # Classification with neural networks
2 library(nnet)
3 library(neuralnet)
4 set.seed(123)
5
6 # Prepare data (iris dataset)
7 data(iris)
8 iris_scaled <- iris
9 iris_scaled[, 1:4] <- scale(iris_scaled[, 1:4])
10
11 # Split data
12 n <- nrow(iris_scaled)
13 train_idx <- sample(1:n, round(0.7*n))
14 train_data <- iris_scaled[train_idx, ]
15 test_data <- iris_scaled[-train_idx, ]
16
17 # Single-layer network with nnet
18 nnet_model <- nnet(Species ~ .,
19                   data=train_data,
20                   size=5,
21                   decay=0.01)
22
23 # Get class predictions
24 nnet_class_pred <- predict(nnet_model, test_data, type="class")
25
26 # Get probability predictions
27 nnet_prob_pred <- predict(nnet_model, test_data, type="raw")
28
29 # Evaluate model
30 nnet_confusion <- table(Predicted=nnet_class_pred, Actual=test_data$Species)
31 nnet_accuracy <- sum(diag(nnet_confusion)) / sum(nnet_confusion)
32
33 print(nnet_confusion)
34 print(paste("Accuracy:", round(nnet_accuracy, 4)))
35
36 # Use neuralnet for comparison
37 # Convert Species to dummy variables for neuralnet
38 species_dummies <- model.matrix(~ Species - 1, train_data)
39 train_data_nn <- cbind(train_data[, 1:4], species_dummies)
40
41 # Create formula for neuralnet
42 input_vars <- paste(names(train_data)[1:4], collapse=" + ")
43 output_vars <- paste(colnames(species_dummies), collapse=" + ")
44 nn_formula <- as.formula(paste(output_vars, "~", input_vars))
45
46 # Train neuralnet model
47 nn_model <- neuralnet(nn_formula,
48                      data=train_data_nn,
49                      hidden=c(5),
50                      linear.output=FALSE)
51
52 # Make predictions
53 test_pred_input <- as.matrix(test_data[, 1:4])
54 nn_prediction <- compute(nn_model, test_pred_input)
55
56 # Convert to class predictions
57 predicted_classes <- max.col(nn_prediction$net.result)
58 actual_classes <- as.numeric(test_data$Species)
59
60 # Calculate accuracy
61 nn_accuracy <- mean(predicted_classes == actual_classes)
62 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

```
1 # Regression with neural networks
2 library(nnet)
3 library(MASS) # For Boston housing dataset
4 set.seed(123)
5
6 # Load and prepare data
7 data(Boston)
8 boston_scaled <- Boston
9 # Scale all predictors
10 boston_scaled[, -14] <- scale(boston_scaled[, -14])
11 # Scale the target variable
12 max_medv <- max(boston_scaled$medv)
13 boston_scaled$medv <- boston_scaled$medv / max_medv
14
15 # Split data
16 n <- nrow(boston_scaled)
17 train_idx <- sample(1:n, round(0.7*n))
18 train_data <- boston_scaled[train_idx, ]
19 test_data <- boston_scaled[-train_idx, ]
20
21 # Train neural network for regression
22 # linout=TRUE is critical for regression tasks
23 nn_reg_model <- nnet(medv ~ .,
24                       data=train_data,
25                       size=10,
26                       decay=0.01,
27                       linout=TRUE,
28                       maxit=1000)
29
30 # Make predictions and rescale back
31 nn_pred <- predict(nn_reg_model, test_data) * max_medv
32 actual_values <- Boston$medv[-train_idx]
33
34 # Evaluate performance
35 rmse <- sqrt(mean((nn_pred - actual_values)^2))
36 r_squared <- cor(nn_pred, actual_values)^2
37
38 print(paste("RMSE:", round(rmse, 4)))
39 print(paste("R-squared:", round(r_squared, 4)))
40
41 # Compare with linear regression
42 lm_model <- lm(medv ~ ., data=train_data)
43 lm_pred <- predict(lm_model, test_data) * max_medv
44 lm_rmse <- sqrt(mean((lm_pred - actual_values)^2))
45 lm_r_squared <- cor(lm_pred, actual_values)^2
46
47 print(paste("Linear Regression RMSE:", round(lm_rmse, 4)))
48 print(paste("Linear Regression R-squared:", round(lm_r_squared, 4)))
49
50 # Visualization
51 par(mfrow=c(1,2))
52 plot(actual_values, nn_pred, main="Neural Network",
53       xlab="Actual", ylab="Predicted", pch=19)
54 abline(0, 1, col="red", lwd=2)
55
56 plot(actual_values, lm_pred, main="Linear Regression",
57       xlab="Actual", ylab="Predicted", pch=19)
58 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

```
1 # Variable importance in neural networks using Olden index
2 library(nnet)
3 library(NeuralNetTools)
4 library(randomForest)
5 set.seed(123)
6
7 # Prepare data (Boston Housing)
8 data(Boston)
9 boston_scaled <- Boston
10 # Scale predictors to [0,1]
11 minmax_scale <- function(x) {
12   return((x - min(x)) / (max(x) - min(x)))
13 }
14 boston_scaled[, -14] <- apply(boston_scaled[, -14], 2, minmax_scale)
15 boston_scaled$medv <- Boston$medv / 50 # Scale response
16
17 # Train neural network
18 nn_model <- nnet(medv ~ .,
19                 data=boston_scaled,
20                 size=10,
21                 decay=0.1,
22                 linout=TRUE)
23
24 # Calculate Olden's variable importance
25 olden_importance <- olden(nn_model, bar_plot=FALSE)
26
27 # Train random forest for comparison
28 rf_model <- randomForest(medv ~ ., data=Boston)
29 rf_importance <- importance(rf_model)
30
31 # Normalize importance values for comparison
32 olden_abs <- abs(olden_importance$importance)
33 olden_normalized <- olden_abs / sum(olden_abs)
34 rf_normalized <- rf_importance / sum(rf_importance)
35
36 # Compare importance values
37 importance_comparison <- data.frame(
38   Variable = rownames(olden_importance),
39   NN_Importance = olden_normalized,
40   RF_Importance = rf_normalized
41 )
42
43 # Plot comparison
44 par(mfrow=c(1,2))
45 barplot(olden_normalized,
46         names.arg=rownames(olden_importance),
47         las=2, main="Neural Network Importance",
48         cex.names=0.7)
49
50 barplot(rf_normalized,
51         names.arg=rownames(rf_importance),
52         las=2, main="Random Forest Importance",
53         cex.names=0.7)
54
55 # Alternative: plot combined importance
56 par(mfrow=c(1,1))
57 importance_matrix <- rbind(
58   NN = olden_normalized,
59   RF = rf_normalized
60 )
61 barplot(importance_matrix, beside=TRUE,
62         names.arg=rownames(olden_importance),
63         las=2, legend.text=TRUE,
64         col=c("lightblue", "salmon"),
65         main="Variable Importance Comparison",
66         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

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

```
1 # Demonstrate effect of adding noise variables to Random Forest
2 library(ISLR)
3 library(randomForest)
4
5 set.seed(4061)
6 dat <- na.omit(Hitters)
7 x <- dat
8 x$Salary <- NULL
9 y <- log(dat$Salary)
10 n <- nrow(x)
11 p <- ncol(x)
12
13 # Baseline model
14 rfo1 <- randomForest(x=x, y=y)
15 oob_error_base <- mean(rfo1$mse)
16
17 # Add random noise variables and track performance
18 K <- 10
19 res <- numeric(K)
20 for(k in 1:K) {
21   # Add k*p noise variables
22   x2 <- cbind(x, matrix(rnorm(n*k*p), nrow=n))
23   rfo2 <- randomForest(x=x2, y=y)
24   res[k] <- mean(rfo2$mse)
25 }
26
27 # Plot change in error with increasing noise variables
28 resf <- c(oob_error_base, res)
29 plot(c(0:K), resf, type='b', pch=20,
30      main="OOB MSE with Increasing Noise Variables",
31      xlab="Number of Noise Variable Groups", ylab="OOB MSE")
32
33 # Plot percentage change
34 resp <- (resf-resf[1])/resf[1]*100
35 plot(c(0:K), resp, type='b', pch=20,
36      main="% Change in OOB MSE",
37      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

```
1 library(ISLR)
2 library(leaps) # For regsubsets function
3
4 # Load and prepare data
5 data <- na.omit(Hitters)
6
7 # Best subset selection (exhaustive search)
8 set.seed(4061)
9 reg.full <- regsubsets(Salary ~ ., data=data,
10                       method="exhaustive",
11                       nvmax=19) # Up to 19 variables
12
13 # Examine results
14 summary_res <- summary(reg.full)
15
16 # Plot model fit metrics
17 par(mfrow=c(1,2))
18 # R-squared
19 plot(summary_res$rsq, type="b", pch=20,
20       xlab="Number of Variables", ylab="R-squared")
21
22 # Adjusted R-squared
23 plot(summary_res$adjr2, type="b", pch=20,
24       xlab="Number of Variables", ylab="Adjusted R-squared")
25
26 # Find optimal model by adjusted R-squared
27 best_model_idx <- which.max(summary_res$adjr2)
28 best_model_vars <- summary_res$which[best_model_idx, ]
29
30 # Print selected variables
31 print(names(which(best_model_vars)[-1])) # Exclude intercept
32
33 # Visualize variable selection across models
34 plot(reg.full, scale="adjr2")
```

Forward and Backward Selection

```
1 # Forward selection
2 set.seed(4061)
3 reg.fwd <- regsubsets(Salary ~ ., data=data,
4                       method="forward",
5                       nvmax=19)
6
7 # Backward selection
8 set.seed(4061)
9 reg.bwd <- regsubsets(Salary ~ ., data=data,
10                      method="backward",
11                      nvmax=19)
12
13 # Compare performance
14 par(mfrow=c(1,2))
15 # Compare RSS
16 plot(summary(reg.bwd)$rss, type='b', pch=20,
17       xlab="Number of Variables", ylab="RSS",
18       main="RSS Comparison")
19 points(summary(reg.fwd)$rss, type='b', pch=15, col=2)
20 legend("topright", c("Backward", "Forward"),
21       pch=c(20,15), col=c(1,2))
22
23 # Compare Adjusted R-squared
24 plot(summary(reg.bwd)$adjr2, type='b', pch=20,
25       xlab="Number of Variables", ylab="Adjusted R-squared",
26       main="Adj R Comparison")
27 points(summary(reg.fwd)$adjr2, type='b', pch=15, col=2)
28 legend("bottomright", c("Backward", "Forward"),
29       pch=c(20,15), col=c(1,2))
30
31 # Optimal models
32 best_bwd_idx <- which.max(summary(reg.bwd)$adjr2)
33 best_fwd_idx <- which.max(summary(reg.fwd)$adjr2)
34
35 # Compare coefficients in 4-variable models
36 coef(reg.fwd, id=4)
37 coef(reg.bwd, id=4)
```

Stepwise Selection with stats::step

```
1 # Stepwise selection using stats::step
2 # Start with full model
3 lm.full <- lm(Salary ~ ., data=data)
4
5 # Bidirectional stepwise selection (both)
6 set.seed(4061)
7 step.both <- step(lm.full, direction="both")
8
9 # Forward stepwise selection
10 set.seed(4061)
11 step.fwd <- step(lm.full, direction="forward")
12
13 # Backward stepwise selection
14 set.seed(4061)
15 step.bwd <- step(lm.full, direction="backward")
16
17 # Examine final model
18 summary(step.bwd)
19
20 # Get coefficients from optimal model
21 coef(step.bwd)
22
23 # Compare variable importance by coefficient magnitude
24 coefs <- abs(coef(step.bwd)[-1]) # Remove intercept
25 coefs_percent <- 100 * coefs / sum(coefs)
26 sort(coefs_percent, decreasing=TRUE)
27
28 # Make predictions
29 # Split data into train and test
30 set.seed(4061)
31 n <- nrow(data)
32 train_idx <- sample(1:n, 0.7*n)
33 train_data <- data[train_idx, ]
34 test_data <- data[-train_idx, ]
35
36 # Train model
37 lm.full <- lm(Salary ~ ., data=train_data)
38 step.model <- step(lm.full, direction="both")
39
40 # Predict and evaluate
41 predictions <- predict(step.model, newdata=test_data)
42 rmse <- sqrt(mean((predictions - test_data$Salary)^2))
43 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

```
1 library(glmnet)
2
3 # Load and prepare data
4 data <- na.omit(Hitters)
5 x <- model.matrix(Salary ~ ., data)[, -1] # Remove intercept
6 y <- data$Salary
7
8 # Split data
9 set.seed(4061)
10 n <- nrow(data)
11 train_idx <- sample(1:n, 0.7*n)
12 x_train <- x[train_idx, ]
13 y_train <- y[train_idx]
14 x_test <- x[-train_idx, ]
15 y_test <- y[-train_idx]
16
17 # Fit LASSO with cross-validation for optimal lambda
18 set.seed(4061)
19 lasso.cv <- cv.glmnet(x_train, y_train, alpha=1)
20 plot(lasso.cv)
21
22 # Get optimal lambda values
23 lambda_min <- lasso.cv$lambda.min # Minimum MSE
24 lambda_1se <- lasso.cv$lambda.1se # 1 Std. Error rule (more regularization)
25 print(c(lambda_min, lambda_1se))
26
27 # Fit LASSO with optimal lambda
28 lasso_min <- glmnet(x_train, y_train, alpha=1, lambda=lambda_min)
29 lasso_1se <- glmnet(x_train, y_train, alpha=1, lambda=lambda_1se)
30
31 # Selected variables
32 coef_min <- coef(lasso_min)
33 coef_1se <- coef(lasso_1se)
34
35 # Count non-zero coefficients
36 sum(coef_min[-1] != 0) # Exclude intercept
37 sum(coef_1se[-1] != 0) # Exclude intercept
38
39 # Print selected variables
40 selected_min <- which(coef_min[-1] != 0)
41 selected_1se <- which(coef_1se[-1] != 0)
42 print(colnames(x)[selected_min])
43 print(colnames(x)[selected_1se])
44
45 # Evaluate models
46 pred_min <- predict(lasso_min, x_test)
47 pred_1se <- predict(lasso_1se, x_test)
48 rmse_min <- sqrt(mean((pred_min - y_test)^2))
49 rmse_1se <- sqrt(mean((pred_1se - y_test)^2))
50 print(paste("RMSE with lambda_min:", round(rmse_min, 2)))
51 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

```
1 library(caret)
2
3 # Load and prepare data
4 data <- na.omit(Hitters)
5 x <- data[, -which(names(data) == "Salary")]
6 y <- data$Salary
7
8 # Split data
9 set.seed(4061)
10 train_idx <- createDataPartition(y, p=0.7, list=FALSE)
11 x_train <- x[train_idx, ]
12 y_train <- y[train_idx]
13 x_test <- x[-train_idx, ]
14 y_test <- y[-train_idx]
15
16 # Configure RFE
17 set.seed(4061)
18 # Define which feature subsets to evaluate
19 subsets <- c(1:5, 10, 15, ncol(x))
20
21 # RFE with linear regression
22 ctrl_lm <- rfeControl(functions=lmFuncs,
23                       method="cv",
24                       number=10,
25                       verbose=FALSE)
26
27 lm_rfe <- rfe(x_train, y_train,
28              sizes=subsets,
29              rfeControl=ctrl_lm)
30
31 # RFE with random forest
32 ctrl_rf <- rfeControl(functions=rfFuncs,
33                       method="cv",
34                       number=5,
35                       verbose=FALSE)
36
37 rf_rfe <- rfe(x_train, y_train,
38              sizes=subsets,
39              rfeControl=ctrl_rf)
40
41 # Print results
42 print(lm_rfe)
43 print(rf_rfe)
44
45 # Plot results
46 par(mfrow=c(1,2))
47 plot(lm_rfe, main="Linear Model RFE")
48 plot(rf_rfe, main="Random Forest RFE")
49
50 # Get selected variables
51 lm_vars <- predictors(lm_rfe)
52 rf_vars <- predictors(rf_rfe)
53 print(lm_vars)
54 print(rf_vars)
55
56 # Compare variable sets
57 common_vars <- intersect(lm_vars, rf_vars)
58 print(paste("Common variables:",
59             paste(common_vars, collapse=" ")))
60
61 # Make predictions using selected features
62 lm_pred <- predict(lm_rfe, x_test)
63 rf_pred <- predict(rf_rfe, x_test)
64 lm_rmse <- sqrt(mean((lm_pred - y_test)^2))
65 rf_rmse <- sqrt(mean((rf_pred - y_test)^2))
66 print(paste("Linear Model RFE RMSE:", round(lm_rmse, 2)))
67 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

```
1 # Demonstration of selection bias in feature selection
2 library(caret)
3 library(randomForest)
4
5 # Example of WRONG approach (selection bias)
6 biased_cv <- function(data, target) {
7   # Filter features using the entire dataset
8   my_filter <- function(x, y, cutoff=0.05) {
9     p <- ncol(x)
10    pvals <- numeric(p)
11    for(i in 1:p) {
12      pvals[i] <- wilcox.test(as.numeric(x[,i]) ~ y)$p.value
13    }
14    return(which(pvals < cutoff))
15  }
16
17  # Select features using all data
18  selected_features <- my_filter(data[, -target], data[, target], 0.2)
19  filtered_data <- data[, c(target, selected_features)]
20
21  # Cross-validate on the filtered data
22  cv_results <- train(
23    x = filtered_data[, -1],
24    y = filtered_data[, 1],
25    method = "rf",
26    trControl = trainControl(method="cv", number=5)
27  )
28
29  return(cv_results)
30 }
31
32 # CORRECT approach (avoid selection bias)
33 unbiased_cv <- function(data, target, B=10) {
34   n <- nrow(data)
35   acc <- numeric(B)
36
37   for(b in 1:B) {
38     # Create bootstrap sample
39     boot_idx <- sample(1:n, n, replace=TRUE)
40     train_data <- data[boot_idx, ]
41     test_idx <- setdiff(1:n, unique(boot_idx))
42     test_data <- data[test_idx, ]
43
44     # Filter features using ONLY training data
45     my_filter <- function(x, y, cutoff=0.05) {
46       p <- ncol(x)
47       pvals <- numeric(p)
48       for(i in 1:p) {
49         pvals[i] <- wilcox.test(as.numeric(x[,i]) ~ y)$p.value
50       }
51       return(which(pvals < cutoff))
52     }
53
54     # Select features using only training data
55     selected_features <- my_filter(
56       train_data[, -target],
57       train_data[, target],
58       0.2
59     )
60
61     # Only proceed if features were selected
62     if(length(selected_features) > 0) {
63       # Train model on filtered training data
64       rf_model <- randomForest(
65         x = train_data[, selected_features],
66         y = train_data[, target]
67       )
68
69       # Test on held-out data
70       predictions <- predict(
71         rf_model,
72         test_data[, selected_features]
73       )
74
75       # Calculate accuracy
76       if(is.factor(test_data[, target])) {
77         acc[b] <- mean(predictions == test_data[, target])
78       } else {
79         acc[b] <- sqrt(mean((predictions - test_data[, target])^2))
80       }
81     }
82   }
83 }
```

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

```
1 # Function to compare feature selection methods
2 compare_feature_selection <- function(data, target_col, k_folds=5) {
3   library(leaps)
4   library(glmnet)
5   library(caret)
6
7   # Extract data
8   x <- data[, -which(names(data) == target_col)]
9   y <- data[[target_col]]
10  n <- nrow(data)
11  p <- ncol(x)
12
13  # Prepare matrix format for glmnet
14  x_mat <- model.matrix(~ ., data=x)[, -1]
15
16  # Create folds
17  set.seed(4061)
18  folds <- createFolds(y, k=k_folds, returnTrain=TRUE)
19
20  # Initialize results storage
21  rmse_full <- rmse_step <- rmse_lasso <- rmse_rfe <- numeric(k_folds)
22  num_vars_step <- num_vars_lasso <- num_vars_rfe <- numeric(k_folds)
23
24  for(i in 1:k_folds) {
25    # Split data
26    train_idx <- folds[[i]]
27    test_idx <- setdiff(1:n, train_idx)
28
29    # Training and test sets
30    x_train <- x[train_idx, ]
31    y_train <- y[train_idx]
32    x_test <- x[test_idx, ]
33    y_test <- y[test_idx]
34
35    # Matrix format for train/test
36    x_mat_train <- x_mat[train_idx, ]
37    x_mat_test <- x_mat[test_idx, ]
38
39    # 1. Full model (baseline)
40    lm_full <- lm(y_train ~ ., data=x_train)
41    pred_full <- predict(lm_full, x_test)
42    rmse_full[i] <- sqrt(mean((pred_full - y_test)^2))
43
44    # 2. Stepwise selection
45    lm_step <- step(lm_full, direction="both", trace=0)
46    pred_step <- predict(lm_step, x_test)
47    rmse_step[i] <- sqrt(mean((pred_step - y_test)^2))
48    num_vars_step[i] <- length(coef(lm_step)) - 1 # Subtract intercept
49
50    # 3. LASSO
51    cv_lasso <- cv.glmnet(x_mat_train, y_train, alpha=1)
52    lambda_1se <- cv_lasso$lambda.1se
53    lasso_model <- glmnet(x_mat_train, y_train, alpha=1, lambda=lambda_1se)
54    pred_lasso <- predict(lasso_model, x_mat_test, s=lambda_1se)
55    rmse_lasso[i] <- sqrt(mean((pred_lasso - y_test)^2))
56    num_vars_lasso[i] <- sum(coef(lasso_model)[-1] != 0)
57
58    # 4. RFE with linear model
59    ctrl <- rfeControl(functions=lmFuncs, method="cv", number=5)
60    subsets <- c(1:min(5, p-1), p/2, p) # Adjust based on p
61    rfe_model <- rfe(x_train, y_train, sizes=subsets, rfeControl=ctrl)
62    pred_rfe <- predict(rfe_model, x_test)
63    rmse_rfe[i] <- sqrt(mean((pred_rfe - y_test)^2))
64    num_vars_rfe[i] <- length(predictors(rfe_model))
65  }
66
67  # Aggregate results
68  results <- data.frame(
69    Method = c("Full Model", "Stepwise", "LASSO", "RFE"),
70    Mean_RMSE = c(mean(rmse_full), mean(rmse_step),
71                  mean(rmse_lasso), mean(rmse_rfe)),
72    SD_RMSE = c(sd(rmse_full), sd(rmse_step),
73               sd(rmse_lasso), sd(rmse_rfe)),
74    Mean_Features = c(p, mean(num_vars_step),
75                     mean(num_vars_lasso), mean(num_vars_rfe))
76  )
77
78  return(results)
79 }
80
```

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 ($p < 40$), consider exhaustive or stepwise methods
- For large p ($p > 40$), 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

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

```
1 # Prepare the dataset
2 dat.nona = na.omit(airquality)
3 dat = airquality
4 dat.nona$Month = as.factor(dat.nona$Month)
5 dat.nona$Day = as.factor(dat.nona$Day)
6 dat$Month = as.factor(dat$Month)
7 dat$Day = as.factor(dat$Day)
8
9 # Association analysis
10 # 1. Correlation matrix
11 round(cor(na.omit(dat)),3)
12
13 # 2. Visual exploratory analysis
14 pairs(airquality)
15 boxplot(Wind~Month, data=airquality)
16 boxplot(Temp~Month, data=airquality)
17 boxplot(Ozone~Month, data=airquality)
18
19 # 3. Single correlation test
20 cor(airquality$Ozone, airquality$Temp, "pairwise")
21
22 # 4. QQ plot to compare distributions
23 qqplot(scale(airquality$Wind), scale(airquality$Temp))
24 abline(a=0, b=1)
25
26 # Extended EDA with DataExplorer package
27 library(DataExplorer)
28
29 # Overall dataset structure
30 introduce(dat)
31 plot_intro(dat)
32 plot_str(dat)
33 plot_missing(dat)
34
35 # Variable distributions
36 plot_bar(dat)
37 plot_bar(dat, with="Ozone")
38 plot_boxplot(dat, by="Solar.R")
39 plot_histogram(dat)
40 plot_qq(dat)
41 plot_qq(dat, by="Ozone")
42
43 # Transform variables for normality if needed
44 dat2 = dat
45 dat2$Solar.R = sqrt(dat2$Solar.R+1)
46 plot_qq(dat2)
47
48 # Correlation analysis
49 plot_correlation(dat[,1:4])
50 cor(na.omit(dat[,1:4]))
51 plot_correlation(na.omit(dat[,1:4]))
52 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

```
1 # 1. Simple imputation with median
2 is = which(is.na(dat$Solar.R))
3 dat$Solar.R[is] = median(dat$Solar.R[-is])
4
5 # Evaluate imputation with a GLM
6 glm1.fit = glm(Wind~., data=dat)
7 cat("RSS after median imputation:", sum((glm1.fit$residuals)^2), "\n")
8
9 # 2. Model-based imputation using quadratic regression
10 is = which(is.na(dat$Ozone))
11 qmo = lm(Ozone~Temp+I(Temp^2), data=na.omit(airquality))
12 Ozq = predict(qmo, newdata=airquality[is,])
13 dat$Ozone[is] = Ozq
14
15 # Evaluate imputation with a GLM
16 glm3.fit = glm(Wind~., data=dat)
17 cat("RSS after model-based imputation:", sum((glm3.fit$residuals)^2), "\n")
18
19 # Visualize the imputation
20 plot(dat$Ozone, dat$Temp)
21 points(airquality$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

```
1 library(tree)
2
3 # 1. Fit an unpruned regression tree
4 tree.fit = tree(Wind~., data=dat)
5 summary(tree.fit)
6
7 # Identify variables actually used in the tree
8 summary(tree.fit)$used
9
10 # Visualize the tree
11 plot(tree.fit)
12 text(tree.fit)
13
14 # Calculate RSS for the unpruned tree
15 rss_unpruned = sum(summary(tree.fit)$residuals^2)
16 cat("RSS for unpruned tree:", rss_unpruned, "\n")
17
18 # 2. Prune the tree to a specific size
19 pruned.fit = prune.tree(tree.fit, best=10)
20 summary(pruned.fit)
21
22 # Identify variables used in the pruned tree
23 summary(pruned.fit)$used
24
25 # Calculate RSS for the pruned tree
26 rss_pruned = sum(summary(pruned.fit)$residuals^2)
27 cat("RSS for pruned tree:", rss_pruned, "\n")
28
29 # Compare RSS values
30 cat("RSS difference (pruned - unpruned):", rss_pruned - rss_unpruned, "\n")
31 cat("Percentage increase in RSS:",
32     100 * (rss_pruned - rss_unpruned) / rss_unpruned, "%\n")
33
34 # The RSS typically increases slightly when pruning as we trade a small
35 # 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

```
1 library(ISLR) # For the Khan dataset
2 library(randomForest)
3 library(gbm)
4
5 # Set up data
6 xtrain = Khan$xtrain
7 xtest = Khan$xtest
8 ytrain = as.factor(Khan$ytrain)
9 ytest = as.factor(Khan$ytest)
10
11 # Check class distributions
12 round(prop.table(table(ytrain)), 3)
13 round(prop.table(table(ytest)), 3)
14 round(table(ytrain)/length(ytrain), 3) * 100
15 round(table(ytest)/length(ytest), 3) * 100
16
17 # Fit random forest
18 set.seed(4061)
19 rfo = randomForest(xtrain, ytrain)
20 print(rfo)
21
22 # Generate predictions
23 rfpr = predict(rfo, xtest, type="prob") # Probability predictions
24 rfp = predict(rfo, xtest)              # Class predictions
25
26 # Evaluate accuracy
27 conf_matrix = table(rfp, ytest)
28 accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)
29 cat("Random Forest Test Accuracy:", accuracy, "\n")
30 print(conf_matrix)
31
32 # Variable importance
33 is = which(rfo$importance > 0.4)
34 important_vars = rownames(rfo$importance)[is]
35 cat("Important variables (importance > 0.4):\n")
36 print(important_vars)
37
38 # Optional: Visualize variable importance
39 # barplot(rfo$importance[is,], las=2)
```

Gradient Boosting Model

```
1 # Fit gradient boosting model for multinomial classification
2 gb.out = gbm(ytrain~., data=as.data.frame(xtrain),
3             distribution='multinomial')
4
5 # Generate predictions
6 gb.fitted = predict(gb.out, n.trees=gb.out$n.trees)
7 gb.pred = predict(gb.out, as.data.frame(xtest),
8                 n.trees=gb.out$n.trees)
9
10 # Convert predictions to class labels
11 gbp = apply(gb.pred, 1, which.max)
12
13 # Evaluate accuracy
14 gb_conf_matrix = table(ytest, gbp)
15 gb_accuracy = sum(diag(gb_conf_matrix)) / sum(gb_conf_matrix)
16 cat("Gradient Boosting Test Accuracy:", gb_accuracy, "\n")
17 print(gb_conf_matrix)
18
19 # Compare model performances
20 cat("Model comparison:\n")
21 cat("Random Forest accuracy:", accuracy, "\n")
22 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