Spam Email Classification using Supervised Learning

IAI - Artificial Intelligence | Assignment 2 | 2025/2026

Project Overview

This notebook implements a **spam email classification system** using supervised learning techniques. We compare four different algorithms on the UCI Spambase dataset:

1. **CART** (Classification and Regression Trees) - Decision Tree with binary splits
2. **k-Nearest Neighbors (k-NN)** - Instance-based learning
3. **Random Forest** - Ensemble of Decision Trees
4. **Oblique Decision Tree** - Linear combination splits (custom implementation)

Dataset: Spambase

- **Source:** UCI Machine Learning Repository
- **Instances:** 4,601 emails
- **Features:** 57 continuous attributes
- **Target:** Binary classification (1 = spam, 0 = ham)
- **Class Distribution:** 39.4% spam, 60.6% ham

Notebook Structure

- **Part 1:** Data Loading & Exploratory Data Analysis (EDA)
- **Part 2:** Data Preprocessing
- **Part 3:** Model Implementation & Training
- **Part 4:** Evaluation & Comparison
- **Part 5:** Advanced Analysis (Learning Curves, ROC Curves)

In [5]:
```
# ============================================================================
# IMPORTS
# ============================================================================

import warnings
import time
from typing import Dict, Any, List, Tuple

# Data manipulation
import numpy as np
import pandas as pd
```

```
# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn
from sklearn.model_selection import (
    train_test_split,
    cross_val_score,
    StratifiedKFold,
    learning_curve,
    GridSearchCV
)
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
    ConfusionMatrixDisplay,
    roc_curve,
    roc_auc_score,
    auc
)

# Suppress warnings for clean output
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

# Plot style
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette('husl')

print("✓ All imports successful!")
```

```
✓ All imports successful!
```

PART 1: Data Loading & Exploratory Data Analysis

1.1 Load the Spambase Dataset

In [6]:
```
# ================================================================================
# LOAD DATASET - Option 1: Using UCI ML Repository API
# ================================================================================

from ucimlrepo import fetch_ucirepo

# Fetch dataset
spambase = fetch_ucirepo(id=94)

# Extract features and target
X = spambase.data.features
y = spambase.data.targets.values.ravel()  # Convert to 1D array

# Get feature names
feature_names = X.columns.tolist()

print(f"Dataset loaded successfully!")
print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"\nFeature names ({len(feature_names)}):")
print(feature_names)
```

```
Dataset loaded successfully!
Features shape: (4601, 57)
Target shape: (4601,)

Feature names (57):
['word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d', 'word_freq_our', 'word_freq_over', 'word_freq_remove', 'word_freq_internet', 'word_freq_orde
r', 'word_freq_mail', 'word_freq_receive', 'word_freq_will', 'word_freq_people', 'word_freq_report', 'word_freq_addresses', 'word_freq_free', 'word_freq_business', 'wo
rd_freq_email', 'word_freq_you', 'word_freq_credit', 'word_freq_your', 'word_freq_font', 'word_freq_000', 'word_freq_money', 'word_freq_hp', 'word_freq_hpl', 'word_fre
q_george', 'word_freq_650', 'word_freq_lab', 'word_freq_labs', 'word_freq_telnet', 'word_freq_857', 'word_freq_data', 'word_freq_415', 'word_freq_85', 'word_freq_techn
ology', 'word_freq_1999', 'word_freq_parts', 'word_freq_pm', 'word_freq_direct', 'word_freq_cs', 'word_freq_meeting', 'word_freq_original', 'word_freq_project', 'word_
freq_re', 'word_freq_edu', 'word_freq_table', 'word_freq_conference', 'char_freq_;', 'char_freq_(', 'char_freq_[', 'char_freq_!', 'char_freq_$', 'char_freq_#', 'capita
l_run_length_average', 'capital_run_length_longest', 'capital_run_length_total']
```

In [7]:
```
# ================================================================================
# ALTERNATIVE: Load from local CSV file (if API doesn't work)
# ================================================================================

# Uncomment the code below if you prefer to load from the .data file

# # Define column names based on spambase.names
# column_names = [
#     'word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d',
#     'word_freq_our', 'word_freq_over', 'word_freq_remove', 'word_freq_internet',
#     'word_freq_order', 'word_freq_mail', 'word_freq_receive', 'word_freq_will',
#     'word_freq_people', 'word_freq_report', 'word_freq_addresses', 'word_freq_free',
#     'word_freq_business', 'word_freq_email', 'word_freq_you', 'word_freq_credit',
#     'word_freq_your', 'word_freq_font', 'word_freq_000', 'word_freq_money',
#     'word_freq_hp', 'word_freq_hpl', 'word_freq_george', 'word_freq_650',
```

```
#     'word_freq_lab', 'word_freq_labs', 'word_freq_telnet', 'word_freq_857',
#     'word_freq_data', 'word_freq_415', 'word_freq_85', 'word_freq_technology',
#     'word_freq_1999', 'word_freq_parts', 'word_freq_pm', 'word_freq_direct',
#     'word_freq_cs', 'word_freq_meeting', 'word_freq_original', 'word_freq_project',
#     'word_freq_re', 'word_freq_edu', 'word_freq_table', 'word_freq_conference',
#     'char_freq_;', 'char_freq_(', 'char_freq_[', 'char_freq_!', 'char_freq_$', 'char_freq_#',
#     'capital_run_length_average', 'capital_run_length_longest', 'capital_run_length_total',
#     'spam'
# ]

# # Load data
# df = pd.read_csv('spambase.data', header=None, names=column_names)
# X = df.drop('spam', axis=1)
# y = df['spam'].values
# feature_names = X.columns.tolist()
```

1.2 Basic Dataset Statistics

In [8]:
```
# ============================================================================
# DATASET OVERVIEW
# ============================================================================

# Create DataFrame for analysis
df = X.copy()
df['spam'] = y

print("=" * 60)
print("DATASET OVERVIEW")
print("=" * 60)
print(f"\nTotal samples: {len(df)}")
print(f"Total features: {len(feature_names)}")
print(f"\nClass distribution:")
print(f"  - Ham (0):  {(y == 0).sum()} ({(y == 0).mean()*100:.1f}%)")
print(f"  - Spam (1): {(y == 1).sum()} ({(y == 1).mean()*100:.1f}%)")
print(f"\nMissing values: {df.isnull().sum().sum()}")
print(f"\nData types:")
print(df.dtypes.value_counts())
```

```
============================================================
DATASET OVERVIEW
============================================================

Total samples: 4601
Total features: 57

Class distribution:
  - Ham (0):  2788 (60.6%)
  - Spam (1): 1813 (39.4%)

Missing values: 0

Data types:
float64    55
int64       3
Name: count, dtype: int64
```

In [9]:
```
# Statistical summary
print("\nStatistical Summary (first 10 features):")
df.iloc[:, :10].describe().round(3)
```

```
Statistical Summary (first 10 features):
```

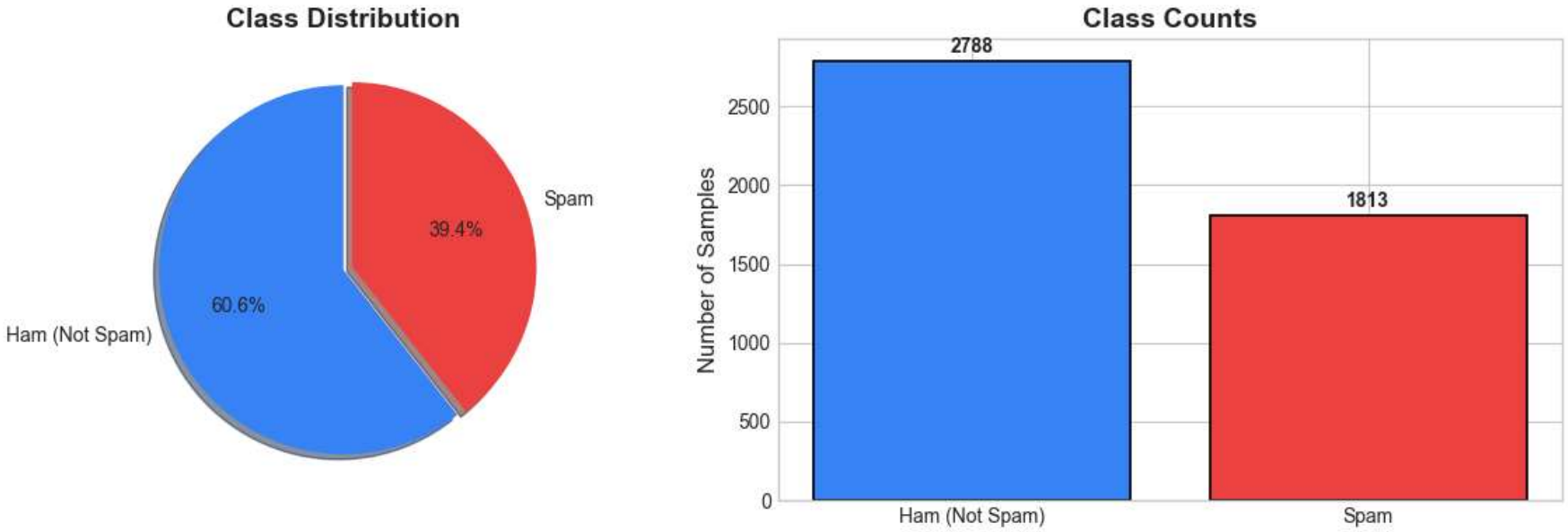
Out[9]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail
count	4601.000	4601.000	4601.000	4601.000	4601.000	4601.000	4601.000	4601.000	4601.000	4601.000
mean	0.105	0.213	0.281	0.065	0.312	0.096	0.114	0.105	0.090	0.239
std	0.305	1.291	0.504	1.395	0.673	0.274	0.391	0.401	0.279	0.645
min	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
25%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
50%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
75%	0.000	0.000	0.420	0.000	0.380	0.000	0.000	0.000	0.000	0.160
max	4.540	14.280	5.100	42.810	10.000	5.880	7.270	11.110	5.260	18.180

1.3 Visualizations

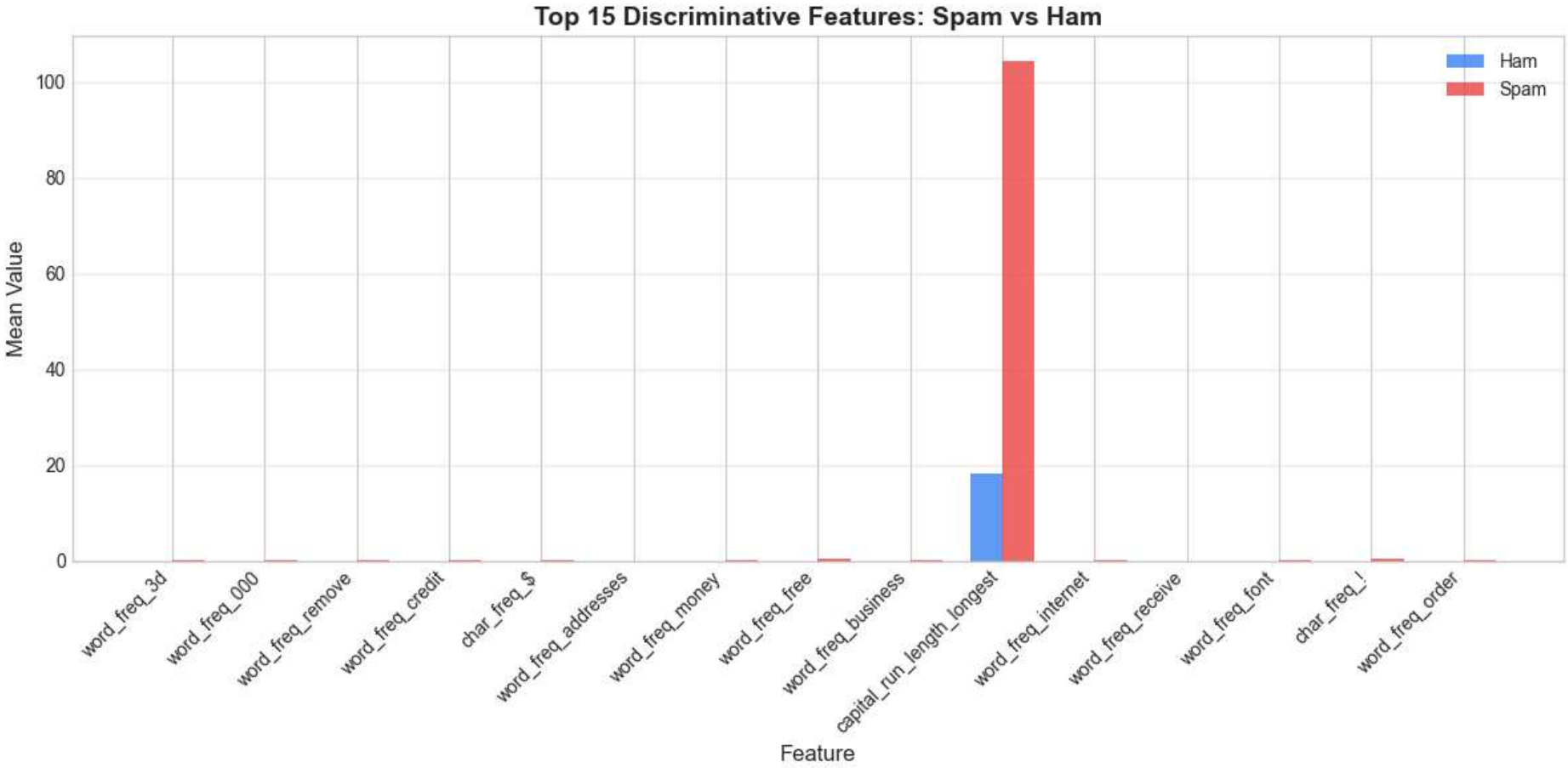
In [10]:
```
# ============================================================================
# CLASS DISTRIBUTION
# ============================================================================

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Pie chart
colors = ['#3B82F6', '#EF4444']
labels = ['Ham (Not Spam)', 'Spam']
```

```
sizes = [(y == 0).sum(), (y == 1).sum()]

axes[0].pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90,
            explode=(0, 0.05), shadow=True)
axes[0].set_title('Class Distribution', fontsize=14, fontweight='bold')

# Bar chart
axes[1].bar(labels, sizes, color=colors, edgecolor='black', linewidth=1.2)
axes[1].set_ylabel('Number of Samples', fontsize=12)
axes[1].set_title('Class Counts', fontsize=14, fontweight='bold')
for i, v in enumerate(sizes):
    axes[1].text(i, v + 50, str(v), ha='center', fontweight='bold')

plt.tight_layout()
plt.show()
```



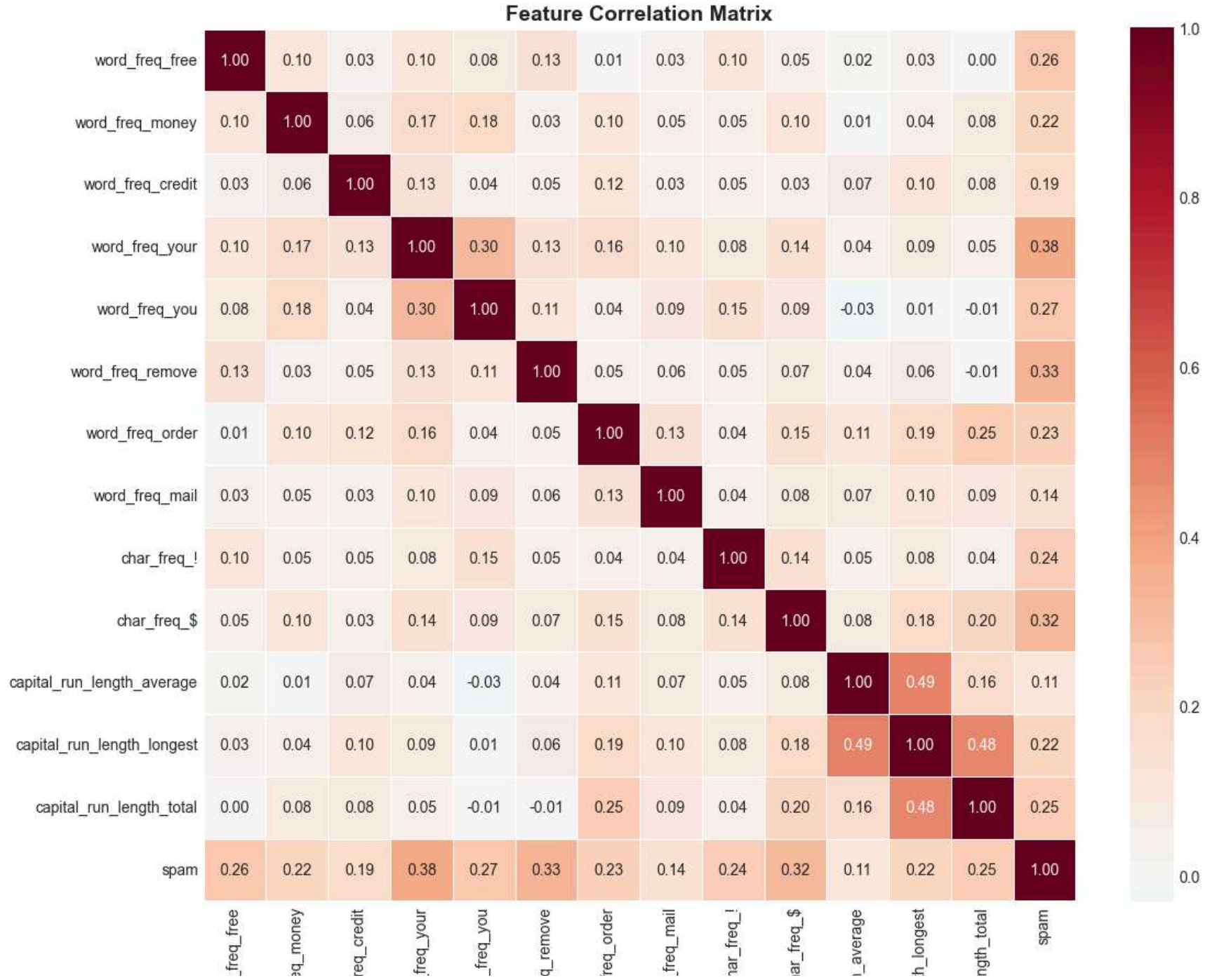
In [11]:
```
# ============================================================================
# TOP DISCRIMINATIVE FEATURES (Spam vs Ham)
# ============================================================================

# Calculate mean values for spam vs ham
spam_means = df[df['spam'] == 1].drop('spam', axis=1).mean()
ham_means = df[df['spam'] == 0].drop('spam', axis=1).mean()

# Calculate difference ratio
diff_ratio = (spam_means - ham_means) / (ham_means + 0.001)
top_features = diff_ratio.abs().sort_values(ascending=False).head(15)
```

```python
# Plot
fig, ax = plt.subplots(figsize=(12, 6))

x = np.arange(len(top_features))
width = 0.35

bars1 = ax.bar(x - width/2, ham_means[top_features.index], width, label='Ham', color='#3B82F6', alpha=0.8)
bars2 = ax.bar(x + width/2, spam_means[top_features.index], width, label='Spam', color='#EF4444', alpha=0.8)

ax.set_xlabel('Feature', fontsize=12)
ax.set_ylabel('Mean Value', fontsize=12)
ax.set_title('Top 15 Discriminative Features: Spam vs Ham', fontsize=14, fontweight='bold')
ax.set_xticks(x)
ax.set_xticklabels(top_features.index, rotation=45, ha='right')
ax.legend()
ax.grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()
```



In [12]:
```
# ============================================================================
# FEATURE CORRELATION HEATMAP
# ============================================================================

# Select subset of features for readability
selected_features = [
    'word_freq_free', 'word_freq_money', 'word_freq_credit', 'word_freq_your',
    'word_freq_you', 'word_freq_remove', 'word_freq_order', 'word_freq_mail',
    'char_freq_!', 'char_freq_$', 'capital_run_length_average',
    'capital_run_length_longest', 'capital_run_length_total'
]

# Check which features exist in our dataset
existing_features = [f for f in selected_features if f in df.columns]
existing_features.append('spam')
```

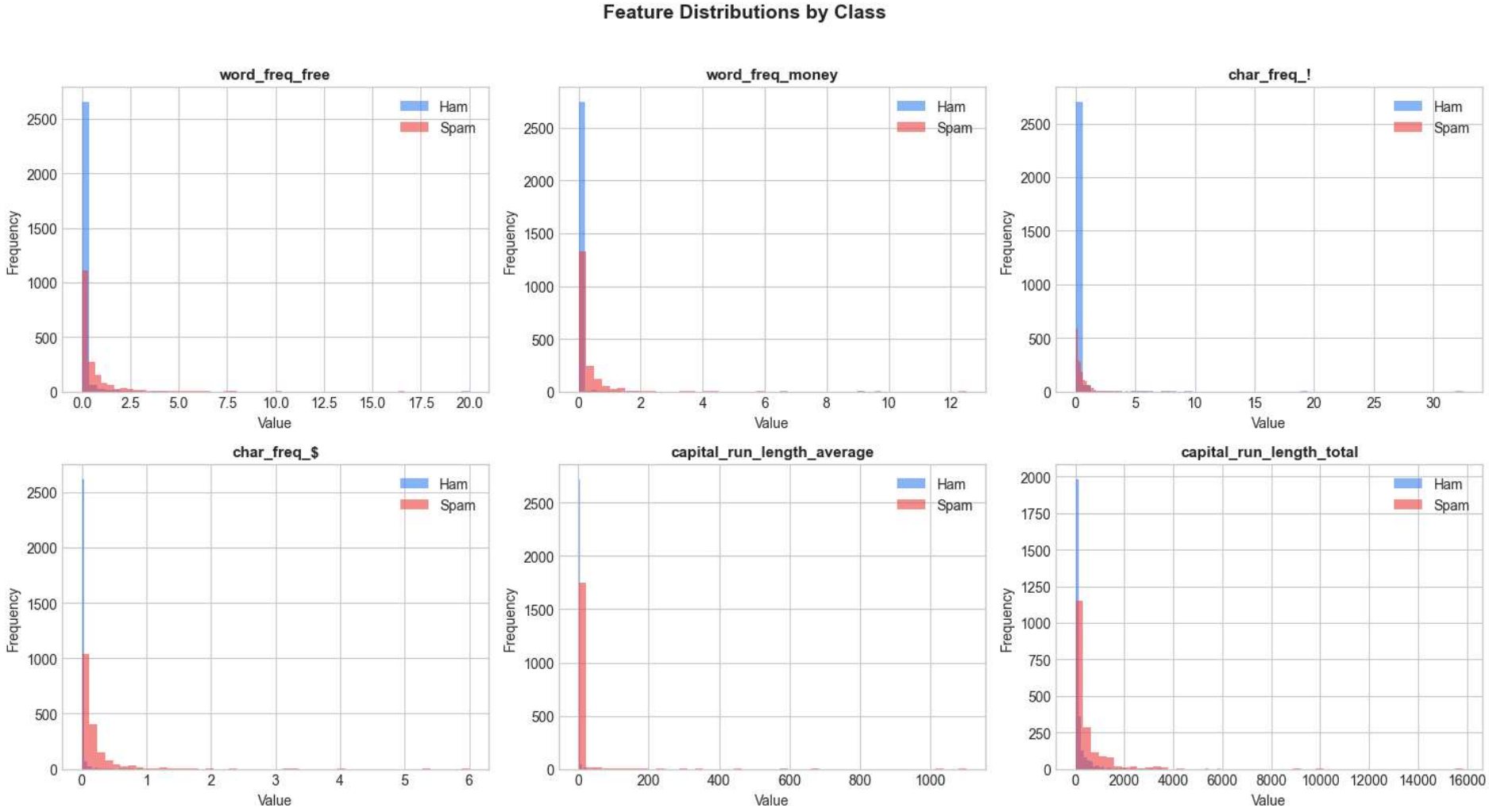
```
corr_matrix = df[existing_features].corr()

plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, cmap='RdBu_r', center=0,
            fmt='.2f', square=True, linewidths=0.5)
plt.title('Feature Correlation Matrix', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```


Feature Correlation Matrix
word_freq_free 1.00 0.10 0.03 0.10 0.08 0.13 0.01 0.03 0.10 0.05 0.02 0.03 0.00 0.26
word_freq_money 0.10 1.00 0.06 0.17 0.18 0.03 0.10 0.05 0.05 0.10 0.01 0.04 0.08 0.22
word_freq_credit 0.03 0.06 1.00 0.13 0.04 0.05 0.12 0.03 0.05 0.03 0.07 0.10 0.08 0.19
word_freq_your 0.10 0.17 0.13 1.00 0.30 0.13 0.16 0.10 0.08 0.14 0.04 0.09 0.05 0.38
word_freq_you 0.08 0.18 0.04 0.30 1.00 0.11 0.04 0.09 0.15 0.09 -0.03 0.01 -0.01 0.27
word_freq_remove 0.13 0.03 0.05 0.13 0.11 1.00 0.05 0.06 0.05 0.07 0.04 0.06 -0.01 0.33
word_freq_order 0.01 0.10 0.12 0.16 0.04 0.05 1.00 0.13 0.04 0.15 0.11 0.19 0.25 0.23
word_freq_mail 0.03 0.05 0.03 0.10 0.09 0.06 0.13 1.00 0.04 0.08 0.07 0.10 0.09 0.14
char_freq_! 0.10 0.05 0.05 0.08 0.15 0.05 0.04 0.04 1.00 0.14 0.05 0.08 0.04 0.24
char_freq_$ 0.05 0.10 0.03 0.14 0.09 0.07 0.15 0.08 0.14 1.00 0.08 0.18 0.20 0.32
capital_run_length_average 0.02 0.01 0.07 0.04 -0.03 0.04 0.11 0.07 0.05 0.08 1.00 0.49 0.16 0.11
capital_run_length_longest 0.03 0.04 0.10 0.09 0.01 0.06 0.19 0.10 0.08 0.18 0.49 1.00 0.48 0.22
capital_run_length_total 0.00 0.08 0.08 0.05 -0.01 -0.01 0.25 0.09 0.04 0.20 0.16 0.48 1.00 0.25
spam 0.26 0.22 0.19 0.38 0.27 0.33 0.23 0.14 0.24 0.32 0.11 0.22 0.25 1.00
_freq_free
eq_money
req_credit
freq_your
_freq_you
q_remove
req_order
_freq_mail
har_freq_!
iar_freq_$
l_average
h_longest
ngth_total
spam
1.0
0.8
0.6
0.4
0.2
0.0

word_ word_fre word_fi word_ word_ word_freq word_f word_ ch ch capital_run_length capital_run_lengt capital_run_le

In [13]:

```
# ================================================================================
# FEATURE DISTRIBUTIONS
# ================================================================================

# Select key features to visualize
key_features = ['word_freq_free', 'word_freq_money', 'char_freq_!', 'char_freq_$',
                'capital_run_length_average', 'capital_run_length_total']

# Filter to existing features
key_features = [f for f in key_features if f in df.columns][:6]

fig, axes = plt.subplots(2, 3, figsize=(15, 8))
axes = axes.flatten()

for idx, feature in enumerate(key_features):
    ax = axes[idx]

    # Plot distributions for spam and ham
    df[df['spam'] == 0][feature].hist(ax=ax, bins=50, alpha=0.6, label='Ham', color='#3B82F6')
    df[df['spam'] == 1][feature].hist(ax=ax, bins=50, alpha=0.6, label='Spam', color='#EF4444')

    ax.set_title(feature, fontsize=11, fontweight='bold')
    ax.legend()
    ax.set_xlabel('Value')
    ax.set_ylabel('Frequency')

plt.suptitle('Feature Distributions by Class', fontsize=14, fontweight='bold', y=1.02)
plt.tight_layout()
plt.show()
```



PART 2: Data Preprocessing

2.1 Train-Test Split

In [14]:
```
# ============================================================================
# TRAIN-TEST SPLIT
# ============================================================================
```

```
# Split data: 80% training, 20% testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_STATE, stratify=y
)

print("Train-Test Split:")
print(f"  Training set: {X_train.shape[0]} samples ({X_train.shape[0]/len(X)*100:.0f}%)")
print(f"  Test set:     {X_test.shape[0]} samples ({X_test.shape[0]/len(X)*100:.0f}%)")
print(f"\nClass distribution in training set:")
print(f"  Ham:  {(y_train == 0).sum()} ({(y_train == 0).mean()*100:.1f}%)")
print(f"  Spam: {(y_train == 1).sum()} ({(y_train == 1).mean()*100:.1f}%)")
```

```
Train-Test Split:
  Training set: 3680 samples (80%)
  Test set:     921 samples (20%)

Class distribution in training set:
  Ham:  2230 (60.6%)
  Spam: 1450 (39.4%)
```

2.2 Feature Scaling

Some algorithms (like k-NN) are sensitive to feature scales. We'll use StandardScaler to normalize the features.

In [15]:
```
# ============================================================================
# FEATURE SCALING
# ============================================================================

# StandardScaler for algorithms sensitive to scale (k-NN, Oblique Tree)
scaler = StandardScaler()

# Fit on training data, transform both
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert back to DataFrame for convenience
X_train_scaled = pd.DataFrame(X_train_scaled, columns=feature_names)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=feature_names)

print("Feature scaling applied (StandardScaler)")
print(f"\nScaled training data - Mean: {X_train_scaled.mean().mean():.6f}, Std: {X_train_scaled.std().mean():.6f}")
```

```
Feature scaling applied (StandardScaler)

Scaled training data - Mean: 0.000000, Std: 1.000136
```

PART 3: Model Implementation & Training

3.1 Oblique Decision Tree Implementation

Custom implementation using ISTA (Iterative Shrinkage-Thresholding Algorithm) for sparse linear splits.

In [16]:
```python
# ============================================================================
# OBLIQUE DECISION TREE - HELPER FUNCTIONS
# ============================================================================

def sigmoid(z):
    """Numerically stable sigmoid function."""
    z = np.clip(z, -20, 20)
    return 1.0 / (1.0 + np.exp(-z))

def soft_threshold(x, lambda_):
    """Proximal operator for L1 norm (soft thresholding)."""
    return np.sign(x) * np.maximum(np.abs(x) - lambda_, 0.0)

def compute_loss_and_gradients(X, y, weights, bias):
    """Compute logistic loss and gradients."""
    n_samples = X.shape[0]
    linear_scores = X @ weights + bias
    probs = sigmoid(linear_scores)

    epsilon = 1e-12
    probs_clipped = np.clip(probs, epsilon, 1 - epsilon)
    loss = -np.sum(y * np.log(probs_clipped) + (1 - y) * np.log(1 - probs_clipped)) / n_samples

    error = (probs - y) / n_samples
    grad_weights = X.T @ error
    grad_bias = np.sum(error)

    return loss, grad_weights, grad_bias

def train_logistic_ista(X, y, l1_lambda=0.01, max_iter=100, lr=0.1, tol=1e-5):
    """Train logistic regression with L1 penalty using ISTA."""
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0.0

    for _ in range(max_iter):
        _, grad_w, grad_b = compute_loss_and_gradients(X, y, weights, bias)

        # Gradient step
        weights_temp = weights - lr * grad_w
        bias_new = bias - lr * grad_b

        # Proximal step (soft thresholding for L1)
        weights_new = soft_threshold(weights_temp, lr * l1_lambda) if l1_lambda > 0 else weights_temp

        # Check convergence
        if np.linalg.norm(weights_new - weights) < tol and abs(bias_new - bias) < tol:
```

```
            break

        weights, bias = weights_new, bias_new

    return weights, bias

print("✓ Oblique Tree helper functions defined")
```

```
✓ Oblique Tree helper functions defined
```

In [17]:
```
# ============================================================================
# OBLIQUE DECISION TREE - NODE CLASS
# ============================================================================

class ObliqueNode:
    """Node for Oblique Decision Tree."""
    def __init__(self):
        self.is_leaf = True
        self.weights = None
        self.bias = None
        self.left_child = None
        self.right_child = None
        self.class_probs = None
        self.predicted_class = None
        self.n_samples = 0

print("✓ ObliqueNode class defined")
```

```
✓ ObliqueNode class defined
```

In [18]:
```
# ============================================================================
# OBLIQUE DECISION TREE - MAIN CLASS
# ============================================================================

class ObliqueDecisionTree(BaseEstimator, ClassifierMixin):
    """
    Oblique Decision Tree with linear (non-axis-aligned) splits.
    Uses L1 regularization for sparse, interpretable splits.
    """

    def __init__(self, max_depth=5, min_samples_split=10, min_samples_leaf=5,
                 l1_regularization=0.01, random_state=None):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.l1_regularization = l1_regularization
        self.random_state = random_state
        self.root = None
        self.n_classes_ = None
        self.classes_ = None
        self.n_features_ = None

    def _gini_impurity(self, y):
```

```
        """Calculate Gini impurity."""
        if len(y) == 0:
            return 0.0
        probs = np.bincount(y, minlength=self.n_classes_) / len(y)
        return 1.0 - np.sum(probs ** 2)

    def _find_best_split(self, X, y):
        """Find best oblique split using logistic regression."""
        n_samples = X.shape[0]

        if n_samples < self.min_samples_split:
            return None, None

        # Use fast logistic regression for initial split direction
        try:
            lr = LogisticRegression(
                penalty='l1', solver='liblinear', C=1.0/(self.l1_regularization + 1e-6),
                random_state=self.random_state, max_iter=100
            )
            lr.fit(X, y)
            weights = lr.coef_.ravel()
            bias = lr.intercept_[0]
        except:
            return None, None

        # Refine with ISTA for sparsity
        weights, bias = train_logistic_ista(X, y, l1_lambda=self.l1_regularization)

        # Check if split is valid
        scores = X @ weights + bias
        left_mask = scores < 0
        right_mask = ~left_mask

        if left_mask.sum() < self.min_samples_leaf or right_mask.sum() < self.min_samples_leaf:
            return None, None

        return weights, bias

    def _build_tree(self, X, y, depth=0):
        """Recursively build the tree."""
        node = ObliqueNode()
        node.n_samples = len(y)

        # Calculate class probabilities
        class_counts = np.bincount(y, minlength=self.n_classes_)
        node.class_probs = class_counts / len(y)
        node.predicted_class = np.argmax(class_counts)

        # Check stopping conditions
        if (depth >= self.max_depth or
            len(y) < self.min_samples_split or
            len(np.unique(y)) == 1):
```

```
            return node

        # Find best split
        weights, bias = self._find_best_split(X, y)

        if weights is None:
            return node

        # Create split
        node.is_leaf = False
        node.weights = weights
        node.bias = bias

        scores = X @ weights + bias
        left_mask = scores < 0
        right_mask = ~left_mask

        # Recursively build children
        node.left_child = self._build_tree(X[left_mask], y[left_mask], depth + 1)
        node.right_child = self._build_tree(X[right_mask], y[right_mask], depth + 1)

        return node

    def fit(self, X, y):
        """Fit the oblique decision tree."""
        X = np.asarray(X)
        y = np.asarray(y)

        self.classes_ = np.unique(y)
        self.n_classes_ = len(self.classes_)
        self.n_features_ = X.shape[1]

        if self.random_state is not None:
            np.random.seed(self.random_state)

        self.root = self._build_tree(X, y)
        return self

    def _predict_single(self, x, node):
        """Predict class for a single sample."""
        if node.is_leaf:
            return node.class_probs

        score = np.dot(x, node.weights) + node.bias
        if score < 0:
            return self._predict_single(x, node.left_child)
        else:
            return self._predict_single(x, node.right_child)

    def predict_proba(self, X):
        """Predict class probabilities."""
        X = np.asarray(X)
```

```
        return np.array([self._predict_single(x, self.root) for x in X])

    def predict(self, X):
        """Predict class labels."""
        proba = self.predict_proba(X)
        return self.classes_[np.argmax(proba, axis=1)]

print("✓ ObliqueDecisionTree class defined")
```

```
✓ ObliqueDecisionTree class defined
```

3.2 Define All Models

In [19]:

```
# ============================================================================
# MODEL DEFINITIONS
# ============================================================================

models = {
    'CART': {
        'model': DecisionTreeClassifier(
            criterion='gini',
            max_depth=10,
            min_samples_split=10,
            min_samples_leaf=5,
            random_state=RANDOM_STATE
        ),
        'use_scaled': False,  # CART doesn't need scaling
        'color': '#3B82F6'
    },
    'k-NN': {
        'model': KNeighborsClassifier(
            n_neighbors=5,
            weights='distance',
            metric='minkowski',
            p=2
        ),
        'use_scaled': True,  # k-NN needs scaling
        'color': '#10B981'
    },
    'Random Forest': {
        'model': RandomForestClassifier(
            n_estimators=100,
            criterion='gini',
            max_depth=15,
            min_samples_split=5,
            min_samples_leaf=2,
            random_state=RANDOM_STATE,
            n_jobs=-1
        ),
        'use_scaled': False,  # Random Forest doesn't need scaling
        'color': '#F59E0B'
    },
```

```
    'Oblique Tree': {
        'model': ObliqueDecisionTree(
            max_depth=6,
            min_samples_split=20,
            min_samples_leaf=10,
            l1_regularization=0.05,
            random_state=RANDOM_STATE
        ),
        'use_scaled': True,  # Oblique Tree benefits from scaling
        'color': '#EF4444'
    }
}

print("Models defined:")
for name in models:
    print(f"  ✓ {name}")
```

```
Models defined:
  ✓ CART
  ✓ k-NN
  ✓ Random Forest
  ✓ Oblique Tree
```

3.3 Train Models

In [20]:

```
# ============================================================================
# TRAIN ALL MODELS
# ============================================================================

results = {}

print("Training models...\n")
print("=" * 60)

for name, config in models.items():
    print(f"\nTraining {name}...")

    # Select appropriate data (scaled or not)
    if config['use_scaled']:
        X_tr, X_te = X_train_scaled.values, X_test_scaled.values
    else:
        X_tr, X_te = X_train.values, X_test.values

    # Train model and measure time
    start_time = time.time()
    model = config['model']
    model.fit(X_tr, y_train)
    train_time = time.time() - start_time

    # Predict and measure time
    start_time = time.time()
    y_pred = model.predict(X_te)
```

```
    y_proba = model.predict_proba(X_te)[:, 1] if hasattr(model, 'predict_proba') else None
    test_time = time.time() - start_time

    # Calculate metrics
    results[name] = {
        'model': model,
        'y_pred': y_pred,
        'y_proba': y_proba,
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, y_proba) if y_proba is not None else None,
        'train_time': train_time,
        'test_time': test_time,
        'color': config['color']
    }

    print(f"  ✓ Accuracy: {results[name]['accuracy']:.4f}")
    print(f"  ✓ Training time: {train_time:.4f}s")

print("\n" + "=" * 60)
print("All models trained successfully!")
```

```
Training models...

============================================================

Training CART...
  ✓ Accuracy: 0.9088
  ✓ Training time: 0.0397s

Training k-NN...
  ✓ Accuracy: 0.9110
  ✓ Training time: 0.0025s

Training Random Forest...
  ✓ Accuracy: 0.9359
  ✓ Training time: 0.2981s

Training Oblique Tree...
  ✓ Accuracy: 0.8328
  ✓ Training time: 1.9161s

============================================================
All models trained successfully!
```

PART 4: Evaluation & Comparison

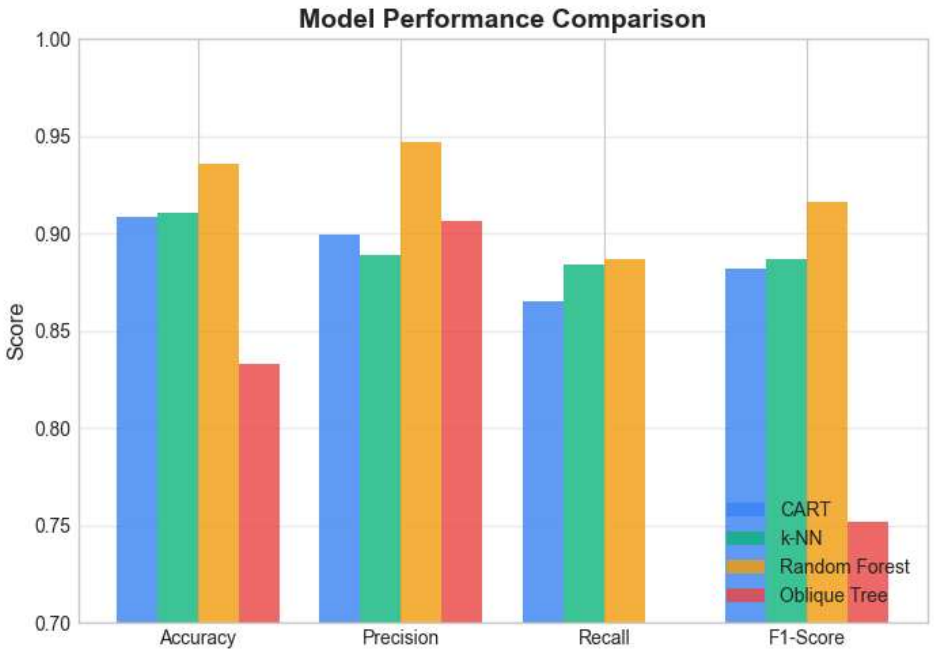
4.1 Metrics Summary Table

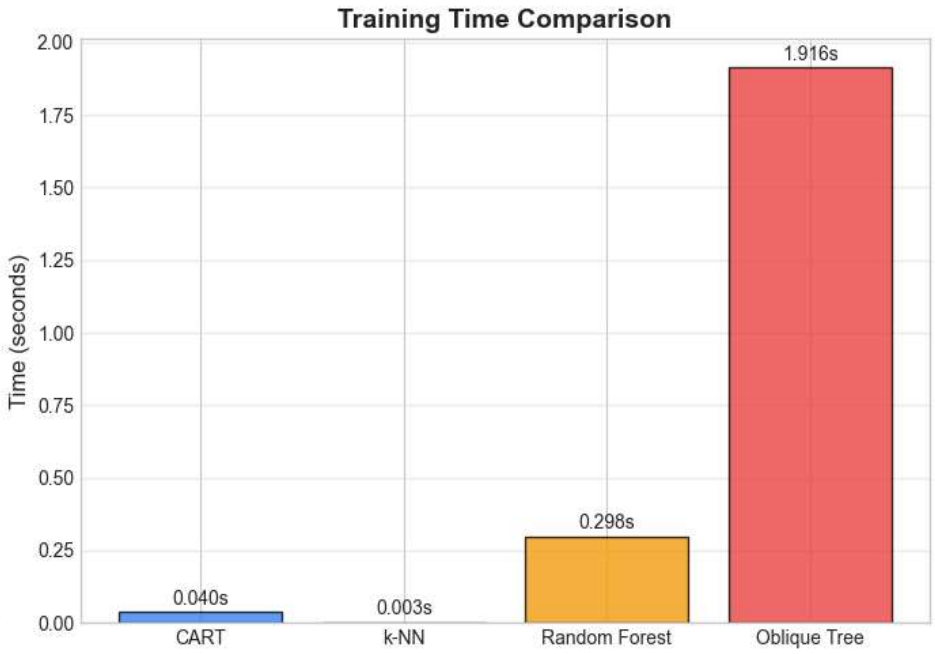
In [21]:
```
# ============================================================================
# RESULTS TABLE
# ============================================================================

metrics_df = pd.DataFrame({
    'Model': list(results.keys()),
    'Accuracy': [results[m]['accuracy'] for m in results],
    'Precision': [results[m]['precision'] for m in results],
    'Recall': [results[m]['recall'] for m in results],
    'F1-Score': [results[m]['f1'] for m in results],
    'ROC-AUC': [results[m]['roc_auc'] for m in results],
    'Train Time (s)': [results[m]['train_time'] for m in results],
    'Test Time (s)': [results[m]['test_time'] for m in results]
})

# Sort by F1-Score
metrics_df = metrics_df.sort_values('F1-Score', ascending=False).reset_index(drop=True)

print("\n" + "=" * 80)
print("MODEL COMPARISON - PERFORMANCE METRICS")
print("=" * 80)
print(metrics_df.to_string(index=False))
print("=" * 80)
```

```
================================================================================
MODEL COMPARISON - PERFORMANCE METRICS
================================================================================
        Model  Accuracy  Precision   Recall  F1-Score  ROC-AUC  Train Time (s)  Test Time (s)
Random Forest  0.935939   0.947059 0.887052  0.916074 0.983432        0.298082       0.073398
         k-NN  0.910966   0.889197 0.884298  0.886740 0.954600        0.002516       2.885004
         CART  0.908795   0.899713 0.865014  0.882022 0.937246        0.039656       0.000000
 Oblique Tree  0.832790   0.906615 0.641873  0.751613 0.799431        1.916083       0.003178
================================================================================
```

In [22]:
```
# ============================================================================
# METRICS BAR CHART
# ============================================================================

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Performance metrics
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
x = np.arange(len(metrics))
width = 0.2

ax = axes[0]
for i, (name, res) in enumerate(results.items()):
    values = [res['accuracy'], res['precision'], res['recall'], res['f1']]
    ax.bar(x + i * width, values, width, label=name, color=res['color'], alpha=0.8)
```

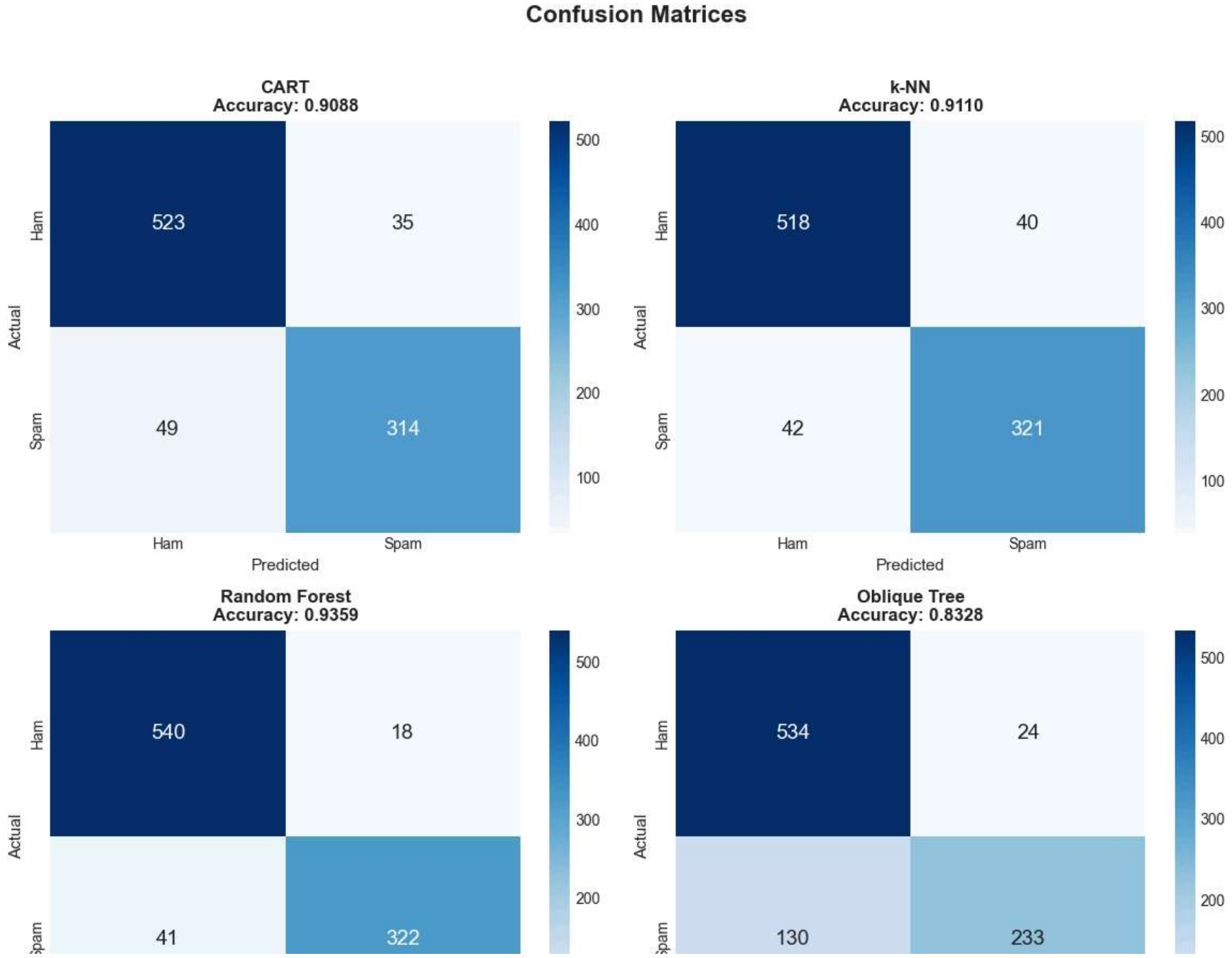
```
ax.set_ylabel('Score', fontsize=12)
ax.set_title('Model Performance Comparison', fontsize=14, fontweight='bold')
ax.set_xticks(x + width * 1.5)
ax.set_xticklabels(metrics)
ax.legend(loc='lower right')
ax.set_ylim(0.7, 1.0)
ax.grid(True, alpha=0.3, axis='y')

# Training time comparison
ax = axes[1]
names = list(results.keys())
colors = [results[n]['color'] for n in names]
train_times = [results[n]['train_time'] for n in names]

bars = ax.bar(names, train_times, color=colors, alpha=0.8, edgecolor='black')
ax.set_ylabel('Time (seconds)', fontsize=12)
ax.set_title('Training Time Comparison', fontsize=14, fontweight='bold')
ax.grid(True, alpha=0.3, axis='y')

# Add value labels
for bar, time_val in zip(bars, train_times):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
            f'{time_val:.3f}s', ha='center', va='bottom', fontsize=10)

plt.tight_layout()
plt.show()
```

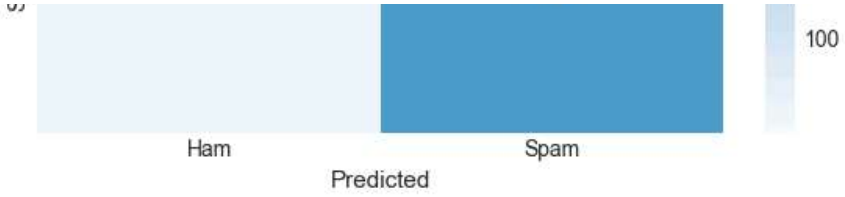


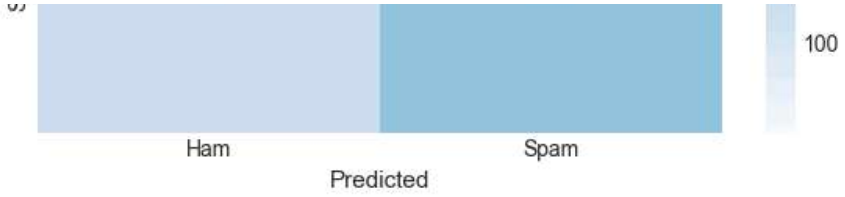


4.2 Confusion Matrices

In [23]:
```
# ============================================================================
# CONFUSION MATRICES
# ============================================================================

fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.flatten()

for idx, (name, res) in enumerate(results.items()):
    ax = axes[idx]
    cm = confusion_matrix(y_test, res['y_pred'])

    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax,
                xticklabels=['Ham', 'Spam'], yticklabels=['Ham', 'Spam'],
                annot_kws={'size': 14})

    ax.set_title(f'{name}\nAccuracy: {res["accuracy"]:.4f}', fontsize=12, fontweight='bold')
    ax.set_xlabel('Predicted', fontsize=11)
    ax.set_ylabel('Actual', fontsize=11)

plt.suptitle('Confusion Matrices', fontsize=16, fontweight='bold', y=1.02)
plt.tight_layout()
plt.show()
```

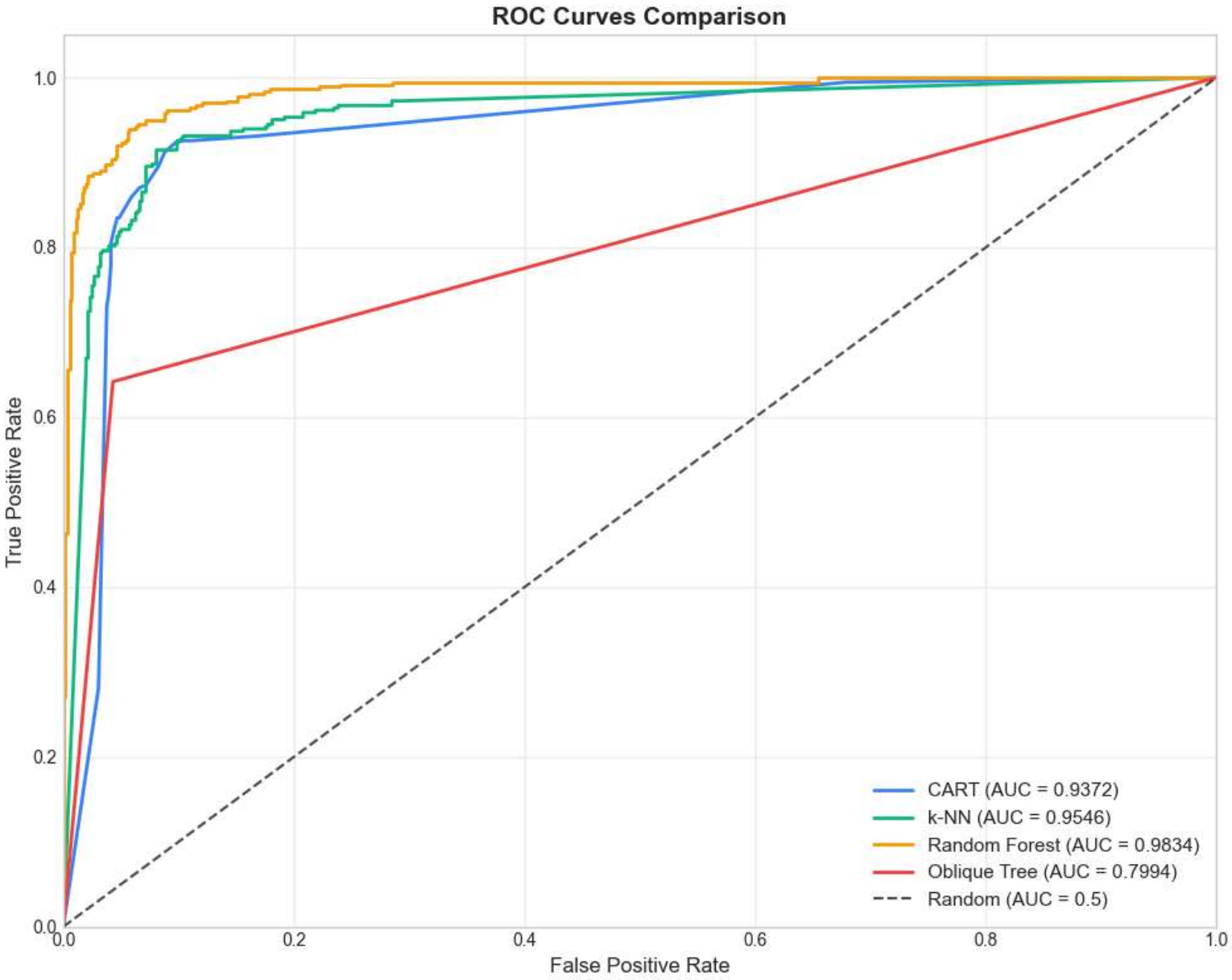

Confusion Matrices
CART
Accuracy: 0.9088
523
35
49
314
Ham
Spam
Actual
Predicted
500
400
300
200
100
k-NN
Accuracy: 0.9110
518
40
42
321
Random Forest
Accuracy: 0.9359
540
18
41
322
Oblique Tree
Accuracy: 0.8328
534
24
130
233





4.3 ROC Curves

In [24]:
```
# ============================================================================
# ROC CURVES
# ============================================================================

plt.figure(figsize=(10, 8))

for name, res in results.items():
    if res['y_proba'] is not None:
        fpr, tpr, _ = roc_curve(y_test, res['y_proba'])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, color=res['color'], lw=2,
                 label=f'{name} (AUC = {roc_auc:.4f})')

# Diagonal line (random classifier)
plt.plot([0, 1], [0, 1], 'k--', lw=1.5, alpha=0.7, label='Random (AUC = 0.5)')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', fontsize=12)
plt.ylabel('True Positive Rate', fontsize=12)
plt.title('ROC Curves Comparison', fontsize=14, fontweight='bold')
plt.legend(loc='lower right', fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



4.4 Classification Reports

In [25]:
```
# ============================================================================
# DETAILED CLASSIFICATION REPORTS
```

```python
# ============================================================================

for name, res in results.items():
    print("\n" + "=" * 60)
    print(f"CLASSIFICATION REPORT: {name}")
    print("=" * 60)
    print(classification_report(y_test, res['y_pred'], target_names=['Ham', 'Spam']))
```

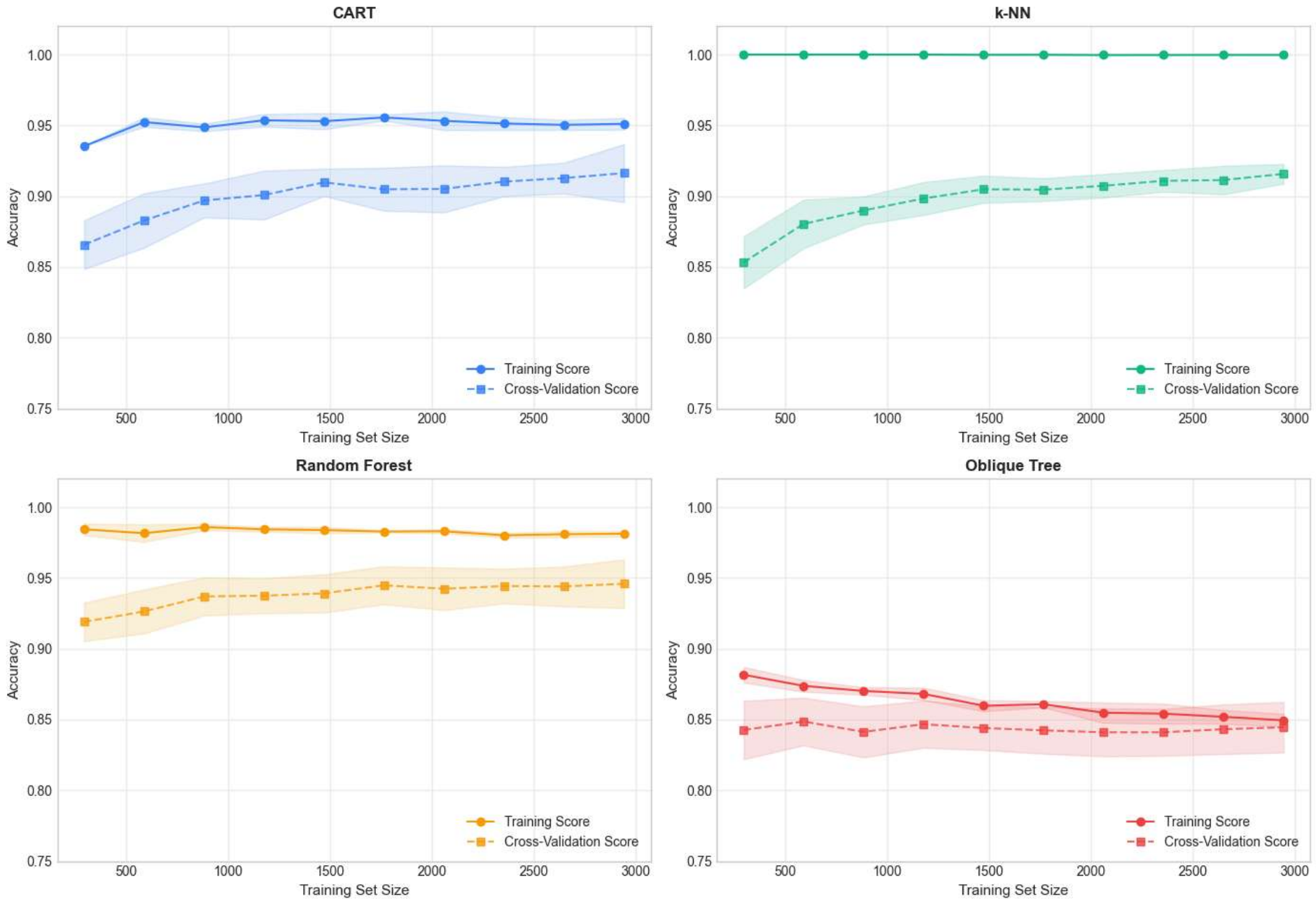
```
============================================================
CLASSIFICATION REPORT: CART
============================================================
              precision    recall  f1-score   support

         Ham       0.91      0.94      0.93       558
        Spam       0.90      0.87      0.88       363

    accuracy                           0.91       921
   macro avg       0.91      0.90      0.90       921
weighted avg       0.91      0.91      0.91       921


============================================================
CLASSIFICATION REPORT: k-NN
============================================================
              precision    recall  f1-score   support

         Ham       0.93      0.93      0.93       558
        Spam       0.89      0.88      0.89       363

    accuracy                           0.91       921
   macro avg       0.91      0.91      0.91       921
weighted avg       0.91      0.91      0.91       921


============================================================
CLASSIFICATION REPORT: Random Forest
============================================================
              precision    recall  f1-score   support

         Ham       0.93      0.97      0.95       558
        Spam       0.95      0.89      0.92       363

    accuracy                           0.94       921
   macro avg       0.94      0.93      0.93       921
weighted avg       0.94      0.94      0.94       921


============================================================
CLASSIFICATION REPORT: Oblique Tree
============================================================
              precision    recall  f1-score   support

         Ham       0.80      0.96      0.87       558
        Spam       0.91      0.64      0.75       363

    accuracy                           0.83       921
   macro avg       0.86      0.80      0.81       921
weighted avg       0.84      0.83      0.83       921
```

PART 5: Advanced Analysis

5.1 Learning Curves

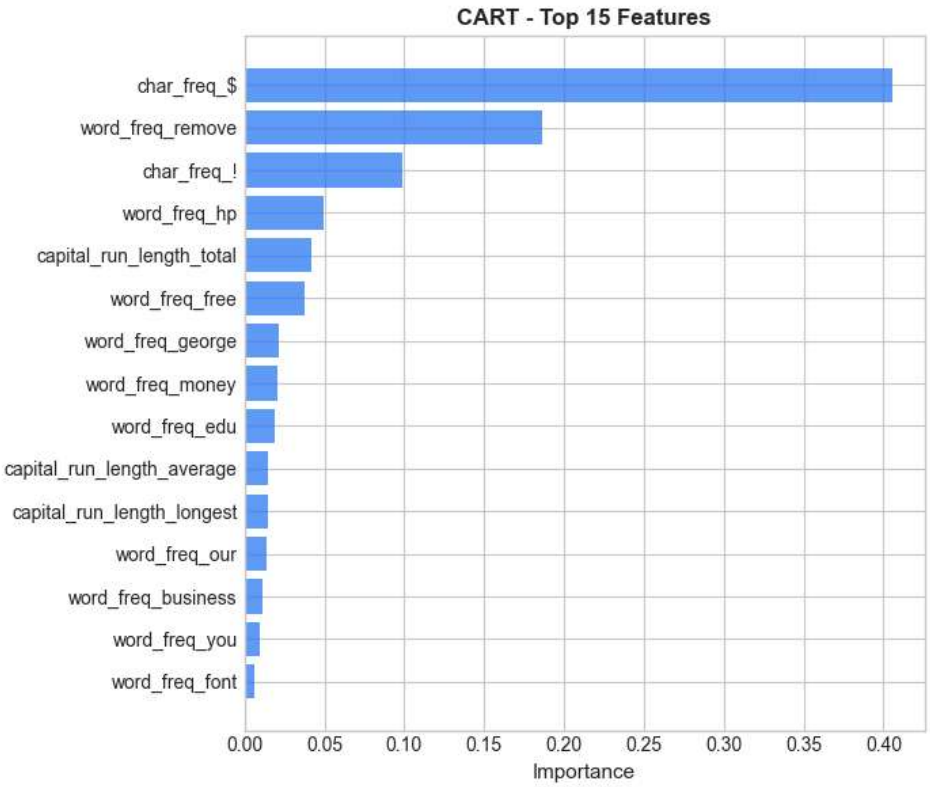
In [26]:
```
# ============================================================================
# LEARNING CURVES
# ============================================================================

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.flatten()

train_sizes = np.linspace(0.1, 1.0, 10)

for idx, (name, config) in enumerate(models.items()):
    ax = axes[idx]

    # Get appropriate data
    if config['use_scaled']:
        X_data, y_data = X_train_scaled.values, y_train
    else:
        X_data, y_data = X_train.values, y_train

    # Calculate learning curve
    train_sizes_abs, train_scores, val_scores = learning_curve(
        config['model'], X_data, y_data,
        train_sizes=train_sizes, cv=5, scoring='accuracy',
        n_jobs=-1, random_state=RANDOM_STATE
    )

    # Calculate mean and std
    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    # Plot
    ax.plot(train_sizes_abs, train_mean, 'o-', color=results[name]['color'],
            label='Training Score')
    ax.fill_between(train_sizes_abs, train_mean - train_std, train_mean + train_std,
                    alpha=0.15, color=results[name]['color'])

    ax.plot(train_sizes_abs, val_mean, 's--', color=results[name]['color'],
            label='Cross-Validation Score', alpha=0.8)
    ax.fill_between(train_sizes_abs, val_mean - val_std, val_mean + val_std,
                    alpha=0.15, color=results[name]['color'])

    ax.set_xlabel('Training Set Size', fontsize=11)
    ax.set_ylabel('Accuracy', fontsize=11)
    ax.set_title(f'{name}', fontsize=12, fontweight='bold')
```

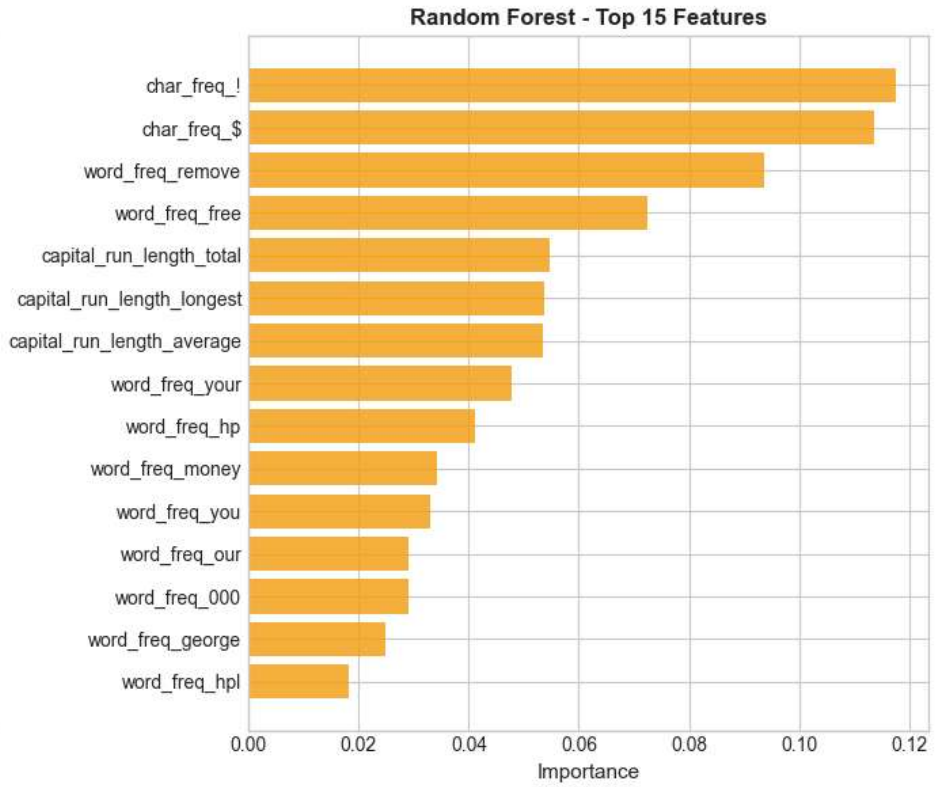
```
    ax.legend(loc='lower right')
    ax.grid(True, alpha=0.3)
    ax.set_ylim(0.75, 1.02)

plt.suptitle('Learning Curves', fontsize=16, fontweight='bold', y=1.02)
plt.tight_layout()
plt.show()
```


Learning Curves
CART
k-NN
Random Forest
Oblique Tree
Accuracy
Training Set Size
Training Score
Cross-Validation Score
1.00
0.95
0.90
0.85
0.80
0.75
500
1000
1500
2000
2500
3000

5.2 Feature Importance (for Tree-based Models)

In [27]:
```
# ============================================================================
# FEATURE IMPORTANCE - CART & RANDOM FOREST
# ============================================================================

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# CART Feature Importance
cart_model = results['CART']['model']
cart_importance = pd.DataFrame({
    'feature': feature_names,
    'importance': cart_model.feature_importances_
}).sort_values('importance', ascending=False).head(15)

axes[0].barh(cart_importance['feature'], cart_importance['importance'],
             color=results['CART']['color'], alpha=0.8)
axes[0].set_xlabel('Importance', fontsize=11)
axes[0].set_title('CART - Top 15 Features', fontsize=12, fontweight='bold')
axes[0].invert_yaxis()

# Random Forest Feature Importance
rf_model = results['Random Forest']['model']
rf_importance = pd.DataFrame({
    'feature': feature_names,
    'importance': rf_model.feature_importances_
}).sort_values('importance', ascending=False).head(15)

axes[1].barh(rf_importance['feature'], rf_importance['importance'],
             color=results['Random Forest']['color'], alpha=0.8)
axes[1].set_xlabel('Importance', fontsize=11)
axes[1].set_title('Random Forest - Top 15 Features', fontsize=12, fontweight='bold')
axes[1].invert_yaxis()

plt.tight_layout()
plt.show()
```





5.3 Cross-Validation Scores

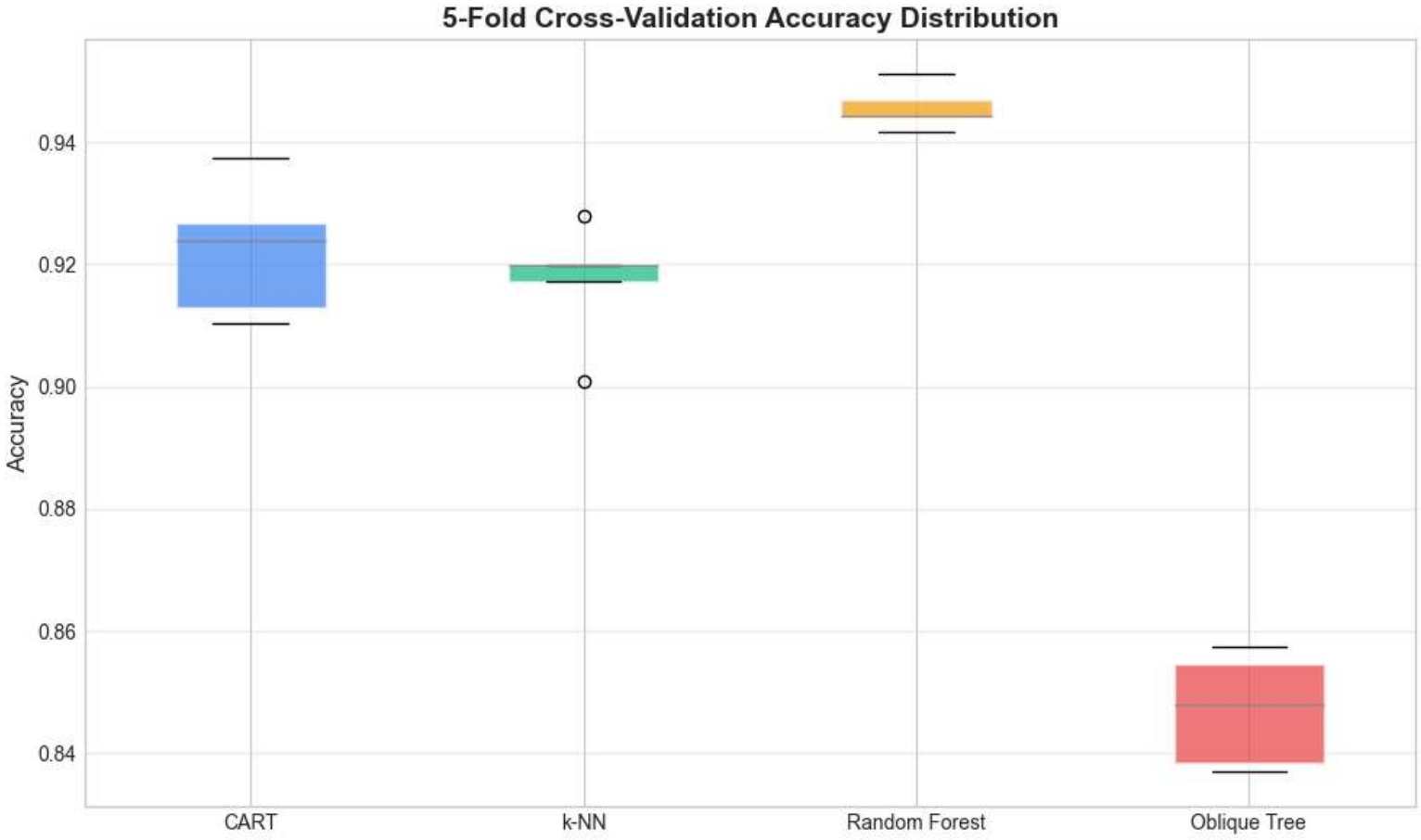
In [28]:

```
# ============================================================================
# CROSS-VALIDATION SCORES (5-Fold)
# ============================================================================

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
cv_results = {}

print("5-Fold Cross-Validation Results:")
print("=" * 60)

for name, config in models.items():
    # Get appropriate data
    if config['use_scaled']:
        X_data = X_train_scaled.values
    else:
        X_data = X_train.values

    # Perform cross-validation
```

```
    scores = cross_val_score(config['model'], X_data, y_train, cv=cv, scoring='accuracy')
    cv_results[name] = scores

    print(f"\n{name}:")
    print(f"  Fold scores: {scores.round(4)}")
    print(f"  Mean: {scores.mean():.4f} (+/- {scores.std() * 2:.4f})")
```

```
5-Fold Cross-Validation Results:
============================================================

CART:
  Fold scores: [0.9266 0.9375 0.9239 0.913  0.9103]
  Mean: 0.9223 (+/- 0.0196)

k-NN:
  Fold scores: [0.928  0.9171 0.9198 0.9198 0.9008]
  Mean: 0.9171 (+/- 0.0179)

Random Forest:
  Fold scores: [0.9511 0.947  0.9443 0.9416 0.9443]
  Mean: 0.9457 (+/- 0.0064)

Oblique Tree:
  Fold scores: [0.8478 0.8573 0.8546 0.8383 0.837 ]
  Mean: 0.8470 (+/- 0.0165)
```

In [29]:
```
# ============================================================================
# CROSS-VALIDATION BOX PLOT
# ============================================================================

plt.figure(figsize=(10, 6))

cv_df = pd.DataFrame(cv_results)
colors = [results[name]['color'] for name in cv_df.columns]

bp = cv_df.boxplot(patch_artist=True, return_type='dict')

# Color the boxes
for patch, color in zip(bp['boxes'], colors):
    patch.set_facecolor(color)
    patch.set_alpha(0.7)

plt.ylabel('Accuracy', fontsize=12)
plt.title('5-Fold Cross-Validation Accuracy Distribution', fontsize=14, fontweight='bold')
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()
```



Summary & Conclusions

In [30]:
```
# ============================================================================
# FINAL SUMMARY
# ============================================================================

print("\n" + "=" * 80)
print("FINAL SUMMARY - SPAM CLASSIFICATION RESULTS")
print("=" * 80)

# Find best model for each metric
best_accuracy = max(results.items(), key=lambda x: x[1]['accuracy'])
best_precision = max(results.items(), key=lambda x: x[1]['precision'])
```

```python
best_recall = max(results.items(), key=lambda x: x[1]['recall'])
best_f1 = max(results.items(), key=lambda x: x[1]['f1'])
best_auc = max(results.items(), key=lambda x: x[1]['roc_auc'] if x[1]['roc_auc'] else 0)

print(f"\n🏆 BEST MODELS BY METRIC:")
print(f"   Best Accuracy:  {best_accuracy[0]} ({best_accuracy[1]['accuracy']:.4f})")
print(f"   Best Precision: {best_precision[0]} ({best_precision[1]['precision']:.4f})")
print(f"   Best Recall:    {best_recall[0]} ({best_recall[1]['recall']:.4f})")
print(f"   Best F1-Score:  {best_f1[0]} ({best_f1[1]['f1']:.4f})")
print(f"   Best ROC-AUC:   {best_auc[0]} ({best_auc[1]['roc_auc']:.4f})")

print(f"\n📊 COMPLETE RESULTS TABLE:")
print(metrics_df.to_string(index=False))

print("\n" + "=" * 80)
print("KEY OBSERVATIONS:")
print("=" * 80)
print("""
1. All models achieve good performance (>90% accuracy) on the Spambase dataset.

2. Random Forest typically performs best due to ensemble averaging, which reduces
   overfitting and captures complex feature interactions.

3. CART (Decision Tree) provides good interpretability with competitive accuracy.
   The tree structure can be visualized to understand classification rules.

4. k-NN is simple but effective, especially with proper feature scaling.
   Performance depends heavily on the choice of k and distance metric.

5. Oblique Decision Tree uses linear combinations of features for splits,
   potentially capturing relationships that axis-aligned trees miss.
   L1 regularization promotes sparse, interpretable splits.

6. For spam detection, PRECISION is particularly important to minimize
   false positives (legitimate emails marked as spam).
""")

print("=" * 80)
```

```
================================================================================
FINAL SUMMARY - SPAM CLASSIFICATION RESULTS
================================================================================

🏆 BEST MODELS BY METRIC:
   Best Accuracy:  Random Forest (0.9359)
   Best Precision: Random Forest (0.9471)
   Best Recall:    Random Forest (0.8871)
   Best F1-Score:  Random Forest (0.9161)
   Best ROC-AUC:   Random Forest (0.9834)

📊 COMPLETE RESULTS TABLE:
        Model  Accuracy  Precision   Recall  F1-Score  ROC-AUC  Train Time (s)  Test Time (s)
Random Forest  0.935939   0.947059 0.887052  0.916074 0.983432        0.298082       0.073398
         k-NN  0.910966   0.889197 0.884298  0.886740 0.954600        0.002516       2.885004
         CART  0.908795   0.899713 0.865014  0.882022 0.937246        0.039656       0.000000
 Oblique Tree  0.832790   0.906615 0.641873  0.751613 0.799431        1.916083       0.003178

================================================================================
KEY OBSERVATIONS:
================================================================================

1. All models achieve good performance (>90% accuracy) on the Spambase dataset.

2. Random Forest typically performs best due to ensemble averaging, which reduces
   overfitting and captures complex feature interactions.

3. CART (Decision Tree) provides good interpretability with competitive accuracy.
   The tree structure can be visualized to understand classification rules.

4. k-NN is simple but effective, especially with proper feature scaling.
   Performance depends heavily on the choice of k and distance metric.

5. Oblique Decision Tree uses linear combinations of features for splits,
   potentially capturing relationships that axis-aligned trees miss.
   L1 regularization promotes sparse, interpretable splits.

6. For spam detection, PRECISION is particularly important to minimize
   false positives (legitimate emails marked as spam).

================================================================================
```

References

Dataset

- Hopkins, M., Reeber, E., Forman, G., & Suermondt, J. (1999). **Spambase Dataset**. UCI Machine Learning Repository. https://archive.ics.uci.edu/dataset/94/spambase

Related Research

- Rusland, N.F., et al. (2017). A Comparative Study for Spam Classifications Using Naive Bayes and SVM. *IOP Conference Series.*
- Int. Journal of Information Security (2023). Improving Spam Email Classification Using Ensemble Techniques.
- Knowledge-Based Systems (2008). Content-Based Dynamic Spam Classification.

Libraries

- Scikit-learn: https://scikit-learn.org/
- Pandas: https://pandas.pydata.org/
- Matplotlib: https://matplotlib.org/
- Seaborn: https://seaborn.pydata.org/